Microsoft® Macro Assembler Programmer's Guide

Microsoft®

Microsoft® Macro Assembler

Programmer's Guide

Version 6.0

For MS® OS/2 and MS-DOS® Operating Systems

Microsoft Corporation

Document No. LN06556-0291

10 9 8 7 6 5 4 3 2 1

Contents

PART 3 Advanced Topics

Appendixes

Figures and Tables

Figures

Tables

Introduction

The Microsoft® Macro Assembler *Programmer's Guide* provides the information you need to write and debug assembly-language programs with the Microsoft Macro Assembler (MASM), version 6.0. This book documents enhanced features of the language and the programming environment for MASM 6.0. It also describes new features that take advantage of the capabilities of the 80386/486 processors.

The *Programmer's Guide* is written for experienced programmers who know assembly language and are familiar with an assembler. The book does not teach the basics of assembly language; it does explain Microsoft-specific features. If you want to learn or review the basics of assembly language, refer to "Books for Further Reading" later in this introduction.

The documentation for MASM 6.0 is an integrated set, comprehensive and cohesive. This book emphasizes writing efficient code with the new and advanced features of MASM. *Installing and Using the Professional Development System* explains not only how to set up MASM 6.0 but also how to use the extensive online reference system, the Microsoft Advisor.

Installing and Using also introduces the integrated environment called the Programmer's WorkBench (PWB) and shows how to manage development projects with it. The Microsoft Macro Assembler *Reference* provides a full listing of all MASM instructions, directives, statements, and operators, and it serves as a quick reference to utility commands.

For more information on these same topics, see the online Microsoft Advisor, which is a complete reference to Macro Assembler language topics, to the utilities, and to PWB. You should be able to find most of the information you need in the Microsoft Advisor. The printed documents give more in-depth and background information.

New and Extended Features in MASM 6.0

Version 6.0 of MASM differs from version 5.1 in many ways, from optional extensions to features that replace or modify previous assembler behavior.

MASM 6.0 includes the Programmer's WorkBench, an integrated software development environment, and the CodeView® source-level debugger. From within PWB you can edit, build, debug, or run a program, and you can perform most of these operations with either menu selections or keyboard commands. You can also customize PWB to suit your individual programming and editing requirements and preferences.

New MASM Language Features

MASM 6.0 includes a number of new features, described in the list below, designed to make programming more efficient and intuitive and to increase your productivity. For example, MASM's new high-level-language features mean that you can get the speed of assembly language with the ease of high-level languages. You can also maintain your programs more easily.

- MASM 6.0 has many enhancements related to types. You can now use the same type specifiers in initializations as in other contexts (**BYTE** instead of **DB**). You can also define your own types, including pointer types, with the new **TYPEDEF** directive. See Chapter 3, "Using Addresses and Pointers," and Chapter 4, "Defining and Using Integers."
- The syntax for defining and using structures and records has been enhanced. You can also define unions with the new **UNION** directive. See Chapter 5, "Defining and Using Complex Data Types."
- MASM now generates complete CodeView information for all types. See Chapter 3, "Using Addresses and Pointers," and Chapter 4, "Defining and Using Integers."
- New control-flow directives let you use high-level-language constructs such as loops and if-then-else blocks defined with **.REPEAT** and **.UNTIL** (or **.UNTILCXZ**); **.WHILE** and **.ENDW**; and **.IF**, **.ELSE**, and **.ELSEIF**. The assembler generates the appropriate code to implement the control structure. See Chapter 7, "Controlling Program Flow."
- MASM now has more powerful features for defining and calling procedures. The extended **PROC** syntax for generating stack frames has been enhanced in version 6.0. You can also use the **PROTO** directive to prototype a procedure, which you can then call with the **INVOKE** directive. **INVOKE** automatically generates code to pass arguments (converting them to a related type, if appropriate) and make the call according to the specified calling convention. See Chapter 7, "Controlling Program Flow."
- MASM optimizes jumps by automatically determining the most efficient coding for a jump and then generating the appropriate code. See Chapter 7, "Controlling Program Flow."
- Maintaining multiple-module programs is easier in MASM 6.0. The **EXTERNDEF** and **PROTO** directives make it easy to maintain all global definitions in include files shared by all the source modules of a project. See Chapter 8, "Sharing Data and Procedures among Modules and Libraries."

The assembler has many new macro features that make complex macros clearer and easier to write:

- You can specify default values for macro arguments or mark arguments as required. And with the **VARARG** keyword, one parameter can accept a variable number of arguments.
- You can implement loops inside of macros in various ways. For example, the new **WHILE** directive expands the statements in a macro body while an expression is not zero.
- You can define macro functions, which return text macros. Several predefined text macros are also provided for processing strings. Macro operators and other features related to processing text macros and macro arguments have been enhanced. For more information on all these macro features, see Chapter 9, "Using Macros."

Finally, MASM 6.0 has improved customizable capabilities:

- With the new **.STARTUP** and **.EXIT** directives you can automatically generate appropriate start-up and exit code for DOS or OS/2 modules. See Chapter 2, "Organizing MASM Segments."
- MASM 6.0 supports flat memory model, available with OS/2 version 2.0. In flat model, segments can be as large as 4 gigabytes instead of 64K (kilobytes). Offsets are 32 bits instead of 16 bits. See Chapter 2, "Organizing MASM Segments."
- The program H2INC.EXE converts C include files to MASM include files and translates data structures and declarations. See Chapter 16, "Converting C Header Files to MASM Include Files."

MASM 6.0 includes many other minor new features as well as extended support for features of earlier versions of MASM. These features are listed in Appendix A, "Differences between MASM 6.0 and 5.1," with cross-references to the chapters where they are discussed in detail.

ML and MASM Command Lines

MASM 6.0 provides a new command-line driver, ML, which is more powerful and flexible than the previous driver (MASM). ML assembles and links with one command. The old MASM driver command syntax is still supported, however, to support existing batch files and makefiles that use MASM command lines.

NOTE The name MASM has traditionally been used to refer to the Microsoft Macro Assembler. It is used in that context throughout this book. But MASM also refers to MASM.EXE, which has been replaced by ML.EXE. In MASM 6.0, the MASM.EXE file is a small utility that translates command-line options to those accepted by ML.EXE, and then calls ML.EXE. The distinction between ML.EXE and MASM.EXE is made whenever necessary. Otherwise, MASM refers to the assembler and its features.

Compatibility with Earlier Versions of MASM

In many cases, MASM 5.1 code will assemble without modification under MASM 6.0. However, MASM 6.0 provides a new **OPTION** directive that lets you selectively modify the assembly process. In particular, you can use the **M510** argument with **OPTION** or the /Zm command-line option to set most features to be compatible with version 5.1 code.

See Appendix A, "Differences between MASM 6.0 and 5.1," for information about obsolete features that will not assemble correctly under MASM 6.0. The appendix also discusses how to update code to use the new features.

Scope and Organization of this Book

The *Programmer's Guide* describes how to get the most out of the Microsoft Macro Assembler 6.0 and the Programmer's WorkBench. The book is arranged by topic, with each topic answering a question or solving a problem. The last section in each chapter lists topics in the online reference system that provide additional information.

The *Programmer's Guide* is divided into three parts:

Part 1, "Programming in Assembly Language," explains how to program efficiently using both the new and old features of MASM. It reviews the basic components of assembly language and also describes the new and enhanced features.

Part 2, "Improving Programmer Productivity," introduces the utility programs included with MASM 6.0. These programs can help you program more quickly and efficiently. For example, the chapters in Part 2 show you how to automatically update your project (Chapter 10), use program lists as input (Chapter 11), use the Microsoft linker (LINK) (Chapter 12), write module-definition files (Chapter 13), customize PWB to suit your programming style (Chapter 14), use the CodeView debugger to record and play back a debugging session (Chapter 15), and easily port data structures from C programs to MASM programs (Chapter 16).

Part 3, "Advanced Topics," covers specialized areas. It describes how to write programs to run under OS/2 (Chapter 17) and how to build dynamic-link libraries (Chapter 18). Chapter 19 shows how to write a terminate-and-stay-resident (TSR) program. Chapter 20, on mixed-language programming, defines the calling conventions and equivalent data types that allow MASM to call and be called by C, FORTRAN, Basic, and Pascal.

In addition, six appendixes and a glossary detail the features of MASM 6.0. Of particular interest are Appendix A, "Differences between MASM 6.0 and 5.1," and Appendix B, "BNF Grammar." Appendix A lists the new features of MASM 6.0 and also explains how to update MASM 5.1 code. The BNF grammar, or Backus-Naur Form for grammar notation, lets you determine the exact syntax for

any MASM language component. It clearly defines recursive definitions and shows all the available options for any placeholder. Other appendixes cover assembly listings, reserved words, default segment names, and error messages.

Books for Further Reading

The following books may help you learn to program in assembly language or write specialized programs. These books are listed only for your convenience. Microsoft makes no specific recommendations concerning any of these books.

Books about Programming in Assembly Language

Abrash, Michael, *Zen of Assembly Language*. Glenview, IL: Scott, Foresman and Co., 1990.

Duntemann, Jeff, *Assembly Language from Square One: For the PC AT and Compatibles*. Glenview, IL: Scott, Foresman and Co., 1990.

Fernandez, Judi N., and Ashley, Ruth, *Assembly Language Programming for the 80386*. New York: McGraw-Hill, 1990.

Miller, Alan R., *DOS Assembly Language Programming*. San Francisco: SYBEX, 1988.

Scanlon, Leo J., *80286 Assembly Language Programming on MS-DOS Computers*. New York: Brady Communications, 1986.

Turley, James L., *Advanced 80386 Programming Techniques*. Berkeley, CA: Osborne McGraw-Hill, 1988.

Books about DOS and BIOS

"Article 11." *MS-DOS Encyclopedia*. Redmond, WA: Microsoft Press, 1988. Contains information about terminate-and-stay-resident programs.

Duncan, Ray, *Advanced MS-DOS*. 2nd ed. Redmond, WA: Microsoft Press, 1988.

Jourdain, Robert, *Programmer's Problem Solver for the IBM PC, XT and AT*. New York: Brady Communications, 1986.

Microsoft MS-DOS Programmer's Reference. Redmond, WA: Microsoft Press, 1986-87.

Norton, Peter and Wilton, Richard, *The New Peter Norton Programmer's Guide to the IBM PC and PS/2*. Redmond, WA: Microsoft Press, 1988.

Wilton, Richard, *Programmer's Guide to PC & PS/2 Video Systems*. Redmond, WA: Microsoft Press, 1987.

Books about OS/2

Duncan, Ray, *Advanced OS/2 Programming*. Redmond, WA: Microsoft Press, 1989.

———, *Essential OS/2 Functions*. Redmond, WA: Microsoft Press, 1989.

Letwin, Gordon, *Inside OS/2*. Redmond, WA: Microsoft Press, 1989.

OS/2 Programmer's Reference. 4 vols. Redmond, WA: Microsoft Press, 1989.

Books about Other Topics

Nelson, Ross P., *The 80386 Book*. Redmond, WA: Microsoft Press, 1988.

Startz, Richard, *8087/80287/80387 for the IBM PC and Compatibles*. Bowie, MD: Robert J. Brady Co., 1988.

Writing ROMable Code in Microsoft C. Costa Mesa, CA: SSI Corporation.

Document Conventions

The following document conventions are used throughout this manual:

Example of Convention	Description
SAMPLE2.ASM	Uppercase letters indicate file names, segment names, registers, and terms used at the command level.
.MODEL	Boldface type indicates assembly-language directives, instructions, type specifiers, and predefined macros, as well as keywords in other programming languages.
placeholders	Italic letters indicate placeholders for information you must supply, such as a file name. Italics are also occasionally used for emphasis in the text.
`target`	This font is used to indicate example programs, user input, and screen output.
;	A semicolon in the first column of an example signals illegal code. A semicolon also marks a comment.

SHIFT	Small capital letters signify names of keys on the keyboard. Notice that a plus (+) indicates a combination of keys. For example, CTRL+E means to hold down the CTRL key while pressing the E key.
[[*argument*]]	Items inside double square brackets are optional.
{*register*|*memory*}	Braces and a vertical bar indicate a choice between two or more items. You must choose one of the items unless double square brackets surround the braces.
Repeating elements...	A horizontal ellipsis (...) following an item indicates that more items having the same form may appear.

`Program`
.
.
.
`Fragment`
	A vertical ellipsis tells you that part of a program has been intentionally omitted.

Getting Assistance and Reporting Problems

If you need help or think you have discovered a problem in the software, please provide the following information to help us locate the problem:

- The version of DOS or OS/2 that you are running
- Your system configuration: the type of machine you are using, its total memory, and its total free memory at assembler execution time, as well as any other information you think might be useful
- The assembly command line used, or the link command line if the problem occurred during linking
- Any object files or libraries you linked with if the problem occurred at link time

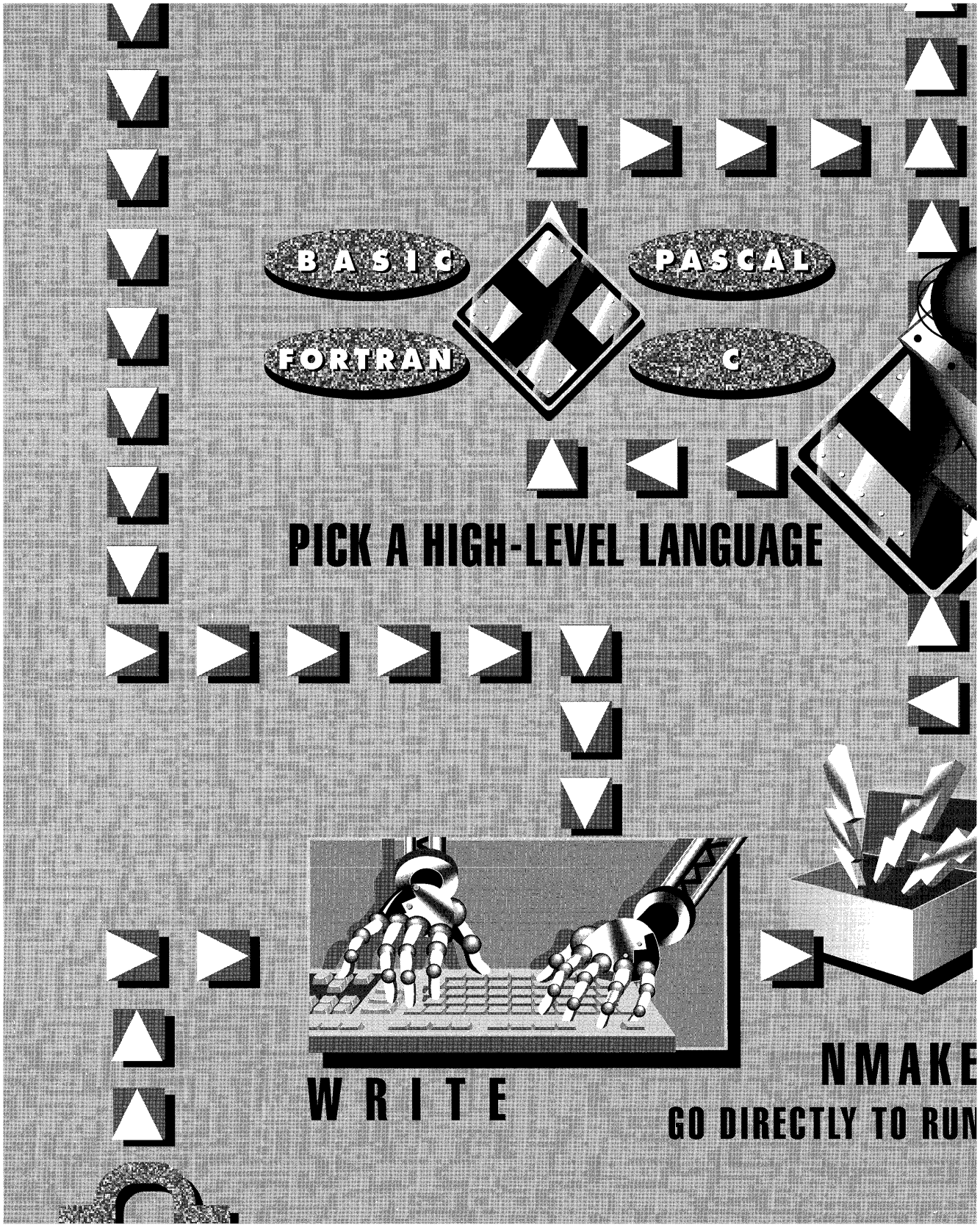
If your program is very large, please try to reduce its size to the smallest possible program that still produces the problem.

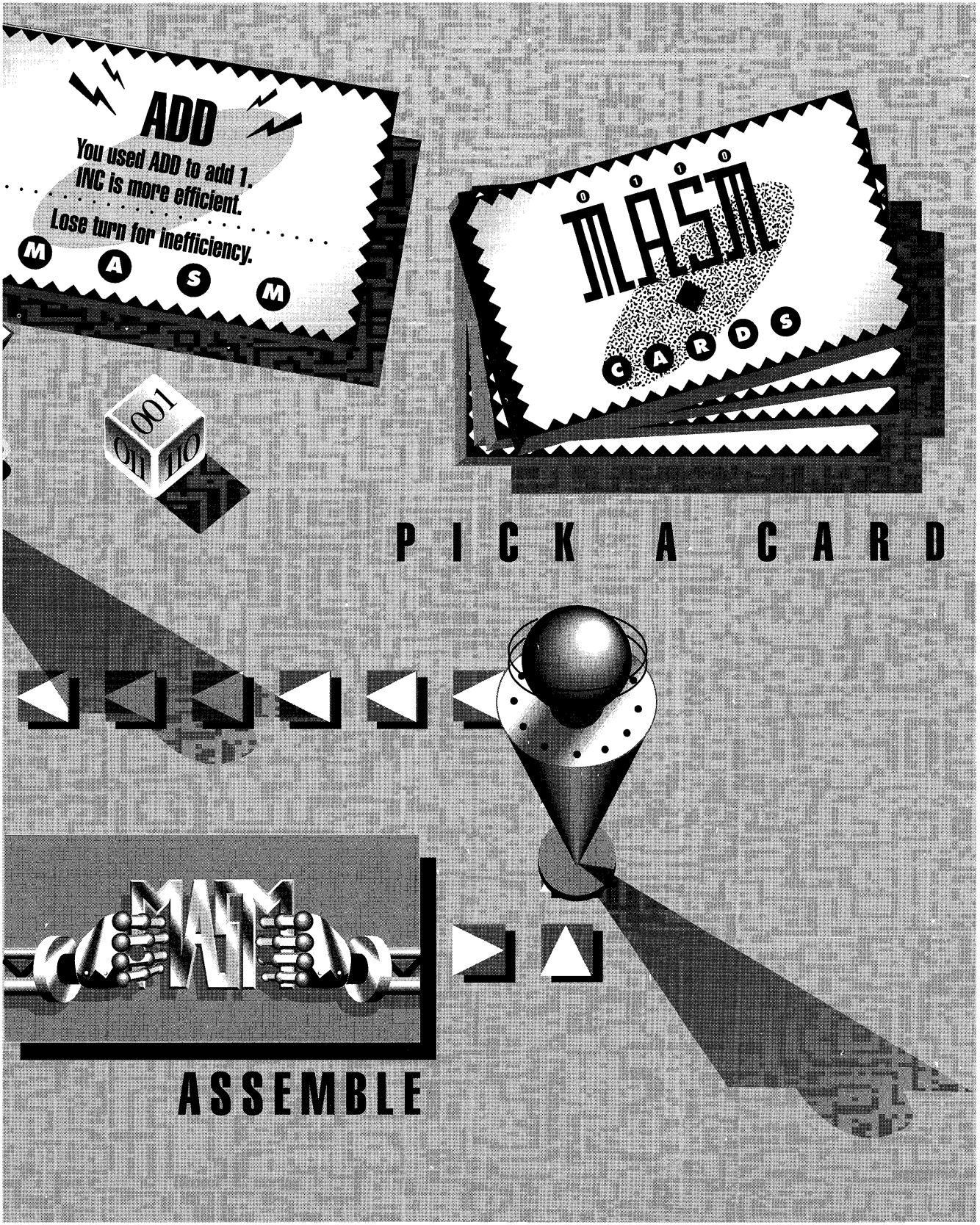
Use the Product Assistance Request form at the back of this book to send this information to Microsoft. If you have comments or suggestions regarding any of the books accompanying this product, please indicate them on the Document Feedback Card at the back of this book.

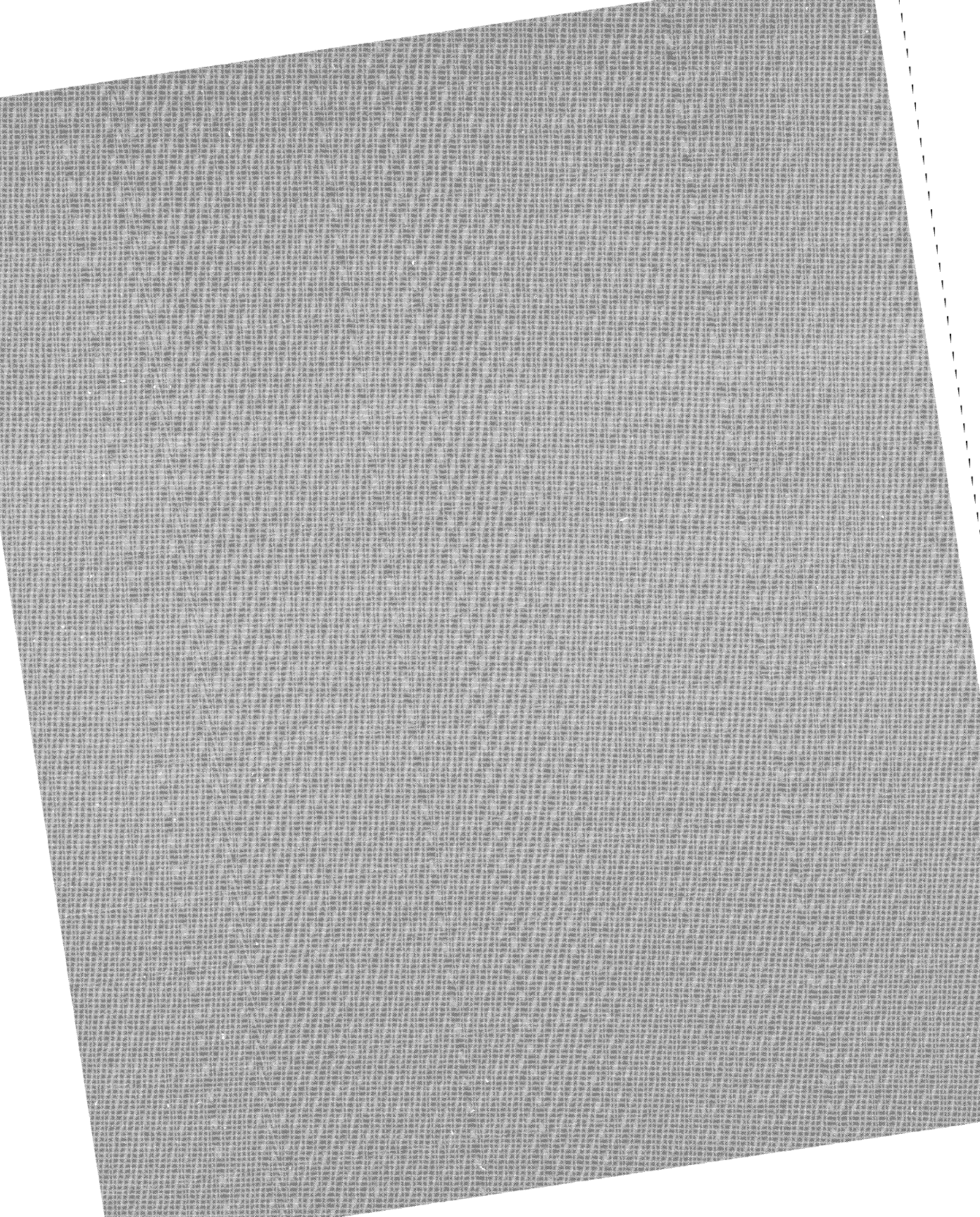
If you are not a registered Macro Assembler owner, you should fill out and return the Registration Card. This enables Microsoft to keep you informed of updates and other information about the assembler.

Part 1
Programming in Assembly Language

Chapters


BASIC
PASCAL
FORTRAN
C
PICK A HIGH-LEVEL LANGUAGE
WRITE
NMAKE
GO DIRECTLY TO RUN


ADD
You used ADD to add 1.
INC is more efficient.
Lose turn for inefficiency.
M A S M
001
011 110
MASM
CARDS
PICK A CARD
MAM
ASSEMBLE



Chapter 1

Understanding Global Concepts

With the development of the Microsoft Macro Assembler (MASM) version 6.0, you now have more options available to you for approaching a programming task. This chapter explains the general concepts of programming in assembly language, beginning with the environment and reviewing the components you need to work in the assembler environment. Even if you are familiar with previous versions of MASM, you should examine this chapter for information on new terms and features.

The first section of the chapter takes a look at the available processors and operating systems and how they work together. It also discusses the relationship of segmented architecture to assembly programming and the differences it makes for programming in OS/2 rather than in DOS.

The second section describes some of the language components of MASM that are common to most programs, such as reserved words, constant expressions, operators, and registers. The rest of this book assumes that you understand the information presented in this section.

The last section summarizes the assembly process, from assembling a program through running it. You can affect this process by the way you develop your code. Finally, this section explores how you can change the assembly process with the **OPTION** directive and conditional assembly.

NOTE This manual does not cover information specific to programming for Microsoft Windows™. For information on this, see the Microsoft Windows Software Development Kit.

1.1 The Processing Environment

The processing environment for MASM 6.0 includes the processor on which your programs run, the operating system your programs will use, and the aspects of the segmented architecture that influence the choice of programming models. This section summarizes these elements of the environment and how they affect your programming choices.

1.1.1 8086-Based Processors

The 8086 "family" of processors uses segments to control data and code. The later 8086-based processors have larger instruction sets and more memory capacity, but they still use the same segmented architecture. Knowing the differences

between the various 8086-based processors can help you select the target processor for your programs.

The instruction set of the 8086 processor is upwardly compatible with its successors. To write code that runs on the widest number of machines, select the 8086 instruction set. By choosing to use the instruction set of a more advanced processor, you increase the capabilities and efficiency of your program, but you also reduce the number of systems on which the program can run.

Table 1.1 lists modes, memory, and segment size of processors on which your application may need to run. Each processor is discussed in more detail below.

Table 1.1 8086 Family of Processors

Processor	Available Modes	Addressable Memory	Segment Size
8086/8088	Real	1 megabyte	16 bit
80186/80188	Real	1 megabyte	16 bit
80286	Real and Protected	16 megabytes	16 bit
80386	Real and Protected	4 gigabytes	16 or 32 bit
80486	Real and Protected	4 gigabytes	16 or 32 bit

Processor Modes Real mode allows only one process to run at a time. The DOS operating system runs in real mode. The OS/2 operating system can execute programs written for DOS, but is designed to provide capabilities available only in protected mode. In protected mode, more than one process can be active at any one time. Memory accessed by these different processes is protected from access by another process.

Protected-mode addresses do not correspond directly to physical memory. Under protected-mode operating systems, the processor allocates and manages memory dynamically. Additional privileged instructions initialize protected mode and control multiple processes. Section 1.1.2 provides more information on operating systems.

8086 and 8088 The 8086 is faster than the 8088 because of its 16-bit data bus; the 8088 has only an 8-bit data bus. The 16-bit data bus allows you to use **EVEN** and **ALIGN** on an 8086 processor to word-align data and thus improve data-handling efficiency. Memory addresses on the 8086 and 8088 refer to actual physical addresses.

80186 and 80188 These two processors are identical to the 8086 and 8088 except that new instructions have been added and several old instructions have been optimized. These processors run significantly faster than the 8086.

80286 The 80286 processor adds some instructions to control protected mode, and it runs faster. It also provides the optional protected mode that can be used by the operating system to allow multiple processes to run at the same time. The 80286 is the minimum for running 16-bit versions of OS/2.

80386 Unlike its predecessors, the 80386 processor can handle both 16-bit and 32-bit data. It is fully software-compatible with the 80286. It implements many new hardware-level features, including virtual paged memory, multiple virtual 8086 processes, addressing of up to four gigabytes of memory, and specialized debugging registers.

Under DOS, the 80836 supports all the instructions of the 80286 as well as several additional ones. It also allows limited use of 32-bit registers and addressing modes. The 80386 operates at faster processor speeds than the 80286 and is the minimum for running 32-bit versions of OS/2 and other 32-bit operating systems.

80486 The 80486 processor is an enhanced version of the 80386, with instruction "pipelining" that executes many instructions two to three times faster. It incorporates an enhanced version of the 80387 coprocessor, as well as an 8K (kilobyte) memory cache. The 80486 includes several new instructions and is fully compatible with 80386 software.

8087, 80287, and 80387 These math coprocessors work concurrently with the 8086 family of processors. Performing floating-point calculations with math coprocessors is up to 100 times faster than emulating the calculations with integer instructions. Although there are technical and performance differences among the three coprocessors, the main difference to the applications programmer is that the 80287 and 80387 can operate in protected mode. The 80387 also has several new instructions. The 80486 does not use any of these coprocessors; its floating-point processor is built in and is functionally equivalent to the 80387.

1.1.2 Operating Systems

With MASM, you can create programs that run under DOS, Windows, or OS/2—or all three, in some cases. For example, ML.EXE can produce executable files that run in any of the target environments, regardless of the programmer's environment. For information on building programs for different environments, see "Building and Running Programs" in PWB's online help.

DOS and OS/2 provide different processing modes. DOS uses the single-process real mode. OS/2 uses the multiple-process protected mode. While OS/2 can also run in real mode, this book assumes it is being used in protected mode.

DOS and OS/2 differ primarily in system access methods, size of addressable memory, and segment selection. Table 1.2 summarizes these differences.

Table 1.2 The DOS and OS/2 Operating Systems

Operating System	System Access	Available Active Processes	Addressable Memory	Contents of Segment Register	Word Length
DOS (and OS/2 1.*x* real mode)	Direct to hardware	One	1 megabyte	Actual address	16 bit
OS/2 1.*x* protected mode	Operating system call	Multiple	16 megabytes	Segment selectors	16 bit
OS/2 2.*x*	Operating system call	Multiple	4 gigabytes	Segment selectors	32 bit

DOS In real-mode programming, you can access system functions by calling DOS, calling the basic input/output system (BIOS), or directly addressing hardware. Access is through DOS interrupt 21h.

Protected-mode programs cannot directly access hardware ports.

OS/2 1.*x* As you can see in Table 1.2, protected mode allows for much larger data structures than real mode, since the addressable memory is extended to 16 megabytes. In protected mode, segment registers contain segment selectors rather than actual segment values. These selectors cannot be calculated by the program; they must be obtained by calling the operating system. Programs that attempt to calculate segment values or to address memory directly do not work.

Note that protected-mode operating systems such as XENIX® and OS/2 provide system functions for memory and hardware accesses that would be prohibited with direct processor commands. This software interface permits access without the possibility of corrupting memory or crashing the system.

Protected mode uses privilege levels to maintain system integrity and security. Programs cannot access data or code that is in a higher privilege level. Some instructions that directly access ports or clear interrupts (such as **CLI**, **STI**, **IN**, and **OUT**) are available at privilege levels normally used only by systems programmers.

OS/2 protected mode enforces the separation of segment values. The segments have selectors that have no relationship to the offset. The operating system combines the segment and offset so that your programs can address up to 16 megabytes of virtual memory in a 16-bit system.

OS/2 2.*x* and flat model eliminate segments.

OS/2 2.*x* OS/2 2.*x* uses an unsegmented architecture. (See Section 1.1.3.) It creates a "flat model" in which the entire address space is within one 32-bit segment. Section 2.2.1, "Defining Basic Attributes with .MODEL," explains how to use the flat model. In a 32-bit system, you can access up to four gigabytes of virtual memory. (The term "virtual memory" means that if the programs running under OS/2 request more memory than is physically available, part of the

memory is temporarily swapped out to disk.) Since code, data, and stack are in the same segment, the value of segment registers never needs to change. Internal mechanisms of OS/2 2.*x* implement protection at a lower level.

1.1.3 Segmented Architecture

The 8086 processors differ from many other microprocessors in that they use a segmented architecture: that is, each address is represented in two parts—a segment and an offset. Segmented addresses affect many aspects of assembly-language programming, especially addresses and pointers.

Only 64K of data can be addressed by a 16-bit segment address.

Segmented architecture was originally designed to enable a 16-bit processor to access an address space larger than 64K. (Section 1.1.5, "Segmented Addressing," explains how the processor uses both the segment and offset to create addresses larger than 64K.) DOS is an example of an operating system that uses segmented architecture on a 16-bit processor.

With the advent of protected-mode processors such as the 80286, segmented architecture gained a second purpose. Segments can separate different blocks of code and data to protect them from undesirable interactions. OS/2 1.*x* is an operating system that takes advantage of the protection features of the 16-bit segments on the 80286.

Segmented architecture went through another significant change with the release of 32-bit processors, starting with the 80386. These processors are backward compatible with the older 16-bit processors, but they also offer a 32-bit mode that minimizes the memory limitations of a 16-bit segmented architecture. Both offer paging to maintain segment protection. XENIX 386 is an example of a 32-bit segmented operating system using segment protection.

OS/2 2.*x* takes advantage of the 32-bit processors to allow a nonsegmented memory configuration. The processor still uses 32-bit segments, but from the user's viewpoint, there is only one segment. The flat memory model used by OS/2 2.*x* places code and data in a single segment. See Section 2.2.1, "Defining Basic Attributes with .MODEL," for more information about the flat memory model.

1.1.4 Segment Protection

Segmented architecture is an important part of the OS/2 memory-protection scheme. In a "multitasking" operating system where numerous programs can run simultaneously, programs must not access the code and data of another process without permission.

In DOS, the data and code segments are usually allocated adjacent to each other, as shown in Figure 1.1. In OS/2, the data and code segments may be anywhere in memory. The programmer knows nothing about their location and has no control

over it. The segments may even be moved to a new memory location or swapped to disk while the program is running.

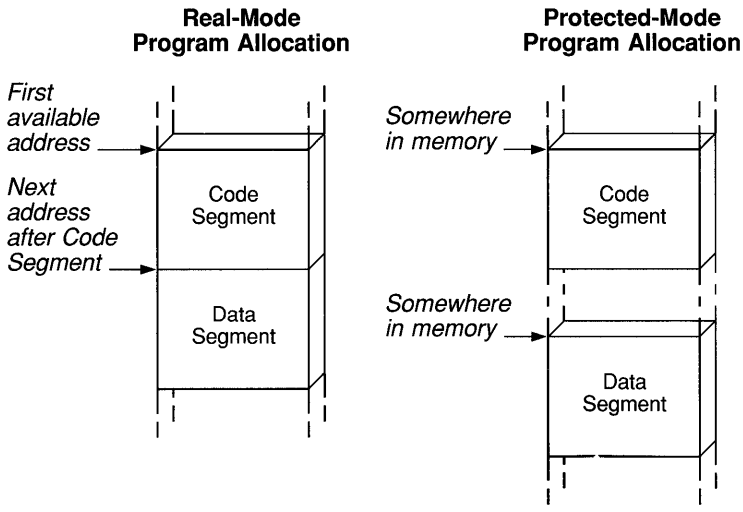


Figure 1.1 Segment Allocation

Segment protection prevents a bug in one program from corrupting another program.

Segment protection makes software development easier and more reliable in OS/2 than in DOS because, in OS/2, any illegal access is detected immediately. The operating system intercepts illegal memory accesses, terminates the program, and displays a message. This makes the bug easier to track down and fix.

In DOS, an illegal access is not detected and may not cause an error until later, when another part of the program attempts to use the corrupted memory.

1.1.5 Segmented Addressing

Segmented addressing is the internal mechanism that combines a segment value and an offset value to create an address. The two parts of an address are represented as

segment:*offset*

The *segment* portion is always a 16-bit value. The *offset* portion is a 16-bit value in 16-bit mode or a 32-bit value in 32-bit mode.

In real mode, the segment value is a physical address that has an arithmetic relationship to the offset value. The segment and offset together create a 20-bit physical address (explained in the next section). Although 20-bit addresses can access up to one megabyte of memory, the operating system on IBM® PCs and compatibles uses part of this memory, leaving 640K of memory for programs.

1.1.6 Segment Arithmetic

Manipulating segment and offset addresses directly in real-mode programming is called "segment arithmetic." Programs that perform segment arithmetic are not portable to protected-mode operating systems, where addresses do not correspond to a known segment and offset.

The segment selects a region of memory; the offset selects the byte within that region.

To perform segment arithmetic successfully, it helps to understand how the processor combines a 16-bit segment and a 16-bit offset to form a 20-bit linear address. In effect, the segment selects a 64K region of memory, and the offset selects the byte within that region. Here's how it works:

1. The processor shifts the segment address to the left by four binary places, producing a 20-bit address ending in four zeros. This operation has the effect of multiplying the segment address by 16.
2. The processor adds this 20-bit segment address to the 16-bit offset address. The offset address is not shifted.
3. The processor uses the resulting 20-bit address, often called the "physical address," to access an actual location in the one-megabyte address space.

Figure 1.2 illustrates this process.

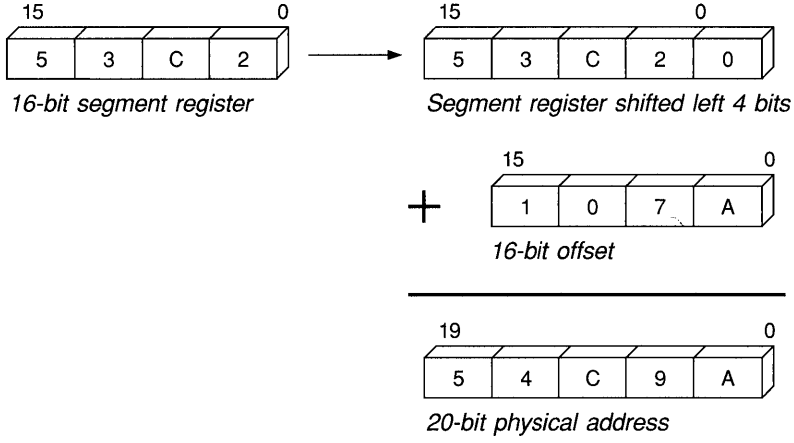


Figure 1.2 Calculating Physical Addresses

A 20-bit physical address may actually be specified by 4,096 equivalent *segment*:*offset* addresses. For example, the 20-bit physical address 0F800 is equivalent to 0000:F800, 0F00:0800, or 0F80:0000.

You may need to convert two segmented addresses with different segments to segmented addresses with the same segment to write TSRs (see Chapter 19), to write code to handle huge arrays, or to determine the size of an area of memory.

1.2 Language Components of MASM

Programming with MASM requires that you understand the MASM concepts of reserved words, identifiers, predefined symbols, constants, expressions, operators, data types, registers, and statements. This section defines important terms and provides lists that summarize these topics. See online help or the MASM *Reference* for detailed information.

1.2.1 Reserved Words

A reserved word has a special meaning fixed by the language. You can use it only under certain conditions. MASM's reserved words include:

- Instructions, which correspond to operations the processor can execute
- Directives, which give commands to the assembler
- Attributes, which provide a value for a field, such as segment alignment
- Operators, which are used in expressions
- Predefined symbols, which return information to your program

MASM reserved words are not case sensitive except for predefined symbols (see Section 1.2.3).

Use OPTION NOKEYWORD if you want to use a reserved word in another context.

The assembler generates an error if you use a reserved word as a variable, code label, or other identifier within your source code. However, if you need to use a reserved word for another purpose, the **OPTION NOKEYWORD** directive can selectively disable a word's status as a reserved word.

For example, to remove the **STR** instruction, the **MASK** operator, and the **NAME** directive from the set of words MASM recognizes as reserved, use this statement in the code segment of your program prior to the first reference to **STR**, **MASK**, or **NAME**:

```
OPTION NOKEYWORD:<STR MASK NAME>
```

The **OPTION** directive is discussed in Section 1.3.2. Appendix D provides a complete list of MASM reserved words.

1.2.2 Identifiers

Identifiers are names of variables of a given type.

An identifier is a name that you invent and attach to a definition. Identifiers can be symbols representing variables, constants, procedure names, code labels, segment names, and user-defined data types such as structures, unions, records, and types defined with **TYPEDEF**. Identifiers longer than 247 characters generate an error.

Certain restrictions limit the names you can use for identifiers. Follow these rules to define a name for an identifier:

- The first character of the identifier can be an alphabetic character (A–Z) or any of these four characters: @ _ $?
- The other characters in the identifier can be any of the characters listed above or a decimal digit (0–9)

Avoid starting an identifier with the at sign (@), because MASM 6.0 predefines some special symbols starting with @ (see Section 1.2.3). Beginning an identifier with @ may also cause conflicts with future versions of the Macro Assembler.

The symbol—and thus the identifier—is visible as long as it remains within scope. (See Section 8.2, "Sharing Symbols with Include Files," for additional information about visibility and scope.)

1.2.3 Predefined Symbols

Macros and conditional-assembly blocks often use predefined symbols.

The assembler includes a number of predefined symbols (also called predefined equates). You can use these symbol names at any point in your code to represent the equate value. For example, the predefined equate **@FileName** represents the base name of the current file. If the current source file is TASK.ASM, the value of **@FileName** is TASK. The MASM predefined symbols are listed below according to the kinds of information they provide. Case is important only if the /Cp option is used. (See online help on ML command-line options for additional details.)

Predefined Symbols for Segment Information

Symbol	Description
@code	Provides the name of the code segment, except in tiny model when it returns DGROUP.
@CodeSize	Returns an integer representing the default code distance.
@CurSeg	Returns the name of the current segment.
@data	Expands to DGROUP except in flat model.
@DataSize	Returns an integer representing the default data distance.

Symbol	Description
@fardata	Represents the name of the segment defined by the **.FARDATA** directive.
@fardata?	Represents the name of the segment defined by the **.FARDATA?** directive.
@Model	Returns the selected memory model.
@stack	Expands to DGROUP for near stacks or STACK for far stacks. (See Section 2.2.3, "Creating a Stack.")
@WordSize	Provides the size attribute of the current segment.

Predefined Symbols for Environment Information

Symbol	Description
@Cpu	Contains a bit mask specifying the processor mode.
@Environ	Returns values of environment variables.
@Interface	Contains information about the language parameters.
@Version	Represents the text equivalent of the MASM version number. In MASM 6.0, this expands to 600.

Predefined Symbols for Date and Time Information

Symbol	Description
@Date	Supplies the current system date.
@Time	Supplies the current system time.

Predefined Symbols for File Information

Symbol	Description
@FileCur	Names the current file (base and suffix).
@FileName	Names the base name of the main file being assembled as it appears on the command line.
@Line	Gives the source line number in the current file.

Predefined Functions for Macro String Manipulation

Symbol	Description
@CatStr	Returns concatenation of two strings.
@InStr	Returns the starting position of a string within another string.
@SizeStr	Returns the length of a given string.
@SubStr	Returns substring from a given string.

1.2.4 Integer Constants and Constant Expressions

An integer constant is a series of one or more numerals followed by an optional radix specifier. For example, in these statements

```
mov     ax, 25
mov     ax, 0B3h
```

the numbers `25` and `0B3h` are integer constants. The `h` appended to `0B3` is a radix specifier. The specifiers are

- `y` for binary (or `b` if radix is less than or equal to 10)
- `o` or `q` for octal
- `t` for decimal (or `d` if radix is less than or equal to 10)
- `h` for hexadecimal

The default radix is decimal.

Radix specifiers can be either uppercase or lowercase letters; sample code in this book uses lowercase. If no radix is specified, the assembler interprets the integer according to the current radix. The default radix is decimal, but it can be changed with the **.RADIX** directive.

Hexadecimal numbers must always start with a decimal digit (0–9). If necessary, add a leading zero to distinguish between symbols and hexadecimal numbers that start with a letter. For example, `ABCh` is interpreted as an identifier. The hexadecimal digits A through F can be either uppercase or lowercase letters. Sample code in this book uses uppercase letters.

Values of integer constants and expressions are known at assembly time.

Constant expressions contain integer constants and (optionally) operators such as shift, logical, and arithmetic operators, and can be evaluated. The assembler evaluates them at assembly time. (In addition to constants, expressions can contain labels, types, registers, and their attributes.) Constant expressions do not change value during program execution.

Symbolic Integer Constants You can define symbolic integer constants with either of the data assignment directives, **EQU** or the equal sign (=). These directives assign values to symbols during assembly, not during program execution. Symbols defined as integer constants can then be used in subsequent statements as immediate operands having the assigned value. Symbolic constants are often used to assign mnemonic names to constant values, which makes your code more readable and easier to maintain.

The assembler does not allocate data storage when you use either **EQU** or =. Instead, it replaces each occurrence of the symbol with the value of the expression.

Symbols defined with EQU cannot be redefined.

The difference between **EQU** and = is that integers defined with the = directive can be changed in your source code, but those defined with **EQU** cannot. Once a symbolic integer constant has been defined with the **EQU** directive, attempting to redefine it generates an error. The syntax is

symbol **EQU** *expression*

The *symbol* must be a unique name. The *expression* can be an integer, a constant expression, a one- or two-character string constant (four-character on the 80386/486), or an expression that evaluates to an address. If a constant value used in numerous places in the source code needs to be changed, you modify the *expression* in one place rather than throughout the source code.

The following example shows the correct use of **EQU** to define symbolic integers.

```
column  EQU     80              ; Constant -    80
row     EQU     25              ; Constant -    25
screen  EQU     column * row    ; Constant  - 2000
line    EQU     row             ; Constant -    25

        .DATA

        .CODE
        .
        .
        .
        mov     cx, column
        mov     bx, line
```

The value of a symbol defined with the = directive can be different at different places in the source code. However, a constant value is assigned during assembly for each use, and that value does not change at run time.

The syntax for the = directive is

symbol = *expression*

Size of Constants The default word size for MASM 6.0 expressions is 32 bits. This behavior can be modified using **OPTION EXPR16** or **OPTION M510**. Both of these options set the expression word size to 16 bits, but **OPTION M510** affects other assembler behavior as well (see Appendix A).

It is illegal to change the expression word size once it has been set with **OPTION M510**, **OPTION EXPR16**, or **OPTION EXPR32**, but you can repeat the same directive in a file. This can be useful for putting an **OPTION EXPR16** in every include file, for example.

1.2.5 Operators

Operators are used in expressions. The value of the expression is determined at assembly time and does not change when the program runs.

Operators should not be confused with processor instructions. The reserved word **ADD** is an instruction. The plus sign (+) is an operator. For example, `Amount+2` is a valid use of the plus operator (+); it tells the assembler to add 2 to `Amount`, which might be a value or an address. This operation, which occurs at assembly time, is different from the **ADD** instruction, which tells the processor to perform addition at run time.

The assembler evaluates expressions that contain more than one operator according to the following rules:

- Operations in parentheses are always performed before any adjacent operations.
- Binary operations of highest precedence are performed first.
- Operations of equal precedence are performed from left to right.
- Unary operations of equal precedence are performed right to left.

The order of precedence for all operators is listed in Table 1.3. Operators on the same line have equal precedence.

Table 1.3 Operator Precedence

Precedence	Operators
1	( ), [ ]
2	**LENGTH, SIZE, WIDTH, MASK**
3	. (structure-field-name operator)
4	: (segment-override operator), **PTR**
5	**LROFFSET, OFFSET, SEG, THIS, TYPE**
6	**HIGH, HIGHWORD, LOW, LOWWORD**
7	+ ,– (unary)
8	*, /, **MOD, SHL, SHR**
9	+, – (binary)
10	**EQ, NE, LT, LE, GT, GE**
11	**NOT**
12	**AND**
13	**OR, XOR**
14	**OPATTR, SHORT, .TYPE**

1.2.6 Data Types

A "data type" describes a set of values. A variable of a given type can have any of a set of values within the range specified for that type.

The intrinsic types for MASM 6.0 are **BYTE, SBYTE, WORD, SWORD, DWORD, SDWORD, FWORD, QWORD,** and **TBYTE.** These types define integers and binary coded decimals (BCDs); they are discussed in Chapter 6. The signed data types **SBYTE, SWORD,** and **SDWORD** are new to MASM 6.0. They are useful in conjunction with directives such as **INVOKE** (for calling procedures) and **.IF** (introduced in Chapter 7). The **REAL4, REAL8,** and **REAL10** directives can be used to define floating-point types. See Chapter 6.

Previous versions of MASM have separate directives for types and initializers. For example, **BYTE** is a type and **DB** is the corresponding initializer. The distinction has been eliminated for MASM 6.0. Any type (intrinsic or user-defined) can be used as an initializer.

MASM does not have specific types for arrays and strings. However, it allows a sequence of data units to be treated as arrays, and character (byte) sequences to be treated as strings. (See Section 5.1, "Arrays and Strings.")

Types can also have attributes such as *langtype* and *distance* (**NEAR** and **FAR**). See Section 7.3.3, "Declaring Parameters with the PROC Directive," for information on these attributes.

You can also define your own types with **STRUCT**, **UNION**, and **RECORD**. The types have fields that contain string or numeric data, or records that contain bits. These data types are similar to the user-defined data types in high-level languages such as C, Pascal, and FORTRAN. (See Chapter 5, "Defining and Using Complex Data Types.")

The TYPEDEF directive defines aliases and pointer types.

You can define new types, including pointer types, with the **TYPEDEF** directive, which is also new to MASM 6.0. **TYPEDEF** assigns a *qualifiedtype* (explained below) to a *typename*.

NOTE The concept of the qualifiedtype is essential to understanding many of the new features in MASM 6.0, including prototypes and the **.IF** and **INVOKE** directives. Descriptions of these topics in later chapters refer to this section.

Once assigned, the *typename* can be used as a data type in your program. Use of the *qualifiedtype* also allows the CodeView debugger to display information on the type. You cannot use a *qualifiedtype* as an initializer, but you can use a type defined with **TYPEDEF**.

The *qualifiedtype* is any MASM type (such as structure types, union types, record types, or an intrinsic type) or can be a pointer to a type with the form

[[*distance*]] **PTR** [[*qualifiedtype*]]

where *distance* is **NEAR, FAR**, or any distance modifier. See Section 7.3.3, "Declaring Parameters with the PROC Directive," for more information on *distance*.

The *qualifiedtype* can also be any type previously defined with **TYPEDEF**. For example, if you use **TYPEDEF** to create an alias for **BYTE**, as shown below, then you can use that `CHAR` type as a *qualifiedtype* when defining the pointer type `PCHAR`.

```
CHAR    TYPEDEF BYTE
PCHAR   TYPEDEF PTR CHAR
```

Section 3.3, "Accessing Data with Pointers and Addresses," shows how to use the **TYPEDEF** directive to define pointers.

Since *distance* and *qualifiedtype* are optional syntax elements, you can use variables of type **PTR** or **FAR PTR**. You can also define procedure prototypes with *qualifiedtype*. See Section 7.3.6, "Declaring Procedure Prototypes," for more information about procedure prototypes.

Several rules govern the use of *qualifiedtype*:

- The only component of a *qualifiedtype* definition that can be forward-referenced is a structure or union type identifier.
- If *distance* is not specified, the right operand and current memory model determine the type of the pointer. If the operand following **PTR** is not a *distance*

or a function prototype, the operand is a pointer of the default data pointer type in the current mode. Otherwise, the type of the pointer is the *distance* of the right operand.

- If **.MODEL** is not specified, **SMALL** model (and therefore **NEAR** pointers) is the default.

A *qualifiedtype* can be used in seven places:

Use	Example
In procedure arguments	`proc1 PROC pMsg:PTR BYTE`
In prototype arguments	`proc2 PROTO pMsg:FAR PTR WORD`
With local variables declared inside procedures	`LOCAL pMsg:PTR`
With the **LABEL** directive	`TempMsg LABEL PTR WORD`
With the **EXTERN** and **EXTERNDEF** directives	`EXTERN pMsg:FAR PTR BYTE` `EXTERN MyProc:PROTO`
With the **COMM** directive	`COMM var1:WORD:3`
With the **TYPEDEF** directive	`PPBYTE TYPEDEF PTR PBYTE` `PFUNC TYPEDEF PROTO MyProc`

Section 3.3.1 shows ways to write a **TYPEDEF** type for a *qualifiedtype*. Attributes such as **NEAR** and **FAR** can also be applied to a *qualifiedtype*.

You can also determine an accurate definition for **TYPEDEF** and *qualifiedtype* from the BNF grammar definitions given in Appendix B. The BNF grammar defines each component of the syntax for any directive, showing the recursive properties of components such as *qualifiedtype*.

1.2.7 Registers

All the 8086 processors have the same base set of 16-bit registers. Some registers can be accessed as two separate 8-bit registers. In the 80386/486, most registers can also be accessed as extended 32-bit registers.

Figure 1.3 shows the registers common to all the 8086-based processors. Each register has its own special uses and limitations.

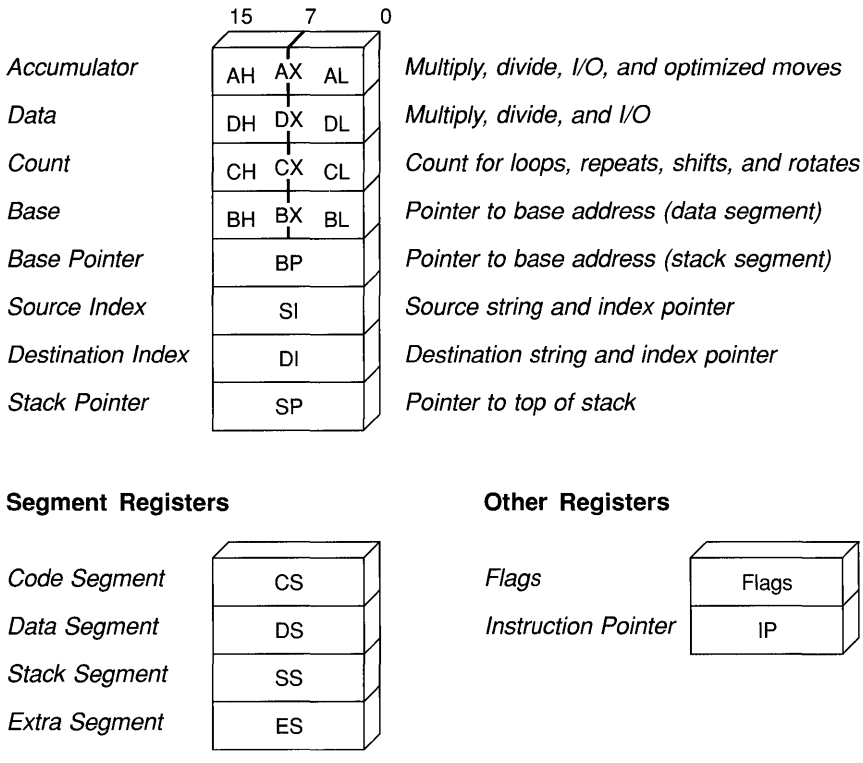


Figure 1.3 Registers for 8088-80286 Processors

80386/486 Only The 80386/486 processors use the same 8-bit and 16-bit registers that the rest of the 8086 family uses. All of these registers can be further extended to 32 bits, except segment registers, which always occupy 16 bits. The extended register names begin with the letter "E." For example, the 32-bit extension of AX is EAX. The 80386/486 processors have two additional segment registers, FS and GS. Figure 1.4 shows the extended registers of the 80386/486.

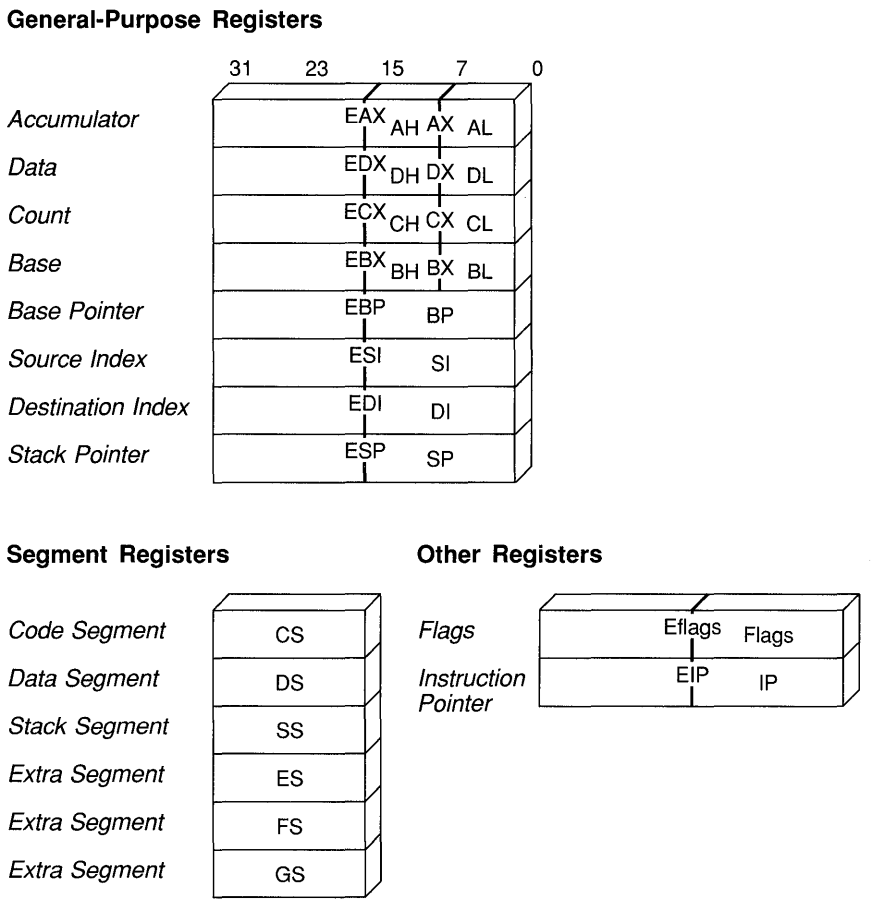


Figure 1.4 Extended Registers for the 80386/486 Processors

1.2.7.1 Segment Registers

At run time, all addresses are relative to one of four segment registers: CS, DS, SS, or ES. (The 80386/486 processors add two more, FS and GS.) These registers, their segments, and their purpose are listed below:

Register and Segment	Purpose
CS (Code Segment)	Contains processor instructions and their immediate operands.
DS (Data Segment)	Normally contains data allocated by the program.
SS (Stack Segment)	Creates stacks for use by **PUSH**, **POP**, **CALLS**, and **RET**.
ES (Extra Segment)	References secondary data segment. Used by string instructions.
FS, GS	Provides extra segments on the 80386/486.

1.2.7.2 General-Purpose Registers

Operations on registers are usually faster than operations on memory locations.

The AX, DX, CX, BX, BP, DI, and SI registers are 16-bit general-purpose registers. They can be used for temporary data storage. Since the processor accesses registers more quickly than it can access memory, you can speed up execution by keeping the most frequently used data in registers.

The 8086 family of processors does not perform memory-to-memory operations. Thus, operations on more than one variable often require the data to be moved into registers.

Four of the general registers, AX, DX, CX, and BX, can be accessed either as two 8-bit registers or as a single 16-bit register. The AH, DH, CH, and BH registers represent the high-order 8 bits of the corresponding registers. Similarly, AL, DL, CL, and BL represent the low-order 8 bits of the registers. All the general registers can be extended to 32 bits on the 80386/486.

1.2.7.3 Special-Purpose Registers

The 8086 family of processors has two additional registers whose values are changed automatically by the processor.

SP (Stack Pointer) The SP register points to the current location within the stack segment. Pushing a value onto the stack decreases the value of SP by 2; popping from the stack increases the value of SP by 2. With 32-bit operands on 80386/486 processors, SP is increased or decreased by 4 instead of 2. Call instructions store the calling address on the stack and decrease SP accordingly; return instructions get the stored address and increase SP. SP can also be manipulated as a general-purpose register with instructions such as **ADD**.

Only the processor can change IP.

IP (Instruction Pointer) The IP register always contains the address of the next instruction to be executed. You cannot directly access or change the instruction pointer. However, instructions that control program flow (such as calls, jumps, loops, and interrupts) automatically change the instruction pointer.

1.2.7.4 Flags Register

Flags reveal the status of the processor.

The 16 bits in the flags register control the execution of certain instructions and reflect the current status of the processor. In 80386/486 processors, the flags register is extended to 32 bits. Some bits are undefined, so there are actually 9 flags for real mode, 11 flags (including a 2-bit flag) for 80286 protected mode, 13 for the 80386, and 14 for the 80486. The extended flags register of the 80386/486 is sometimes called "Eflags."

Figure 1.5 shows the bits of the 32-bit flags register for the 80386/486. Only the lower word is used for the other 8086-family processors. The unmarked bits are reserved for processor use; do not modify them.

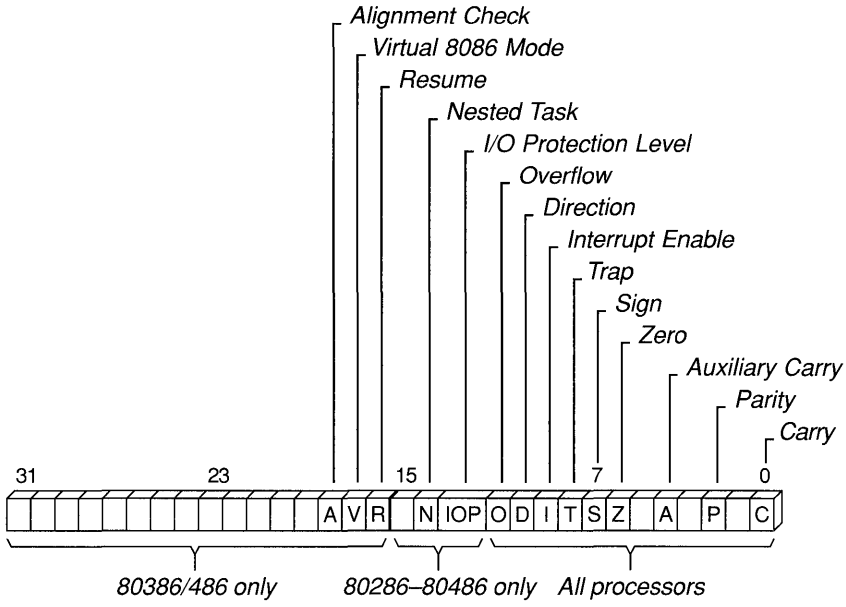


Figure 1.5 Flags for 8088-80486 Processors

The nine flags common to all 8086-family processors are summarized below, starting with the low-order flags. In these descriptions, "set" means the bit value is 1, and "cleared" means the bit value is 0.

Flag	Description
Carry	Set if an operation generates a carry to or a borrow from a destination operand.
Parity	Set if the low-order bits of the result of an operation contain an even number of set bits.
Auxiliary Carry	Set if an operation generates a carry to or a borrow from the low-order four bits of an operand. This flag is used for binary coded decimal (BCD) arithmetic.
Zero	Set if the result of an operation is 0.
Sign	Equal to the high-order bit of the result of an operation (0 is positive, 1 is negative).
Trap	If set, the processor generates a single-step interrupt after each instruction. A debugging program can use this feature to execute a program one instruction at a time.
Interrupt Enable	If set, interrupts are recognized and acted on as they are received. The bit can be cleared to turn off interrupt processing temporarily.
Direction	Set to make string operations process down from high addresses to low addresses; can be cleared to make string operations process up from low addresses to high addresses.
Overflow	Set if the result of an operation is too large or small to fit in the destination operand.

1.2.8 Statements

Statements are the line-by-line components of source files. Each MASM statement specifies an instruction or directive for the assembler. Statements have up to four fields. The syntax is shown below:

[[*name*]] [[*operation*]] [[*operands*]] [[;*comment*]]

The fields are explained below:

Field	Purpose
name	Defines a label that can be accessed from elsewhere in the program. For example, it can name a variable, type, segment, or code location.
operation	States the action of the statement. This field contains either an instruction or an assembler directive.

Field	Purpose
operands	Lists one or more items on which the instruction or directive operates.
comment	Provides a comment for the programmer. Comments are for documentation only; they are ignored by the assembler.

The following line contains all four fields:

```
mainlp: mov     ax, 7   ; Comments follow the semicolon
```

Here, `mainlp` is the label, `mov` is the operation, and `ax` and `7` are the operands, separated by a comma. The comment follows the semicolon.

All fields are optional, although certain directives and instructions require an entry in the name or operand field. Some instructions and directives place restrictions on the choice of operands. By default, MASM is not case sensitive.

Each field (except the comment field) must be separated from other fields by white-space characters (spaces or tabs). MASM also requires code labels to be followed by a colon, operands to be separated by commas, and comments to be preceded by a semicolon.

The backslash character joins physical lines into one logical line.

A logical line can contain up to 512 characters and occupy one or more physical lines. To extend a logical line into two or more physical lines, put the backslash character (\) as the last non-whitespace character before the comment or end of the line. You can place a comment after the backslash as shown in this example:

```
.IF     (x > 0)     \  ; X must be positive
 &&     (ax > x)    \  ; Result from function must be > x
 &&     (cx == 0)      ; Check loop counter too
 mov    dx, 20h
.ENDIF
```

Multiline comments can also be specified with the **COMMENT** directive. The assembler ignores all code between the delimiter character following the directive and the line containing the next instance of the delimiter character. This example illustrates the use of **COMMENT**.

```
COMMENT ^               The assembler
                        ignores this text
^       mov     ax, 1   and this code
```

1.3 The Assembly Process

Creating and running an executable file involves several processes:

- Assembling the source code into an object file
- Linking the object file with other modules or libraries into an executable program
- Loading that program into memory
- Running the program

Once you have written your assembly-language program, MASM provides several options for assembling it. The **OPTION** directive, new to MASM 6.0, has several different arguments that let you control the way MASM assembles your programs.

You can control assembly behavior with conditional assembly.

Conditional assembly allows you to create one source file that can generate a variety of programs, depending on the status of various conditional-assembly statements.

1.3.1 Generating and Running Executable Programs

This section briefly lists all the actions that take place during each of the assembly steps. You can change the behavior of some of these actions in various ways, for example, by using macros instead of procedures, or by using the **OPTION** directive or conditional assembly. The other chapters in this book discuss specific programming methods; this list simply gives you an overview.

1.3.1.1 Assembling

The ML.EXE program does two things to create an executable program. First, it assembles the source code into an intermediate object file. Second, it calls the linker, LINK.EXE, which links the object files and libraries into an executable program (usually with the .EXE extension).

At assembly time, the assembler

- Evaluates conditional-assembly directives, assembling if the conditions are true.
- Expands macros and macro functions.
- Evaluates constant expressions such as `MYFLAG AND 80H`, substituting the calculated value for the expression.

- Encodes instructions and nonaddress operands. For example, `mov cx, 13` can be encoded at assembly time because the instruction does not access memory.
- Saves memory offsets as offsets from their segment.
- Passes segments and segment attributes to the object file.
- Saves placeholders for offsets and segments (relocatable addresses).
- Outputs a listing if requested.
- Passes messages (such as **INCLUDELIB** and **.DOSSEG**) directly to the linker.

See Section 1.3.3 for information about conditional assembly; see Chapter 9 for macros. Chapters 2 and 3 give further details about segments and offsets, and Appendix C explains listing files.

1.3.1.2 Linking

Once your source code is assembled, the resulting object file is passed to the linker. At this point, the linker may combine several object files into an executable program.

At link time, the linker

- Combines segments according to the instructions in the object files, rearranging the positions of segments that share the same class or group.
- Fills in placeholders for offsets (relocatable addresses).
- Writes relocations for segments into the header of .EXE files (but not .COM files).
- Writes an executable image.

Section 2.3.4, "Defining Segment Groups," defines classes and groups. Chapter 3, "Using Addresses and Pointers," explains segments and offsets.

1.3.1.3 Loading

The operating system loads the file generated by the linker into memory. When the executable file is loaded into memory, DOS

- Reads the program segment prefix (PSP) header into memory.
- Allocates memory for the program, based on the values in the PSP.
- Loads the program.
- Calculates the correct values for absolute addresses from the relocation table.

- Loads the segment registers SS, CS, DS, and ES with values that point to the proper areas of memory.
- Loads the instruction pointer (IP) to point to the start address in the code segment and the stack pointer (SP) to point to the stack.
- Begins execution of the program.

The process is similar for OS/2.

See Section 1.2.7, "Registers," for information about segment registers, the instruction pointer (IP), and the stack pointer (SP). See MASM online help or a DOS reference for more information on the PSP.

1.3.1.4 Running

Your program is now ready to run. Some program operations cannot be handled until the program runs, such as resolving indirect memory operands. See Section 7.1.1.2, "Indirect Operands."

1.3.2 Using the OPTION Directive

The **OPTION** directive lets you modify global aspects of the assembly process. With **OPTION**, you can change command-line options and default arguments. These changes affect only statements that follow the use of **OPTION**.

For example, you may have MASM code in which the first character of a variable, macro, structure, or field name is a dot (.). Since a leading dot causes MASM 6.0 to generate an error, you can use this statement in your program:

```
OPTION DOTNAME
```

This enables the use of the dot for the first character.

Changes made with **OPTION** override any corresponding command-line option. For example, suppose you compile a module with this command line (which enables M510 compatibility):

```
ML /Zm TEST.ASM
```

but this statement is in the module:

```
OPTION NOM510
```

From this point on in the module, the M510 compatibility options are disabled.

The lists below explain each of the arguments for the **OPTION** directive. You can put more than one **OPTION** statement on one line if you separate them by commas.

Options for M510 Compatibility

Argument	Description
CASEMAP: *maptype*	**CASEMAP:NONE** (or /Cx) causes internal symbol recognition to be case sensitive and causes the case of identifiers in the .OBJ file to be the same as specified in the **EXTERNDEF**, **PUBLIC**, or **COMM** statement. The default is **CASEMAP:NOTPUBLIC** (or /Cp). It specifies case insensitivity for internal symbol recognition and the same behavior as **CASEMAP:NONE** for case of identifiers in .OBJ files. **CASEMAP:ALL** (/Cu) specifies case insensitivity for identifiers and converts all identifier names to uppercase.
DOTNAME | **NODOTNAME**	Enables the use of the dot (.) as the leading character in variable, macro, structure, union, and member names. **NODOTNAME** is the default.
M510 | **NOM510**	Sets all features to be compatible with MASM version 5.1, disabling the **SCOPED** argument and enabling **OLDMACROS**, **DOTNAME**, and, **OLDSTRUCTS**. **OPTION M510** conditionally sets other arguments for the **OPTION** directive. The default is **NOM510**. See Appendix A for more information on using **OPTION M510**.
OLDMACROS | **NOOLDMACROS**	Enables the version 5.1 treatment of macros. MASM 6.0 treats macros differently. The default is **NOOLDMACROS**.

Argument	Description
OLDSTRUCTS | **NOOLDSTRUCTS**	Enables compatibility with MASM 5.1 for treatment of structure members. See Section 5.2 for information on structures.
SCOPED | **NOSCOPED**	Guarantees that all labels inside procedures are local to the procedure when **SCOPED** (the default) is enabled.

Options for Procedure Use

Argument	Description
LANGUAGE: *langtype*	Specifies the default language type (**C**, **PASCAL**, **FORTRAN**, **BASIC**, **SYSCALL**, or **STDCALL**) to be used with **PROC**, **EXTERN**, and **PUBLIC**. This use of the **OPTION** directive overrides the **.MODEL** directive but is normally used when **.MODEL** is not given.
EPILOGUE: *macroname*	Instructs the assembler to call the *macroname* to generate a user-defined epilogue instead of the standard epilogue code when a **RET** instruction is encountered. See Section 7.3.8.
PROLOGUE: *macroname*	Instructs the assembler to call *macroname* to generate a user-defined prologue instead of generating the standard prologue code. See Section 7.3.8.
PROC: *visibility*	Allows the default visibility to be set explicitly. The default *visibility* is **PUBLIC**. The *visibility* can also be either **EXPORT** or **PRIVATE.**

Other Options

Argument	Description
EXPR16 | **EXPR32**	Sets the expression word size to 16 or 32 bits. The default is 32 bits. The **M510** argument to the **OPTION** directive sets the word size to 16 bits. Once set with the **OPTION** directive, the expression word size cannot be changed.
EMULATOR | **NOEMULATOR**	Controls the generation of floating-point instructions. The **NOEMULATOR** option generates the coprocessor instructions directly. The **EMULATOR** option generates instructions with special fixup records for the linker so that the Microsoft floating-point emulator, supplied with other Microsoft languages, can be used. It produces the same result as setting the /Fpi command-line option. You can set this option only once per module.
LJMP | **NOLJMP**	Enables automatic conditional-jump lengthening. The default is LJMP. See Section 7.1.2 for information about conditional-jump lengthening.
NOKEYWORD:<*keywordlist*>	Disables the specified reserved words. See Section 1.2.1, "Reserved Words," for an example of the syntax for this argument.
NOSIGNEXTEND	Overrides the default sign-extended opcodes for the **AND**, **OR**, and **XOR** instructions and generates the larger non-sign-extended forms of these instructions. Provided for compatibility with NEC V25® and NEC V35™ controllers.

Argument	Description
OFFSET: *offsettype*	Determines the result of **OFFSET** operator fixups. **SEGMENT** sets the defaults for fixups to be segment-relative (compatible with MASM 5.1). **GROUP**, the default, generates fixups relative to the group (if the label is in a group). **FLAT** causes fixups to be relative to a flat frame. (The **.386** mode must be enabled to use **FLAT**.) See Appendix A for more information.
READONLY | **NOREADONLY**	Enables checking for instructions that modify code segments, thereby guaranteeing that read-only code segments are not modified. Replaces the /p command-line option of MASM 5.1. It is useful for OS/2, where code segments are normally read-only.
SEGMENT: *segSize*	Allows global default segment size to be set. Also determines the default address size for external symbols defined outside any segment. The *segSize* can be **USE16**, **USE32**, or **FLAT**.

1.3.3 Conditional Directives

MASM 6.0 provides conditional-assembly directives and conditional-error directives. You can also use conditional-assembly directives when you want to test for a specified condition and assemble a block of statements if the condition is true. You can use conditional-error directives when you want to test for a specified condition and generate an assembly error if the condition is true.

Both kinds of conditional directives test assembly-time conditions, not run-time conditions. Only expressions that evaluate to constants during assembly can be compared or tested. Predefined symbols are often used in conditional assembly. See Section 1.2.3.

Conditional-Assembly Directives

The **IF** and **ENDIF** directives enclose the statements to be considered for conditional assembly. The optional **ELSEIF** and **ELSE** blocks follow the **IF** directive. There are many forms of the **IF** and **ELSE** directives. Online help provides a complete list.

The syntax used for the **IF** directives is shown below. The syntax for other condition-assembly directives follow the same form.

IF *expression1*
ifstatements
[[**ELSEIF** *expression2*
elseifstatements]]
[[**ELSE**
elsestatements]]
ENDIF

The *statements* following the **IF** directive can be any valid statements, including other conditional blocks, which in turn can contain any number of **ELSEIF** blocks. **ENDIF** ends the block.

The statements following the **IF** directive are assembled only if the corresponding condition is true. If the condition is not true and an **ELSEIF** directive is used, the assembler checks to see if the corresponding condition is true. If so, it assembles the statements following the **ELSEIF** directive. If no **IF** or **ELSEIF** conditions are satisfied, the statements following the **ELSE** directive are assembled.

For example, you may want to assemble a line of code only if a particular variable has been defined. In this example,

```
        IFDEF   buffer
buff    BYTE    buffer DUP(?)
        ENDIF
```

`buff` is allocated only if `buffer` has been previously defined.

The following list summarizes the conditional-assembly directives:

Directive	Use
IF and **IFE**	Tests the value of an expression and allows assembly based on the result.
IFDEF and **IFNDEF**	Tests whether a symbol has been defined and allows assembly based on the result.
IFB and **IFNB**	Tests to see if a specified argument was passed to a macro and allows assembly based on the result.
IFIDN and **IFDIF**	Compares two macro arguments and allows assembly based on the result. (**IFDIFI** and **IFIDNI** perform the same action but are case insensitive.)

Conditional-Error Directives

You can use conditional-error directives to debug programs and check for assembly-time errors. By inserting a conditional-error directive at a key point in your code, you can test assembly-time conditions at that point. You can also use conditional-error directives to test for boundary conditions in macros.

Like other severe errors, those generated by conditional-error directives cause the assembler to return a nonzero exit code. If a severe error is encountered during assembly, MASM does not generate the object module.

For example, the **.ERRNDEF** directive produces an error if some label has not been defined. In this example, **.ERRNDEF** at the beginning of the conditional block makes sure that a `publevel` actually exists.

```
.ERRNDEF    publevel
IF          publevel LE 2
PUBLIC      var1, var2
ELSE
PUBLIC      var1, var2, var3
ENDIF
```

These directives use the syntax given in the previous section. The following list summarizes the conditional-error directives.

Directive	Use
.ERR	Forces an error where the directives occur in the source file. The error is generated unconditionally when the directive is encountered, but the directives can be placed within conditional-assembly blocks to limit the errors to certain situations.
.ERRE and **.ERRNZ**	Tests the value of an expression and conditionally generates an error based on the result.
.ERRDEF and **.ERRNDEF**	Tests whether a symbol is defined and conditionally generates an error based on the result.
.ERRB and **.ERRNB**	Tests whether a specified argument was passed to a macro and conditionally generates an error based on the result.
.ERRIDN and **.ERRDIF**	Compares two macro arguments and conditionally generates an error based on the result. (**.ERRIDNI** and **.ERRDIFI** perform the same action but are case sensitive.)

1.4 Related Topics in Online Help

In addition to information covered in this chapter, information on the following topics can be found in online help.

Topic	Access
Predefined symbols	From the "MASM 6.0 Contents" screen, choose "Predefined Symbols"
Operator precedence	From the list of tables on the "MASM 6.0 Contents" screen, choose "Operator Precedence"
Data types	Choose "Directives" from the "MASM 6.0 Contents" screen; then choose "Data Allocation" or "Complex Data Types" from the resulting screen
Registers	From the "MASM 6.0 Contents" screen, choose "Language Overview"; then choose "Processor Register Summary"
Processor directives	To see a table of directives, choose "Processor Selection" from the "MASM 6.0 Contents" screen
Conditional assembly and conditional errors	Choose "Directives" from the "MASM 6.0 Contents" screen
EVEN, ALIGN, OPTION	From the "MASM 6.0 Contents" screen, choose "Directives," then "Miscellaneous"
Radix specifiers	From the "MASM 6.0 Contents" screen, choose "Language Overview"
ML command-line options	From the "Microsoft Advisor Contents" screen, choose "Macro Assembler" from the " Command Line" list

Chapter 2

Organizing MASM Segments

A segment is a collection of instructions or data whose addresses are all relative to the same segment register. The code in your assembly-language program defines and organizes them.

Segments can be defined by using simplified segment directives or full segment definitions. Section 2.2, "Using Simplified Segment Directives," covers the directives you can use to begin, end, and organize segment program modules. It also discusses how to access far data and code with simplified segment directives.

Section 2.3, "Using Full Segment Definitions," describes how to order, combine, and divide segments, as well as how to use the **SEGMENT** directive to define full segments. It also tells you how to create a segment group so that you can use just 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. See Chapter 8, "Sharing Data and Procedures among Modules and Libraries," for more information about multiple-module programming.

2.1 Overview of Memory Segments

A physical segment is an area of memory in which all locations are contiguous and share the same segment address. A segment always begins on a 16-byte (paragraph) boundary (unless an alignment attribute is specified with **ALIGN**). While 16-bit segments can occupy up to 64K (kilobytes), 32-bit segments can be as large as 4 gigabytes.

Segments reflect the architecture of the original 8086 processor. Prior to the 80386 processors and OS/2 2.*x*, assembly-language programming meant using segmented memory. A flat address space is now available on 80386/486 processors in 32-bit mode. This space is still segmented at the hardware level, but it allows you to ignore most segmentation concerns.

Segments provide a means for associating similar kinds of data. Most programs have segments for code, data, constant data, and the stack. These logical segments are allocated by the assembler at assembly time.

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 are easier to use than full segment definitions.

Simplified segment directives hide many of the details of segment definition and assume the same conventions used by Microsoft high-level languages. (See Section 2.2.) 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 Section 2.3.) If you use full segment definitions, you must write code to handle all the tasks performed automatically by the simplified segment directives.

2.2 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**. These directives and the arguments they take are discussed in the following sections.

The main module is where execution begins.

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), the default operating system (**OS_DOS**), 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
```

A module must always finish with the END directive.

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, whether or not it is the main module.

2.2.1 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 list below summarizes the *memorymodel* field and the *modeloptions* fields (language, operating system, 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.
Operating system	**OS_OS2** or **OS_DOS**. Determines behavior of **.STARTUP** and **.EXIT**.
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, os_os2 ; Medium memory model,
                                ;  Pascal conventions,
                                ;  OS/2 start-up/exit
```

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. Your choice of a memory model does not limit the kind of instructions you can write. It does, however, control segment defaults and determine whether data and code are near or far by default (see Table 2.1).

Table 2.1 Attributes of Memory Models

Memory Model	Default Code	Default Data	Operating System	Data and Code Combined
Tiny	Near	Near	DOS	Yes
Small	Near	Near	DOS, OS/2 1.*x*	No
Medium	Far	Near	DOS, OS/2 1.*x*	No
Compact	Near	Far	DOS, OS/2 1.*x*	No
Large	Far	Far	DOS, OS/2 1.*x*	No
Huge	Far	Far	DOS, OS/2 1.*x*	No
Flat	Near	Near	OS/2 2.*x*	Yes

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

The predefined symbol **@Model** returns the memory model. It encodes memory models as integers 1 through 7. See Section 1.2.3 for more information on predefined symbols, and see online help for an example of how to use them.

The seven memory models supported by MASM 6.0 divide into three groups.

Small, Medium, Compact, Large, and Huge Models The traditional memory models recognized by many DOS and OS/2 1.*x* 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 OS/2 does not support tiny model, but DOS does under MASM 6.0. This model places all data and code in a single segment. Therefore, the total program size can be 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 DOS memory allocation services.

Tiny model produces DOS .COM files. Specifying `.MODEL tiny` automatically sends a /TINY to the linker. Therefore, /AT 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 for 32-bit operating systems. It is similar to tiny model in that all code and data go in a single 32-bit segment.

OS/2 2.*x* uses flat model when you specify the **.386** or **.486** directive before **.MODEL FLAT**. All data and code (including system resources) are in a single 32-bit segment. Segment registers are initialized automatically at load time; the programmer needs to modify them only when mixing 16-bit and 32-bit segments in a single application. CS, DS, ES, and SS are all assumed to the supergroup **FLAT**. FS and GS are assumed to **ERROR**, since 32-bit versions of OS/2 reserve the use of these registers. Addresses and pointers passed to system services are always 32-bit near addresses and pointers. Although the theoretical size of the single flat segment is four gigabytes, OS/2 2.0 actually limits it to 512 megabytes in flat model.

Choosing the Language Convention

The language type is most important when you write a mixed-language program.

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. OS/2 system calls require the **PASCAL** calling convention for OS/2 1.*x*, but require the **SYSCALL** convention for OS/2 2.*x*. Specifying **STDCALL** for the calling convention enables a different calling convention and the same naming convention (see Section 20.1).

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. Chapter 7, "Controlling Program Flow," tells how to use these directives. You can also use the **OPTION** directive to set the language type. (See Section 1.3.2.) 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. See online help for a description of the bit flags.

See Chapter 20, "Mixed-Language Programming," for more information on calling and naming conventions. See Chapter 7, "Controlling Program Flow," for information about writing procedures and prototypes. See Chapter 8, "Sharing Data and Procedures among Modules and Libraries," for information on multiple-module programming.

Specifying the Operating System

The operating-system options (**OS_DOS** or **OS_OS2**) are arguments of **.MODEL**. They specify the start-up and exit code generated by the **.STARTUP** and **.EXIT** directives. (See Section 2.2.6.) If you do not use **.STARTUP** and **.EXIT**, you can omit this option. The default is **OS_DOS**.

Setting the Stack Distance

The **NEARSTACK** setting places the stack segment in a group, DGROUP, shared with data. 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 yourself or your program may fail to run. (See Section 2.2.6 for information about start-up code.) 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.

Having SS equal to DS gives some programming advantages.

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 (DLLs) when you cannot assume that the caller's stack is near.

The stack specification also affects the **ASSUME** statement generated by **.MODEL** and **.STACK**. You can use the predefined symbol **@Stack** to determine if the stack location is DGROUP (for near stacks) or STACK (for far stacks).

2.2.2 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 available 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 **.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. Privileged instructions are not necessary for writing applications, even for OS/2. Generally, you don't need privileged instructions unless you are writing operating-systems code or device drivers.

Processor directives affect availability of various MASM language features.

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.

2.2.3 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."

2.2.4 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 Section 2.3.4, "Defining Segment Groups"). These four directives also prevent instructions from appearing in data segments by assuming CS to **ERROR**. (See Section 2.3.3 for information about **ASSUME**.)

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 DOS or 512 megabytes under flat model in OS/2 2.0. It is placed in a special group identified as DGROUP, which is also limited to 64K.

Near data pointers always point to DGROUP.

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 for defining 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 wish. If you use C start-up 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 Section 1.2.3, "Predefined Symbols."

Far Data Segments

The compact, large, and huge memory models use far data addresses by default. With these memory models, however, you can still use **.DATA**, **.DATA?**, and **.CONST** to create data segments. 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
```

See Section 3.1.2 for more information on far data.

2.2.5 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 Section 2.2, "Using Simplified Segment Directives," 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.

The assembler assigns names to code segments.

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**. (See Appendix E, "Default Segment Names," for a complete list of segment names generated by MASM.)

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 example below 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 Section 3.1.2, "Near and Far Addresses."

NOTE The **ASSUME** directive is never necessary when you change code segments. In MASM 6.0, the assembler always assumes that the CS register contains the address of the current code segment or group. See Section 2.3.3 for more information about **ASSUME** used with segment registers.

2.2.6 Starting and Ending Code with .STARTUP and .EXIT

The easiest way to begin and end a 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.

.STARTUP generates the start-up code required by either DOS or OS/2.

These directives make programs easy to maintain. They automatically generate code appropriate to the operating system and stack types specified with **.MODEL**. Thus, you can specify the program is for a different operating system or stack type by altering keywords in the **.MODEL** directive.

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. When **.STARTUP** is present, the assembler ignores any argument to **END**.

The code generated by **.STARTUP** depends on the operating system specified after **.MODEL**.

If your program uses DOS for its operating system (the default), the initialization code sets DS to DGROUP, and adjusts SS:SP so that it is relative to the group for near data, DGROUP. To initialize a DOS program with the default **NEARSTACK** attribute, **.STARTUP** generates 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
```

A DOS program with the **FARSTACK** attribute does not need to adjust SS:SP, so it just initializes DS:

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

OS/2 initializes DS so that it points to DGROUP and sets SS:SP as desired. Thus, when the **OS_OS2** attribute is given, **.STARTUP** generates only a starting address. This does not show up in the listing file, however, since the /Sg option for listing files shows only the generated instructions.

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 1 means an error terminated the program. The **.EXIT** directive accepts the exit code as its one optional argument:

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

This directive generates a DOS interrupt or OS/2 system call, depending on the operating system specified in **.MODEL**. The code generated under DOS depends on the argument provided to **.EXIT**. One example is

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

if a return value is specified. The return value can be a constant, a memory reference, or a register that can be moved into the AL register. If no return value is specified, the first line in the example code above is not generated.

For OS/2, **.EXIT** invokes **DosExit** if you provide a prototype for **DosExit** and if you include OS2.LIB. The listing file shows the statements generated by **INVOKE** if the /Sg command-line option is specified. If you specify a return value as an expression, the code generated passes the expression instead of the register contents to the **DosExit** function. See Chapter 17 for information on writing programs for OS/2.

2.3 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 DOS without **.MODEL** and **.STARTUP**, you must initialize registers yourself and use the **END** directive to indicate the starting address. Under OS/2 you do not have to initialize registers. Section 2.3.2, "Controlling the Segment Order," describes typical start-up code.

2.3.1 Defining Segments with the SEGMENT Directive

The **SEGMENT** directive begins a segment, and the **ENDS** directive ends a segment:

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.

Options used with the SEGMENT directive can be in any order.

The optional types that follow the **SEGMENT** directive give the linker and the assembler instructions on how to set up and combine segments. The list below summarizes these types; the following sections explain them in more detail.

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.

Once you define a segment, you can reopen it later with another **SEGMENT** directive. When you reopen a segment, you need only give the segment name.

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 one of these:

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 are described in full detail in online help and are summarized below.

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 below.

Combine Type	Linker Action
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 below.
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 it is useful for defining structures or variables that correspond to specific far memory locations, such as a screen buffer or low memory. The **AT** combine type cannot be used 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 DOS tiny model (.COM) program, nor would you need a separate stack in a DLL library that used 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 80386/486 can operate in 16-bit or 32-bit mode.

The size attribute can be **USE16**, **USE32**, or **FLAT**. If the 80386 or 80486 processor has been selected with the **.386** or **.486** directive, and this directive precedes **.MODEL**, then **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.

NOTE Mixing 16-bit and 32-bit segments in the same program is possible but usually is necessary only in systems programming.

Setting Segment Order with Class Type

Segments of the same class are grouped together in the executable file.

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 orders segments in the order encountered. The **.ALPHA**, **.SEQ**, or **.DOSSEG** directive determines this order in each .OBJ file. The most common application for specifying a class type is to place all code segments first in the executable file.

2.3.2 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 they are declared.

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 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 in this order:

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. (See "Setting Segment Order with Class Type" in the previous section for more about segment classes.)

Linker Control

Most of the segment-ordering techniques (class names, **.ALPHA**, **.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.

.DOSSEG handles segment ordering.

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 Section 2.3.1, "Defining Segments with the SEGMENT Directive.")

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.

2.3.3 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.

The assembler must know the location of segment addresses.

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**]]

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*. Beginning with MASM 6.0, the assembler automatically sets CS to have 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 assumptions. Initializing segment registers at run time is discussed in Section 3.1.1.1, "Informing the Assembler about Segment Values."

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 Section 3.2.2, "Immediate Operands")
- 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.

The ASSUME directive can be used anywhere in your program.

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 Section 3.2.3, "Direct Memory Operands"). The segment-override operator is more convenient for one-time overrides, whereas the **ASSUME** directive may be more convenient if previous assumptions must be overridden for a sequence of instructions.

You can also prevent the use of a register with

```
ASSUME  SegRegister : ERROR
```

The assembler does 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.

See Section 3.3.2 for information on other applications of **ASSUME**.

2.3.4 Defining Segment Groups

A group is a collection of segments totalling not more than 64K in 16-bit mode. Each code or data item in the group can be addressed relative to the beginning of the group through DS or SS.

Segments within a group can be treated as if they shared the same segment address.

A group lets you develop separate 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 four 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.

2.4 Related Topics in Online Help

In addition to information covered in this chapter, information on the following topics can be found in online help.

Topic	Access
Memory models	Choose "Memory Models" from the list of tables on the "MASM 6.0 Contents" screen
@Model, @CodeSize, @DataSize	Choose "Predefined Symbols" from the "MASM 6.0 Contents" screen
Calling conventions	From the MASM Index, choose "Calling Convention"
Coprocessor Directives	From the "MASM 6.0 Contents" screen, choose "Directives"; then choose "Processor Selection"
Simplified and full (complete) segment control	From the "MASM 6.0 Contents" screen, choose "Directives"; then choose "Simplified Segment Control" or "Complete Segment Control"

Chapter 3

Using Addresses and Pointers

Most processor and operating-system modes require the use of segmented addresses to access the code and data for MASM applications. The address of the code or data in a segment is relative to an address in a segment register. You can also use pointers to access data in MASM programs.

The first section of this chapter describes how to initialize default segment registers to access near and far addresses. The next section describes how to use the available addressing modes to access the code and data. It also describes the related operators, syntax, and displacements.

The third section of this chapter explains how to use the **TYPEDEF** directive to declare pointers (variables containing addresses) and the **ASSUME** directive to give the assembler information about registers containing pointers. This section also shows you how to do typical pointer operations and how to write code that works for pointer variables in any memory model.

3.1 Programming Segmented Addresses

Before you use segmented addresses in your programs, you need to initialize the segment registers. The initialization process depends on the registers used and on your choice of simplified segment directives or full segment definitions. The simplified segment directives (introduced in Section 2.2) handle most of the initialization process for you. This section explains how to inform the assembler and the processor of segment addresses, and how to access the near and far code and data in those segments.

3.1.1 Initializing Default Segment Registers

The segmented architecture of the 8086-family of processors does not require you to specify two addresses every time you access memory. As Chapter 2, "Organizing MASM Segments," explains, the 8086 family of processors uses a system of default segment registers to simplify access to the most commonly used data and code.

The segment registers DS, SS, and CS are normally initialized to default segments at the beginning of a program. If you write the main module in a high-level language, the compiler initializes the segment registers. If you write the

main module in assembly language, you must initialize them yourself. Follow these two steps to initialize segments:

1. Tell the assembler which segment is associated with a register. The assembler must know the default segments at assembly time.
2. Tell the processor which segment is associated with a register by writing the necessary code to load the correct segment value into the segment register on the processor.

These steps are discussed separately in the following sections.

3.1.1.1 Informing the Assembler about Segment Values

Use ASSUME to inform the assembler about default segments.

The first step in initializing segments is to tell the assembler which segment to associate with a register. You do this with the **ASSUME** directive. If you use simplified segment directives, the assembler generates the appropriate **ASSUME** statements automatically. If you use full segment definitions, you must code the **ASSUME** statements for registers other than CS yourself. (**ASSUME** can also be used on general-purpose registers, as explained in Section 3.3.2, "Defining Register Types with ASSUME.")

With simplified segment directives, the **.STARTUP** directive and the start-up code initialize DS to be equal to SS (unless you specify **FARSTACK**), which allows default data to be accessed through either SS or DS. This can improve efficiency in the code generated by compilers. The "DS equals SS" convention may not work with certain applications, such as memory-resident programs in DOS and multithread programs in OS/2. The code generated for **.STARTUP** is shown in Section 2.2.6, "Starting and Ending Code with .STARTUP and .EXIT." You can use similar code to set DS equal to SS in programs using full segment definitions.

Here is an example using full segment definitions; it is equivalent to the **ASSUME** statement generated with simplified segment directives in small model with **NEARSTACK**:

```
ASSUME cs:_TEXT, ds:DGROUP, ss:DGROUP
```

In the example above, DS and SS are part of the same segment group. It is also possible to have different segments for data and code, and to use **ASSUME** to set ES, as shown below:

```
ASSUME cs:MYCODE, ds:MYDATA, ss:MYSTACK, es:OTHER
```

Correct use of the **ASSUME** statement can help find addressing errors. With **.CODE**, the assembler assumes CS to the current segment. When you use the simplified segment directives **.DATA**, **.DATA?**, **.CONST**, **.FARDATA**, or **.FARDATA?**, the assembler automatically assumes CS to **ERROR**. This prevents

instructions from appearing in these segments. If you use full segment definitions, you can accomplish the same by placing `ASSUME CS:ERROR` in a data segment.

With either simple or full segments, you can cancel the control of an **ASSUME** statement by assuming **NOTHING**. No assumptions is the default condition. For example, you cancel the assumption for ES above with the following statement:

```
ASSUME es:NOTHING
```

Prior to the **.MODEL** statement (or in its absence), the assembler sets the **ASSUME** statement for DS, ES, and SS to the current segment.

3.1.1.2 Informing the Processor about Segment Values

The second step in initializing segments is to inform the processor of segment values at run time. How segment values are initialized at run time differs for each segment register and depends on your use of simplified segment directives or full segment definitions and on the operating system.

Specifying a Starting Address The CS segment register and the IP (instruction pointer) register are initialized automatically if you use the **.STARTUP** directive with simplified segment directives. If you use full segment definitions, you must specifically set a label in the code segment at the instruction you want executed first. Then provide that label as an argument to the **END** directive. Both CS and IP are set at load time to the start address the linker gets from the **END** directive:

```
_TEXT   SEGMENT WORD PUBLIC 'CODE
        ORG     100h    ; Use this declaration for .COM files only
start:                  ; First instruction here
        .
        .
        .
_TEXT   ENDS
        END     start   ; Name of starting label
```

The operating system automatically resolves the value of CS:IP at load time. The label specified as the start address becomes the initial value of IP. In an executable (.EXE) file, the start address is encoded into the header and is initialized by the operating system at load time. In a .COM file, the initial IP is always assumed to be 100h. Therefore, you must use the **ORG** directive to set the start address to 100h. CS and IP cannot be directly modified except through jump, call, and interrupt instructions.

DS is initialized automatically under OS/2, but you must initialize it for DOS.

Initializing DS The DS register is automatically initialized to the correct value (DGROUP) if you use **.STARTUP** or if you are writing a program for OS/2. If you do not use **.STARTUP** with DOS, you must initialize DS using the following instructions:

```
mov     ax, DGROUP
mov     ds, ax
```

The initialization requires two instructions because the segment name is a constant and the assembler does not allow a constant to be loaded directly to a segment register. The example above loads DGROUP, but you can load any valid segment or group.

SS and SP are initialized automatically.

Initializing SS and SP The SS and SP registers are initialized automatically if you use the **.STACK** directive with simplified segments or if you define a segment that has the **STACK** combine type with full segment definitions. Using the **STACK** directive initializes SS to the stack segment. If you want SS to be equal to DS, use **.STARTUP** or its equivalent. (See "Combining Segments" in Section 2.3.1.) For an executable file, the values are encoded into the executable header and resolved at link time. For a .COM file, SS is initialized to the first address of the 64K program segment and SP is initialized to 0FFFEh.

If you do not need to access far data in your program, you do not need to initialize the ES register, although you can do so. Use the same technique as for the DS register. You can initialize SS to a far stack in the same way.

3.1.2 Near and Far Addresses

Addresses which have an implied segment name or segment registers associated with them are called "near addresses." Addresses which have an explicit segment associated with them are called "far addresses." The assembler handles near and far code automatically, as described below. You must specify how to handle far data.

The Microsoft segment model puts all near data and the stack in a group called DGROUP. Near code is put in a segment called _TEXT. Each module's far code or far data is placed in a separate segment. This convention is described in Section 2.3.2, "Controlling the Segment Order."

The assembler cannot determine the address for some program components, which are said to be relocatable. The assembler generates a fixup record and the linker provides the address once the location of all segments has been determined. Usually a relocatable operand references a label, but there are exceptions. Examples in the next two sections include information about the relocatability of near and far data.

Near Code Control transfers within near code do not require changes to segment registers. The processor automatically handles changes to the offset in the IP register when control-flow instructions such as **JMP**, **CALL**, and **RET** are used. The statement

```
call    nearproc        ; Change code offset
```

changes the IP register to the new address but leaves the segment unchanged. When the procedure returns, the processor resets IP to the offset of the next instruction after the call.

Far Code The processor automatically handles segment register changes when dealing with far code. The statement

```
call    farproc         ; Change code segment and offset
```

automatically moves the segment and offset of the `farproc` procedure to the CS and IP registers. When the procedure returns, the processor sets CS to the original code segment and sets IP to the offset of the next instruction after the call.

Near Data Near data can usually be accessed directly. That is, a segment register already holds the correct segment for the data item. The term "near data" is often used to refer to the data in the DGROUP group.

After the first initialization of the DS and SS registers, these registers normally point into DGROUP. If you modify the contents of either of these registers during the execution of the program, the register may need to be reloaded prior to being used for addressing DGROUP data.

If a stack variable is accessed directly through BP or SP, the SS register is the default. Otherwise, the default is DS:

```
nearvar WORD    0
        .
        .
        .
        mov     ax, nearvar ; Access near data through DS or SS
        mov     ax, [bp+6]  ; Access near data through SS
```

In this example, `nearvar` is a relocatable label. The assembler does not know where the memory for `nearvar` will be allocated. The linker provides the address at link time. The expression `[bp+6]` is not relocatable. The linker does not need to provide an address for this expression.

Far Data To read or modify a far address, a segment register must point to the segment of the data. This requires two steps. First load the segment (normally either ES or DS) with the correct value, and then (optionally) set an assume of the segment register to the segment of the address (or to **NOTHING**).

NOTE In flat model (OS/2 2.*x*), far addresses are rarely used. By default, all addressing is relative to the initial values of the segment registers. Thus, this section on far addressing does not apply to most flat model programs.

You can initialize ES.

One method commonly used to access far data is to initialize the ES segment register. This example shows two ways to do this:

```
; First method
        mov     ax, SEG farvar  ; Load segment of the far address
        mov     es, ax
        mov     ax, es:farvar   ; Provide an explicit segment
                                ;  override on the addressing
; Second method
        mov     ax, SEG farvar2 ; Load the segment of the
                                ;  far address
        mov     ex, ax
        ASSUME  ES:SEG farvar2  ; Tell the assembler that ES points
                                ;  to the segment containing farvar2
        mov     ax, farvar2     ; The assembler provides the ES
                                ;  override since it knows that
                                ;  the label is addressable
```

After loading the segment of the address into the ES segment register, you can either explicitly override the segment register so that the addressing is correct (method 1) or allow the assembler to insert the override for you (method 2). The assembler uses **ASSUME** statements to determine which segment register can be used to address a segment of memory. To use the segment override operator, the left operand must be a segment register, not a segment name. (See Section 3.2.3 for more information on segment overrides.)

If an instruction needs a segment override, the resulting code is slightly larger and slower, since the override must be encoded into the instruction. However, the resulting code may still be smaller than the code for multiple loads of the default segment register for the instruction.

The DS, SS, FS, and GS segment registers (FS and GS are available only on the 80386/486 processors) may also be used to provide for addressing through other segments.

If a program uses ES to access far data, it need not restore ES when finished (unless the program uses flat model). Some compilers require that you restore ES before returning to a module written in a high-level language.

You can reinitialize DS.

For a series of memory accesses to far data, you can reinitialize DS to the far data and then restore DS when you are finished. Use the **ASSUME** directive to let the assembler know that DS is no longer associated with the default data segment, as shown below:

```
push    ds                ; Save original segment
mov     ax, SEG fararray  ; Move segment into data register
mov     ds, ax            ; Initialize segment register
ASSUME  ds:SEG fararray   ; Tell assembler where data is
mov     ax, fararray[0]   ; Direct access faster
mov     dx, fararray[2]   ; (A relocatable expression)
.
.
.
pop     ds                ; Restore segment
ASSUME  ds:@DATA          ;  and default assumption
```

The additional overhead of saving and restoring the DS register in this data access method may be worthwhile to avoid repeated segment overrides.

If a program changes DS to access far data, it should restore DS when finished. This allows procedures to assume that DS is the segment for near data. This is a convention used in many compilers, including Microsoft compilers.

Relocatable Data The memory expression `es:farvar` is a relocatable memory expression, since the assembler cannot determine the address at assembly time.

Since no label is referenced, you may expect

```
mov ax, _myseg:0
```

to be nonrelocatable (in small model). However, in this case, `_myseg:0` is a location in a local module whose memory location is dependent on the link order, so `mov ax, _myseg:0` is relocatable.

A group name is also an immediate constant representing the beginning of the group. The first three expressions below are relocatable expressions; the fourth is not.

```
mov ax, DGROUP            ; Relocatable
mov ax, @data             ; Relocatable
mov ax, mygroup           ; Relocatable
mov ax, ds:0              ; Not relocatable
```

3.2 Specifying Addressing Modes

The 8086 family of processors recognizes four kinds of instruction operands: register, immediate, direct memory, and indirect memory. Each type of operand corresponds to a different addressing mode.

The four types of operands are summarized in the following list and described at length in the rest of this section.

Operand Type	Addressing Mode
Register	An 8-bit or 16-bit register on the 8086–80486; can also be 32-bit on the 80386/486
Immediate	A constant value contained in the instruction itself
Direct memory	A fixed location in memory
Indirect memory	A memory location determined at run time by using the address stored in one or two registers and a constant

3.2.1 Register Operands

A register operand specifies that the value in a particular register is an operand. Code for the register or registers used in operands is encoded into the instruction at assembly time.

Register operands can be used anywhere you need an operand. The following examples show typical register operands:

```
mov     bx, 10          ; Load constant to BX
add     ax, bx          ; Add AX and BX
jmp     di              ; Jump to the address in DI
```

Register operands have a specific use related to addresses.

An offset stored in a base or index register is often used as a pointer into memory. An offset can be stored in one of the base or index registers; the register can then be used as an indirect memory operand (see Section 3.2.4). For example:

```
mov     [bx], dl ; Store DL in indirect memory operand
inc     bx       ; Increment register operand
mov     [bx], dl ; Store DL in new indirect memory operand
```

This example moves the value in DL to two consecutive bytes of a memory location pointed to by BX. Any instruction that changes the register value also changes the data item pointed to by the register.

3.2.2 Immediate Operands

An immediate operand is a constant value that is specified at assembly time. It can be a constant or the result of a constant expression. Immediate values are usually encoded into the internal representation of the instruction at assembly time. These are typical examples:

```
mov     cx, 20          ; Load constant to register
add     var, 1Fh        ; Add hex constant to variable
sub     bx, 25 * 80     ; Subtract constant expression
```

The OFFSET Operator Address constants are a special case of immediate operand and consist of an offset or segment value. The **OFFSET** operator specifies the offset of a memory location, as shown below:

```
mov     bx, OFFSET var  ; Load offset address
```

For information on differences between MASM 5.1 behavior and MASM 6.0 behavior related to **OFFSET**, see Appendix A.

An OFFSET expression is resolved at link time.

Since segments in different modules may be combined into a single segment, the true base of the segment is not known. Thus, the offset cannot be resolved until link time and `var` is a relocatable immediate.

The SEG Operator The **SEG** operator specifies the segment of a memory location:

```
mov     ax, SEG farvar  ; Load segment address
mov     es, ax
```

A SEG expression is resolved at load time.

The actual value of a particular segment is never known until the program is loaded into memory. Constant segments are encoded into the header of the executable file at link time. Executable files in the DOS .COM format (tiny model) cannot contain relocatable segment expressions.

When you use the **SEG** operator with a variable that is not external, MASM 6.0 returns the address of the frame (the segment, group, or segment register) if one has been explicitly set. Otherwise, it returns the group if one has been specified. In the absence of a defined group, **SEG** returns the segment where the variable is defined.

For external variables that are not defined in a segment, the linker fills in the segment portion of the address, which may be a segment or group.

This behavior can be changed with the /Zm command-line option or with the **OPTION OFFSET:SEGMENT** statement (see Appendix A, "Differences between MASM 6.0 and 5.1"). Section 1.3.2 introduces the **OPTION** directive.

3.2.3 Direct Memory Operands

A direct memory operand specifies the data at a given address. The address and size of the data are encoded into the internal representation of the instruction. However, the instruction acts on the contents of the address, not the address itself. You must usually specify the size of these operands so that the instruction knows how much memory to operate on.

The offset value of a direct memory operand is not resolved until link time, and the segment must always be in a segment register at run time. The assembler automatically handles address resolution.

You usually represent a direct memory operand in source code as a symbolic name previously declared with a data directive such as **BYTE**, as illustrated below:

```
        .DATA?          ; Segment for uninitialized data
var     BYTE    ?       ; Reserve one byte at current address
                        ;  and assign this address to var
        .CODE
        .
        .
        .
        mov     var, al ; Load contents of byte register into
                          address specified by var
```

Any location in memory can be a direct memory operand as long as a size is specified and the location is fixed. The data at the address can change, but the address cannot. By default, instructions that use direct memory addressing use the DS register. You can create an expression that points to a memory location using any of the following operators:

Operator Name	Symbol
Plus	+
Minus	–
Index	[ ]
Structure member	.
Segment override	:

These operators are discussed in more detail below.

Several operators can be used in expressions that evaluate to direct memory operands.

Plus and Minus The result of combining a memory operand and a constant number with the plus or minus operator is a direct memory operand. However, the result of combining two memory operands with the minus operator is an immediate operand. For example:

```
memvar  EQU     array + 5       ; Address five bytes beyond array
immexp  EQU     mem1 - mem2     ; Distance between addresses
```

The second expression is legal only if both addresses are in the same segment.

The expression `mem1 - mem2` is not relocatable, since the reference to the two labels represents a difference in addresses (offsets). The linker does not need to know about the labels in this statement.

Index The index operator (brackets enclosing an index value) specifies the register or registers for indirect operands. It should contain a constant index when used with direct memory operands. It is equivalent to the plus operator. For example, the following statements are the same:

```
mov     ax, array[5]
mov     ax, array+5
```

Any direct memory operand can be enclosed in the index operator. The following are equivalent:

```
mov     ax, var
mov     ax, [var]
```

Some programmers prefer to enclose the operand in brackets to show that the contents, not the address, are used.

Structure Field The structure operator (a period) accesses elements of a structure. A field within a structure variable can be accessed as a direct memory operand:

```
mov     bx, structvar.field1
```

The address of the structure operand is the sum of the offsets of `structvar` and `field1`. See Section 5.2, "Structures and Unions," for more information about structures.

Segment Override The segment override operator (a colon) specifies a segment portion of the address that is different from the default segment. When used with instructions, this operator can apply to segment registers or segment names:

```
mov     ax, es:farvar           ; Use segment override
```

The assembler will not generate a segment override if the default segment is explicitly provided. Thus, the following two statements are equivalent:

```
mov     [bx], ax
mov     ds:[bx], ax
```

A segment name override or the segment override operator forces the operand to be an address expression.

```
        mov     WORD PTR FARSEG:0, ax   ; Segment name override
        mov     WORD PTR es:100h, ax    ; Legal and equivalent
        mov     WORD PTR es:[100h], ax  ;  expressions
;       mov     WORD PTR [100h], ax     ; Illegal, not an address
```

As the example shows, a constant expression cannot be an address expression unless it has a segment override.

3.2.4 Indirect Memory Operands

Like direct memory operands, indirect memory operands specify the contents of a given address. However, the processor calculates the address at run time by referring to the contents of registers. Since values in the registers can change at run time, indirect memory operands provide dynamic access to memory.

Indirect memory operands make possible run-time operations such as pointer indirection and dynamic indexing of array elements, including indexing of multidimensional arrays.

Strict rules govern which registers can be used for indirect memory operands under 16-bit versions of the 8086-based processors. The rules change significantly for 32-bit processors starting with the 80386. However, the new rules apply only to code that does not need to be backward compatible.

This section first discusses features of indirect operands in either mode. Then it explains the specific 16-bit rules and 32-bit rules separately.

3.2.4.1 Indirect Operands with 16- and 32-Bit Registers

Some rules and options for indirect memory operands always apply, regardless of the size of the register. For example, you must always specify the register and operand size for indirect memory operands. But you can use various syntaxes to indicate an indirect memory operand. This section describes the rules that apply to both 16-bit and 32-bit register modes.

Certain rules govern the use of base and index registers.

Specifying Indirect Memory Operands The index operator specifies the register or registers for indirect operands. The processor uses the data pointed to by the register. For example, the following instruction moves the word-sized data at the address contained in DS:BX into AX:

```
mov     ax, WORD PTR [bx]
```

When you specify more than one register, the processor adds the two addresses together to determine the effective address (the address of the data to operate on):

```
mov     ax, [bx+si]
```

An indirect memory operand can have a displacement.

Specifying Displacements You can specify an address displacement— a constant value to add to the effective address. A direct memory specifier is the most common displacement:

```
mov     ax, table[si]
```

In the relocatable expression above, the displacement `table` is the base address of an array; SI holds an index to an array element. The SI value is calculated at run time, often in a loop. The element loaded into AX depends on the value of SI at the time the instruction is executed.

Each displacement can be an address or numeric constant. If there is more than one displacement, the assembler adds them together at assembly time and encodes the total displacement. For example, in the statement

```
table   WORD    100 DUP (0)
        .
        .
        .
        mov     ax, table[bx][di]+6
```

both `table` and `6` are displacements. The assembler adds the value of `table` to `6` to get the total displacement. However, this statement is not legal:

```
mov ax, mem1[si] + mem2
```

Indirect memory operands must always have a size.

Specifying Operand Size Indirect memory operands must always have a specified size. Often the size is specified by the size of the identifier. In the example above, the size of the `table` array determines the operand size. If an indirect memory operand is used with a register operand, the register size determines the size of the memory object:

```
mov     ax, [bx]        ; Size is 2 bytes - same as AX
mov     table[bx], 0    ; Size is 2 bytes - from size
                        ;  of table
```

If there is no address or register operand, the size must be given specifically with the **PTR** operator, as shown below:

```
inc     WORD PTR [bx]           ; Word size
mov     BYTE PTR [bp+6], 0      ; Byte size
```

Syntax Options The assembler allows a variety of syntaxes for indirect memory operands. However, all registers must be inside brackets. You can enclose each register in its own pair of brackets, or you can place the registers in the same pair of brackets separated by a plus operator (+). All the following variations are legal and equivalent:

```
mov     ax, table[bx][di]
mov     ax, table[di][bx]
mov     ax, table[bx+di]
mov     ax, [table+bx+di]
mov     ax, [bx][di]+table
```

All of these statements move the value in `table` indexed by `BX+DI` into `AX`.

Registers pointing into arrays must be zero-based and scaled for the size of the array.

Scaling Indexes The value of index registers pointing into arrays must often be adjusted for zero-based arrays and scaled according to the size of the array items. For a word array, the item number must be multiplied by two (shifted left two places). When you are using 16-bit registers, scaling must be done with separate instructions, as shown below:

```
mov     bx, 5           ; Get sixth element (adjust for 0)
shl     bx, 1           ; Scale by two (word size)
inc     wtable[bx]      ; Increment sixth element in table
```

When using 32-bit registers on the 80386/486 processor, you can include scaling in the operand, as described in Section 3.2.4.3, "Indirect Memory Operands with 32-Bit Registers."

Accessing Structure Elements The structure member operator can be used in indirect memory operands to access structure elements. In this example, the structure member operator loads the `year` field of the fourth element of the `students` array into AL:

```
STUDENT STRUCT
  grade WORD    ?
  name  BYTE    20 DUP (?)
  year  BYTE    ?
STUDENT ENDS

students        STUDENT  < >
        .
        .                               ; Assume array initialized
        .                               ;  earlier
        mov     bx, OFFSET students     ; Point to array of students
        mov     ax, 4                   ; Get fourth element
        mov     di, SIZE STUDENT        ; Get size of STUDENT
        mul     di                      ; Multiply size times
                                        ;  elements to point to
                                        ;  current element
                                        ; Load field from element:
        mov     al, (STUDENT PTR[bx+di]).year
```

See Section 5.2 for more information on MASM structures.

3.2.4.2 Indirect Memory Operands with 16-Bit Registers

For 8086-based computers and DOS, you must follow the strict indexing rules established for the 8086 processor. Only four registers are allowed—BP, BX, SI, and DI—and those only in certain combinations.

BP and BX are base registers. SI and DI are index registers. You can use either a base or an index register by itself. But if you combine two registers, one must be a base and one an index. Here are legal and illegal forms:

```
        mov     ax, [bx+di]     ; Legal
        mov     ax, [bx+si]     ; Legal
        mov     ax, [bp+di]     ; Legal
        mov     ax, [bp+si]     ; Legal
;       mov     ax, [bx+bp]     ; Illegal - two base registers
;       mov     ax, [di+si]     ; Illegal - two index registers
```

Table 3.1 shows the modes in which registers can be used to specify indirect memory operands.

Table 3.1 Indirect Addressing Modes with 16-Bit Registers

Mode	Syntax	Effective Address
Register indirect	[BX] [BP] [DI] [SI]	Contents of register
Base or index	*displacement*[BX] *displacement*[BP] *displacement*[DI] *displacement*[SI]	Contents of register plus *displacement*
Base plus index	[BX][DI] [BP][DI] [BX][SI] [BP][SI]	Contents of base register plus contents of index register
Base plus index with displacement	*displacement*[BX][DI] *displacement*[BP][DI] *displacement*[BX][SI] *displacement*[BP][SI]	Sum of base register, index register, and *displacement*

Different combinations of registers and displacements have different timings, as shown in the Macro Assembler *Reference*.

3.2.4.3 Indirect Memory Operands with 32-Bit Registers

Instructions for the 80386/486 processor can be given in two segment modes—16-bit and 32-bit. Indirect memory operands are different in each mode. The segment mode is independent of the register size; you can use 32-bit registers in either mode.

In 16-bit mode, the 80386/486 operates in the mode used by all other 8086-based processors, with one difference: you can use 32-bit registers. If the 80386/486 processor is enabled (with the **.386** or **.486** directive), 32-bit general-purpose registers are available in either segment mode. Using them eliminates many of the limitations of 16-bit indirect memory operands. Using 80386/486 features can

make your DOS programs run faster and more efficiently if you are willing to sacrifice backward compatibility with other processors.

In 32-bit mode, an offset address can be up to four gigabytes. (Segments are still represented in 16 bits.) This effectively eliminates size restrictions on each segment, since few programs need four gigabytes of memory. OS/2 2.*x* uses 32-bit mode and flat model, which spans all segments. XENIX 386 uses 32-bit mode with multiple segments.

Any general-purpose 32-bit register can be used as either the base or the index.

80386/486 Enhancements On the 80386/486, the processor allows any general-purpose 32-bit register to be used as either the base or the index register (except ESP, which can be a base but not an index). The same register can also be used as both the base and index, but you cannot combine 16-bit and 32-bit registers. Several examples are shown below:

```
add     edx, [eax]              ; Add double
mov     dl, [esp+10]            ; Add byte from stack
dec     WORD PTR [edx][eax]     ; Decrement word
cmp     ax, array[ebx][ecx]     ; Compare word from array
jmp     FWORD PTR table[ecx]    ; Jump into pointer table
```

The index register can have a scaling factor of 1, 2, 4, or 8.

Scaling Factors With 80386/486 registers, the index register can have a scaling factor of 1, 2, 4, or 8. Any register except ESP can be the index register and can have a scaling factor. Specify the scaling factor by using the multiplication operator (*) adjacent to the register.

You can use scaling to index into arrays with different sizes of elements. For example, the scaling factor is 1 for byte arrays (no scaling needed), 2 for word arrays, 4 for doubleword arrays, and 8 for quadword arrays. There is no performance penalty for using a scaling factor. Scaling is illustrated in the following examples:

```
mov     eax, darray[edx*4]      ; Load double of double array
mov     eax, [esi*8][edi]       ; Load double of quad array
mov     ax, wtbl[ecx+2][edx*2] ; Load word of word array
```

Scaling is also necessary on earlier processors, but it must be done with separate instructions before the indirect memory operand is used, as described in Section 3.2.4.2, "Indirect Memory Operands with 16-Bit Registers."

The number of registers and the scaling factor affect base and index registers.

The default segment register is SS if the base register is EBP or ESP; it is DS for all other base registers. If two registers are used, only one can have a scaling factor. The register with the scaling factor is defined as the index register. The other register is defined as the base. If scaling is not used, the first register is the base. If only one register is used, it is considered the base for deciding the default segment unless it is scaled. The following examples illustrate how to determine the base register:

```
mov   eax, [edx][ebp*4] ; EDX base (not scaled - seg DS)
mov   eax, [edx*1][ebp] ; EBP base (not scaled - seg SS)
mov   eax, [edx][ebp]   ; EDX base (first - seg DS)
mov   eax, [ebp][edx]   ; EBP base (first - seg SS)
mov   eax, [ebp*2]      ; EBP base (only - seg SS)
```

Mixing 16-Bit and 32-Bit Registers Statements can mix 16-bit and 32-bit registers if the register use is correct. For example, the following statement is legal for either 16-bit or 32-bit segments:

```
mov     eax, [bx]
```

This statement moves the 32-bit value pointed to by BX into the EAX register. Although BX is a 16-bit pointer, it can still point into a 32-bit segment.

However, the following statement is never legal, since the CX register cannot be used as a 16-bit pointer (although ECX can be used as a 32-bit pointer):

```
;       mov     eax, [cx]       ; illegal
```

Operands that mix 16-bit and 32-bit registers are also illegal:

```
;       mov     eax, [ebx+si]   ; illegal
```

The following statement is legal in either mode:

```
mov     bx, [eax]
```

This statement moves the 16-bit value pointed to by EAX into the BX register. This works fine in 32-bit mode. However, in 16-bit mode, moving a 32-bit pointer into a 16-bit segment is illegal. If EAX contains a 16-bit value (the top half of the 32-bit register is 0), the statement works. However, if the top half of the EAX register is not 0, the operand points into a part of the segment that doesn't exist, and this generates an error. If you use 32-bit registers as indexes in 16-bit mode, you must make sure that the index registers contain valid 16-bit addresses.

3.3 Accessing Data with Pointers and Addresses

In high-level languages, a "pointer" (or pointer variable) is an address that is stored in a variable. Assembly language also uses pointer variables, but the term "pointer" has a wider use. The indirect memory operands discussed in the previous section can be thought of as pointers stored in registers.

An address can be stored in a pointer variable for later use. Program procedures (including OS/2 systems calls) frequently pass pointer variables onto the stack to transfer data between the calling program and the called procedure.

A pointer variable must be transferred to registers before it can be used.

Regardless of the reason for maintaining it, a pointer variable to data cannot in itself be directly used in MASM statements. (Pointers to code can be used directly.) It must first be loaded into registers as an indirect memory operand.

There is a difference between a far address and a far pointer. A "far address" is the address of a variable located in a far data segment. A "far pointer" is a variable that can specify both a segment and an offset. Like any other variable, a pointer variable can be located in either the default (near) data segment or in a far segment.

Previous versions of MASM allow pointer variables but provide little support for them. In previous versions, any address loaded into a variable can be considered a pointer, as in the following statements:

```
Var     BYTE    0               ; Variable
npVar   WORD    Var             ; Near pointer to variable
fpVar   DWORD   Var             ; Far pointer to variable
```

If a variable is initialized to the name of another variable, the initialized variable is a pointer, as shown in the example above. However, in previous versions of MASM, the CodeView debugger recognizes `npVar` and `fpVar` as word and doubleword variables. CodeView does not treat them as pointers, nor does it recognize the type of data they point to (bytes, in the example).

The new directive **TYPEDEF** and the new capabilities of **ASSUME** make it easier to manage pointers in registers and variables. These directives are discussed in the next two sections. Basic pointer and address operations are covered in Section 3.3.3.

3.3.1 Defining Pointer Types with TYPEDEF

Once defined, a TYPEDEF is considered the same as an intrinsic type.

You can define types for pointer variables using the **TYPEDEF** directive. A type so defined is considered the same as the intrinsic types provided by the assembler and can be used in the same contexts. The syntax for **TYPEDEF** when used to define pointers is

typename **TYPEDEF** [[*distance*]] **PTR** *qualifiedtype*

The *typename* is the name assigned to the new type. The *distance* can be **NEAR**, **FAR**, or any distance modifier. The *qualifiedtype* can be any previously intrinsic or defined MASM type, or a type previously defined with **TYPEDEF**. (See Section 1.2.6, "Data Types," for a full definition of *qualifiedtype*.)

Here are some examples of user-defined types:

```
PBYTE   TYPEDEF      PTR BYTE   ; Pointer to bytes
NPBYTE  TYPEDEF NEAR PTR BYTE   ; Near pointer to bytes
FPBYTE  TYPEDEF FAR  PTR BYTE   ; Far pointer to bytes
PWORD   TYPEDEF      PTR WORD   ; Pointer to words
NPWORD  TYPEDEF NEAR PTR WORD   ; Near pointer to words
FPWORD  TYPEDEF FAR  PTR WORD   ; Far pointer to words
```

```
PPBYTE  TYPEDEF      PTR PBYTE  ; Pointer to pointer to bytes
                                ;  (in C, an array of strings)
PVOID   TYPEDEF      PTR        ; Pointer to any type of data

STRUCT  PERSON                  ; Structure type
  name  BYTE    20 DUP (?)
  num   WORD    ?
PERSON  ENDS
PPERSON TYPEDEF      PTR PERSON ; Pointer to structure type
```

The distance of a pointer can either be set specifically or determined automatically by the memory model (set by **.MODEL**) and the segment size (16 or 32 bits). If you don't use **.MODEL**, near pointers are the default.

In 16-bit mode, a near pointer is two bytes that contain the offset of the object pointed to. A far pointer requires four bytes, and it contains both the offset and the segment. In 32-bit mode, a near pointer is four bytes and a far pointer is six bytes. If you specify the distance with **NEAR** or **FAR**, the default distance of the current segment size is used. You can use **NEAR16**, **NEAR32**, **FAR16**, and **FAR32** to override the defaults set by the current segment size. In flat model, **NEAR** is the default.

A pointer type created with **TYPEDEF** can be used to declare pointer variables. Here are some examples using the pointer types defined above:

```
; Type declarations
Array   WORD    25 DUP (0)
Msg     BYTE    "This is a string", 0
pMsg    PBYTE   Msg             ; Pointer to string
pArray  PWORD   Array           ; Pointer to word array
npMsg   NPBYTE  Msg             ; Near pointer to string
npArray NPWORD  Array           ; Near pointer to word array
fpArray FPWORD  Array           ; Far pointer to word array
fpMsg   FPBYTE  Msg             ; Far pointer to string

S1      BYTE    "first", 0      ; Some strings
S2      BYTE    "second", 0
S3      BYTE    "third", 0
pS123   PBYTE   S1, S2, S3, 0   ; Array of pointers to strings
ppS123  PPBYTE  pS123           ; A pointer to pointers to strings

Andy    PERSON  <>              ; Structure variable
pAndy   PPERSON Andy            ; Pointer to structure variable

                                ; Procedure prototype

EXTERN  ptrArray:PBYTE          ; External variable
Sort    PROTO   pArray:PBYTE    ; Parameter for prototype

; Parameter for procedure
Sort    PROC    pArray:PBYTE
```

```
        LOCAL   pTmp:PBYTE      ; Local variable
        .
        .
        .
        ret
Sort    ENDP
```

Once defined, pointer types can be used in any context where intrinsic types are allowed.

3.3.2 Defining Register Types with ASSUME

Beginning with MASM 6.0, you can use the **ASSUME** directive with general-purpose registers to specify that a register is a pointer to a certain size of object. For example:

```
        ASSUME  bx:PTR WORD     ; BX is word pointer until further
                                ;  notice
        inc     [bx]            ; Increment word pointed to by BX
        add     bx, 2           ; Point to next word
        mov     [bx], 0         ; Word pointed to by BX = 0
        .
        .                       ; Other pointer operations with BX
        .
        ASSUME  bx:NOTHING      ; Cancel assumptions
```

In this example, BX is specified to be a pointer to a word. After a sequence of using BX as a pointer, the assumption is cancelled by assuming **NOTHING**.

Without the assumption to **PTR WORD**, many instructions need a size specifier. The **INC** and **MOV** statements from the examples above would have to be written like this to specify the sizes of the memory operands:

```
        inc     WORD PTR [bx]
        mov     WORD PTR [bx], 0
```

When you have used **ASSUME**, attempts to use the register for other purposes generate assembly errors. In the example above, while the **PTR WORD** assumption is in effect, any use of BX inconsistent with its **ASSUME** declaration generates an error. For example,

```
;       mov     al, [bx]        ; Can't move word to byte register
```

You can also use the **PTR** operator to override defaults:

```
        mov     ax, BYTE PTR [bx] ; Legal
```

Similarly, you can use **ASSUME** to prevent the use of a register as a pointer or even to disable a register:

```
        ASSUME  bx:WORD, dx:ERROR
;       mov     al, [bx] ; Error - BX is an integer, not a pointer
;       mov     ax, dx   ; Error - DX disabled
```

See Section 2.3.3 for information on using **ASSUME** with segment registers.

3.3.3 Basic Pointer and Address Operations

You can do these basic operations with pointers and addresses:

- Initialize a pointer variable by storing an address in it
- Load an address into registers, directly or from a pointer

The sections in the rest of this chapter describe variations of these tasks with both pointers and addresses. The examples in these sections assume that you have previously defined the following pointer types with the **TYPEDEF** directive:

```
PBYTE   TYPEDEF      PTR BYTE   ; Pointer to bytes
NPBYTE  TYPEDEF NEAR PTR BYTE   ; Near pointer to bytes
FPBYTE  TYPEDEF FAR  PTR BYTE   ; Far pointer to bytes
```

3.3.3.1 Initializing Pointer Variables

Let the assembler initialize pointer variables when possible.

If the value of a pointer is known at assembly time, the assembler can initialize it automatically so that no processing time is wasted on the task at run time. The following example illustrates how to do this:

```
Msg     BYTE    "String", 0
pMsg    PBYTE   Msg
```

If a pointer variable can be conditionally defined to one of several constant addresses, initialization must be delayed until run time. The technique is different for near pointers than for far pointers, as shown below:

```
Msg1    BYTE    "String1"
Msg2    BYTE    "String2"
npMsg   NPBYTE  ?
fpMsg   FPBYTE  ?
        .
        .
        .
        mov     npMsg, OFFSET Msg1              ; Load near pointer

        mov     WORD PTR fpMsg[0], OFFSET Msg2  ; Load far offset
        mov     WORD PTR fpMsg[2], SEG Msg2     ; Load far segment
```

If you know that the segment for a far pointer is currently in a register, you can load it directly:

```
        mov     WORD PTR fpMsg[2], ds           ; Load segment of
                                                ;  far pointer
```

Dynamic Addresses Often the address to be initialized is dynamic. You know the register or registers containing the address, and you want to save them in a variable for later use. Typical situations include memory allocated by DOS (see interrupt 21h function 48h in online help) and addresses found by the **SCAS** or **CMPS** instructions (see Section 5.1.3.1). The technique for saving dynamic addresses is illustrated below:

```
; Dynamically allocated buffer
fpBuf   FPBYTE  0               ; Initialize so offset will be zero
        .
        .
        .
        mov     ah, 48h         ; Allocate memory
        mov     bx, 10h         ; Request 16 paragraphs
        int     21h             ; Call DOS
        jc      error           ; Return segment in AX
        mov     WORD PTR fpBuf[2], ax   ; Load segment
        .                               ;  (offset is already 0)
        .
        .
error:                          ; Handle error
```

There are several options for copying pointers.

Copying Pointers Sometimes one pointer variable must be initialized by copying from another. Here are two ways to copy a far pointer:

```
fpBuf1  FPBYTE  ?
fpBuf2  FPBYTE  ?
        .
        .
        .
; Copy through registers is faster, but requires a spare register
        mov     bx, WORD PTR fpBuf1[0]
        mov     WORD PTR fpBuf2[0], bx
        mov     bx, WORD PTR fpBuf1[2]
        mov     WORD PTR fpBuf2[2], bx

; Copy through stack is slower, but does not use a register
        push    WORD PTR fpBuf1[0]
        push    WORD PTR fpBuf1[2]
        pop     WORD PTR fpBuf2[2]
        pop     WORD PTR fpBuf2[0]
```

Pointers passed as procedure arguments are pushed onto the stack.

Pointers as Arguments When a pointer is passed as an argument to a procedure, it must be pushed onto the stack. The procedure then sets up a stack frame so that it can access the arguments from the stack. This technique is discussed in detail in Section 7.3.2, "Passing Arguments on the Stack." Pushing a pointer is illustrated below:

```
; Push a far pointer (segment always pushed first)
        push    WORD PTR fpMsg[2]       ; Push segment
        push    WORD PTR fpMsg[0]       ; Push offset
```

Pushing an address is somewhat different:

```
; Push a far address as a far pointer
        mov     ax, SEG fVar    ; Load and push segment
        push    ax
        mov     ax, OFFSET fVar ; Load and push offset
        push    ax
```

On the 80186 and later processors, you can shorten pushing a constant to one step:

```
        push    SEG fVar        ; Push segment
        push    OFFSET fVar     ; Push offset
```

3.3.3.2 Loading Addresses into Registers

Loading an address into a pair of registers is one of the most common tasks in assembly-language programming. You cannot do processing work with a constant address or a pointer variable until the address is loaded into registers.

Certain register pairs have standard uses.

You often load addresses into particular *segment*:*offset* pairs. The following pairs have specific uses:

Segment:Offset Pair	Standard Use
DS:SI	Source for string operations
ES:DI	Destination for string operations
DS:DX	Input for DOS functions
ES:BX	Output from DOS functions

In addition, you can use ES:SI, DS:DI, DS:BX, or any *segment*:*offset* pair for your own indirect memory operands. You can use SS:BP with a displacement to access procedure arguments or local variables in procedures.

Addresses from Data Segments For near addresses, you need only load the offset; the segment is assumed as SS for stack-based data and as DS for other data. You must load both segment and offset for far pointers.

Here is an example of loading an address to DS:BX from a near data segment:

```
        .DATA
Msg     BYTE    "String"
        .
        .
        .
        mov     bx, OFFSET Msg  ; Load address to BX
                                ;  (DS already loaded)
```

If the data is in a far data segment, it is loaded like this:

```
        .FARDATA
Msg     BYTE    "String"
        .
        .
        .
        mov     ax, SEG Msg     ; Load address to ES:BX
        mov     es, ax
        mov     bx, OFFSET Msg
```

Stack Variables The technique for loading the address of a stack variable is significantly different from the technique for loading near addresses. You may need to put the correct segment value into ES for string operations. The following example illustrates how to load the address of a local (stack) variable to ES:DI:

```
Task    PROC
        LOCAL   Arg[4]:BYTE

        push    ss      ; Since it's stack-based, segment is SS
        pop     es      ; Copy SS to ES
        lea     di, Arg ; Load offset to DI
```

Use LEA to load the offset of an indirect memory operand.

The local variable in this case actually evaluates to SS:[BP-4]. This is an offset from the stack frame (described in Section 7.3.2, "Passing Arguments on the Stack"). Since you cannot use the **OFFSET** operator to get the offset of an indirect memory operand, you must use the **LEA** (Load Effective Address) instruction.

Use MOV and OFFSET to load the offset of a direct memory operand.

Direct Memory Operands To get the address of a direct memory operand, you can use the **MOV** instruction with **OFFSET** or the **LEA** instruction. MASM 6.0 automatically optimizes the **LEA** statement by generating the smaller and faster code, as shown in this example:

```
        lea     si, Msg         ; If you code this statement,
        mov     si, OFFSET Msg  ;  MASM 6.0 generates this code
```

The **LEA** instruction can be used to determine the address of indirect memory operands, as shown below.

```
        lea     si, [bx]        ; Legal - LEA required for indirect
;       mov     si, OFFSET [bx] ; Illegal - no OFFSET on indirect
```

Far Pointers Use the **LES** and **LDS** instructions to load far pointers. Use the **MOV** instruction to load a near pointer. The following example shows how to load a far pointer to ES:DI and a near pointer to SI (assuming DS as the segment):

```
InBuf   BYTE    20 DUP (1)
OutBuf  BYTE    20 DUP (0)

npIn    NPBYTE  InBuf
fpOut   FPBYTE  OutBuf
        .
        .
        .
        les     di, fpOut       ; Load far pointer to ES:DI

        mov     si, npIn        ; Load near pointer to SI (assume DS)
```

Copying between Segment Pairs Copying from one register pair to another is complicated by the fact that you cannot copy one segment register directly to another. Two methods are shown below. Timings are for the 8088 processor:

```
; Copy DS:SI to ES:DI, generating smaller code
        push    ds              ; 1 byte, 14 clocks
        pop     es              ; 1 byte, 12 clocks
        mov     di, si          ; 2 bytes, 2 clocks

; Copy DS:SI to ES:DI, generating faster code
        mov     di, ds          ; 2 bytes, 2 clocks
        mov     es, di          ; 2 bytes, 2 clocks
        mov     di, si          ; 2 bytes, 2 clocks
```

3.3.3.3 Model-Independent Techniques

Use conditional assembly to write memory-model independent code.

Often you may want to write code that is memory-model independent. If you are writing libraries that must be available for different memory models, you can use conditional assembly to handle different sizes of pointers. You can use the predefined symbols **@DataSize** and **@Model** to test the current assumptions.

Use conditional assembly to handle pointers that have no specified distance.

You can use conditional assembly to write code that works with pointer variables that have no specified distance. The predefined symbol **@DataSize** tests the pointer size for the current memory model:

```
Msg1    BYTE    "String1"
pMsg    PBYTE   ?
        .
        .
        .
        IF      @DataSize
        mov     WORD PTR pMsg[0], OFFSET Msg1    ; Load far offset
        mov     WORD PTR pMsg[2], SEG Msg1       ; Load far segment
        ELSE
        mov     pMsg, OFFSET Msg1                ; Load near pointer
        ENDIF
```

In the following example, a procedure receives as an argument a pointer to a word variable. The code inside the procedure uses **@DataSize** to determine whether the current memory model supports far or near data. It loads and processes the data accordingly:

```
; Procedure that receives an argument by reference
mul8    PROC    arg:PTR WORD

        IF      @DataSize
        les     bx, arg     ; Load far pointer to ES:BX
        mov     ax, es:[bx] ; Load the data pointed to
        ELSE
        mov     bx, arg     ; Load near pointer to BX (assume DS)
        mov     ax, [bx]    ; Load the data pointed to
        ENDIF
        shl     ax, 1       ; Multiply by 8
        shl     ax, 1
        shl     ax, 1
        ret
mul8    ENDP
```

If you have many routines, writing the conditionals for each case can be tedious. The following conditional statements generate the proper instructions and segment overrides automatically.

```
; Equates for conditional handling of pointers
        IF @DataSize
lesIF   TEXTEQU    <les>
ldsIF   TEXTEQU    <lds>
esIF    TEXTEQU    <es:>
        ELSE
lesIF   TEXTEQU    <mov>
ldsIF   TEXTEQU    <mov>
esIF    TEXTEQU    <>
        ENDIF
```

Once you define these conditionals, you can use them to simplify code that must handle several types of pointers. This next example rewrites the above `mul8` procedure to use conditional code.

```
mul8    PROC    arg:PTR WORD

        lesIF   bx, arg         ; Load pointer to BX or ES:BX
        mov     ax, esIF [bx]   ; Load the data from [BX] or ES:[BX]
        shl     ax, 1           ; Multiply by 8
        shl     ax, 1
        shl     ax, 1
        ret
mul8    ENDP
```

The conditional statements from the examples above can be defined once in an include file and used whenever you need to handle pointers.

3.4 Related Topics in Online Help

In addition to information covered in this chapter, information on the following topics can be found in online help.

Topics	Access
LROFFSET, THIS	From the "MASM 6.0 Contents" screen, choose "Operators"; then choose "Address"
LFS, LGS, and **LSS**	From the "MASM 6.0 Contents" screen, choose "Processor Instructions"; then choose "Data Transfer"
ALIGN, EVEN, ORG	From the "MASM 6.0 Contents" screen, choose "Directives"; then choose "Miscellaneous"
NEAR, NEAR16, NEAR32, FAR16, FAR32, and **TYPE**	From the "MASM 6.0 Contents" screen, choose "Operators"; then choose "Type and Size"
PTR	From the "MASM 6.0 Contents" screen, choose "Operators"; then choose "Miscellaneous"
PUSHCONTEXT and **POPCONTEXT**	Access from the Macro Assembler Index
ASSUME, .MODEL	From the "MASM 6.0 Contents" screen, choose "Directives"; then choose "Simplified Segment Control"
@DataSize, @Model	From the "MASM 6.0 Contents" screen, choose "Predefined Symbols"

Chapter 4

Defining and Using Integers

The 8086 family of processors is designed to operate on integer data; therefore, most assembler statements are integer operations. Even string elements (discussed in Chapter 5, "Defining and Using Complex Data Types") are byte-sized integers to the assembler.

This chapter covers the concepts essential for using integer variables in assembly-language programs. The first section shows how to declare integer variables. The second section describes basic integer operations including moving, loading, and sign-extending integers, as well as calculating with integers. Finally, the last section describes how to do various operations with integers at the bit level, such as using bitwise logical instructions and shifting and rotating bits.

The complex data types introduced in the next chapter—arrays, strings, structures, unions, and records—use many of the integer operations illustrated in this chapter, since the components of complex data types are often integers. Floating-point operations require a different set of instructions and techniques. These are covered in Chapter 6, "Using Floating-Point and Binary Coded Decimal Numbers."

4.1 Declaring Integer Variables

You declare integer variables in the data segment of your program to allocate memory for data. The **EQU** and = directives define integer constants. Integer variables allocated with the data allocation directives can be initialized in several ways. MASM 6.0 provides new forms of the data allocation directives. This section discusses these features and explains how to use the **SIZEOF** and **TYPE** operators to provide information to the assembler about the types in your program. For information on symbolic integer constants, see Section 1.2.4, "Integer Constants and Constant Expressions."

4.1.1 Allocating Memory for Integer Variables

When you declare an integer variable by assigning a label to a data allocation directive, the assembler allocates memory space for the integer. The variable's name becomes a label for the memory space. The syntax is

[[*name*]] *directive initializer*

These directives, listed below, indicate the integer's size and value range.

Directive	Description of Initializers
BYTE, DB (bytes)	Allocates unsigned numbers from 0 to 255.
SBYTE (signed bytes)	Allocates signed numbers from –128 to +127.
WORD, DW (words = 2 bytes)	Allocates unsigned numbers from 0 to 65,535 (64K).
SWORD (signed words)	Allocates signed numbers from –32,768 to +32,767.
DWORD, DD (doublewords = 4 bytes)	Allocates unsigned numbers from 0 to 4,294,967,295 (4 megabytes).
SDWORD (signed doublewords)	Allocates signed numbers from –2,147,483,648 to +2,147,483,647.
FWORD, DF (farwords = 6 bytes)	Allocates 6-byte (48-bit) integers. These values are normally used only as pointer variables on the 80386/486 processors.
QWORD, DQ (quadwords = 8 bytes)	Allocates 8-byte integers used with 8087-family coprocessor instructions.
TBYTE, DT (10 bytes)	Allocates 10-byte (80-bit) integers if the initializer has a radix specifying the base of the number.

See Chapter 6 for information on the **REAL4**, **REAL8**, and **REAL10** directives that allocate real numbers.

The assembler enforces only the size of initializers.

MASM does not enforce the range of values assigned to an integer. If the value does not fit in the space allocated, however, the assembler generates an error.

The **SIZEOF** and **TYPE** operators, when applied to a type, return the size of an integer of that type. The following list gives the size attribute associated with each data type.

Data Type	Bytes
BYTE, SBYTE	1
WORD, SWORD	2
DWORD, SDWORD	3
FWORD	6
QWORD	8
TBYTE	10

The **SBYTE**, **SWORD**, and **SDWORD** data types are new to MASM 6.0. Use of these signed data types tells the assembler to treat the initializers as signed data. It is important to use these signed types with high-level constructs such as **.IF**, **.WHILE**, and **.REPEAT** (see Section 7.2.1, "Loop-Generating Directives"), and with **PROTO** and **INVOKE** directives (see Sections 7.3.6, "Declaring Procedure Prototypes," and 7.3.7, "Calling Procedures with INVOKE").

The assembler stores integers with the least significant bytes lowest in memory. Note that assembler listings and most debuggers show the bytes of a word in the opposite order—high byte first.

Figure 4.1 illustrates the integer formats.

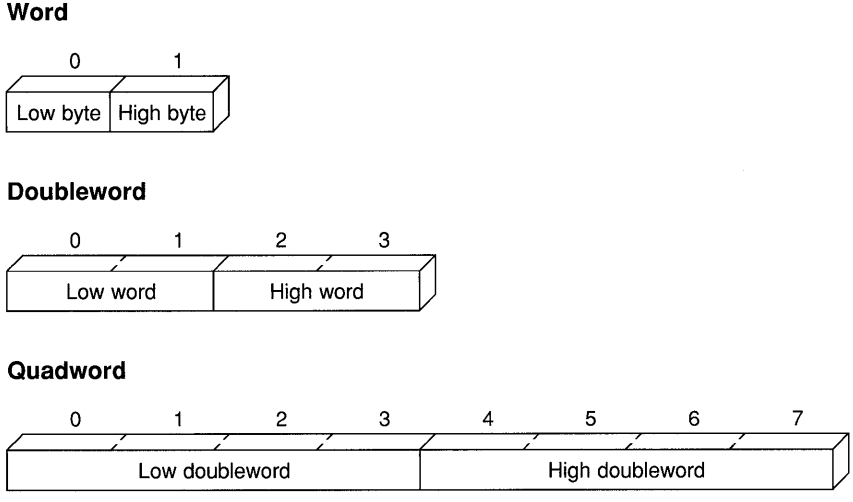


Figure 4.1 Integer Formats

TYPEDEF can define integer aliases.

Although the **TYPEDEF** directive's primary purpose is to define pointer variables (see Section 3.3.1), you can also use **TYPEDEF** to create an alias for any integer type. For example, these declarations

```
char    TYPEDEF SBYTE
longint TYPEDEF DWORD
float   TYPEDEF REAL4
double  TYPEDEF REAL8
```

allow you to use `char`, `longint`, `float`, or `double` in your programs if you prefer the C data labels.

4.1.2 Data Initialization

You can initialize variables when you declare them by giving initial values—that is, constants or expressions that evaluate to integer constants. The assembler generates an error if you specify an initial value too large for the specified variable type. Variables can also be initialized with **?** if there are no initial values.

You can declare and initialize variables in one step with the data directives, as these examples show.

```
integer         BYTE    16          ; Initialize byte to 16
negint          SBYTE   -16         ; Initialize signed byte to -16
expression      WORD    4*3         ; Initialize word to 12
signedexp       SWORD   4*3         ; Initialize signed word to 12
empty           QWORD   ?           ; Allocate uninitialized long
                                    ;  integer
                BYTE    1,2,3,4,5,6 ; Initialize six unnamed bytes
long            DWORD   4294967295  ; Initialize doubleword to
                                    ;  4,294,967,295
longnum         SDWORD  -2147433648 ; Initialize signed doubleword
                                    ;  to -2,147,433,648
tb              TBYTE   2345t       ; Initialize 10-byte binary
                                    ;  number
```

See Section 5.1, "Arrays and Strings," for information on arrays and on using the **DUP** operator to allocate initializer lists.

Once you have declared integer variables in your program, you can use them in integer operations such as adding, moving, loading, and exchanging. The next section describes these operations.

4.2 Integer Operations

You often need to copy, move, exchange, load, and sign-extend integer variables in your MASM code. This section shows how to do these operations as well as how to add, subtract, multiply, and divide integers; push and pop integers onto

the stack; and do bit-level manipulations with logical, shift, and rotate instructions.

The PTR operator tells the assembler the size of the operand.

Since MASM instructions require operands to be the same size, you may need to operate on data in a size other than the size originally declared. The **PTR** operator lets you do this. For example, you can use the **PTR** operator to access the high-order word of a **DWORD**-size variable. The syntax for the **PTR** operator is

type **PTR** *expression*

where the **PTR** operator forces *expression* to be treated as having the type specified. An example of this use is

```
        .DATA
num     DWORD   0
        .CODE

        mov     ax, WORD PTR num[0] ; Loads a word-size value from
        mov     dx, WORD PTR num[2] ;  a doubleword variable
```

You might choose not to use **PTR**, in contrast to this example. In that case, trying to move `num[0]` into AX generates an error.

4.2.1 Moving and Loading Integers

The primary instructions for moving integers from operand to operand and loading them into registers are **MOV** (Move), **XCHG** (Exchange), **XLAT** (Translate), **CWD** (Convert Word to Double), and **CBW** (Convert Byte to Word).

4.2.1.1 Moving Integers

The most common method of moving data, the **MOV** instruction, can be thought of as a copy instruction, since it always copies the source operand to the destination operand. Immediately after a **MOV** instruction, both the source and destination operands contain the same value.

The statements in the following example illustrate each type of memory move that can be performed with a single instruction. Note that you cannot move memory operands to memory operands in one operation.

```
; Immediate value moves
        mov     ax, 7       ; Immediate to register
        mov     mem, 7      ; Immediate to memory direct
        mov     mem[bx], 7  ; Immediate to memory indirect
; Register moves
        mov     mem, ax     ; Register to memory direct
        mov     mem[bx], ax ; Register to memory indirect
        mov     ax, bx      ; Register to register
        mov     ds, ax      ; General register to segment
                            ;  register
```

```
; Direct memory moves
        mov     ax, mem      ; Memory direct to register
        mov     ds, mem      ; Memory to segment register

; Indirect memory moves
        mov     ax, mem[bx] ; Memory indirect to register
        mov     ds, mem[bx] ; Memory indirect to segment register

; Segment register moves
        mov     mem, ds      ; Segment register to memory
        mov     mem[bx], ds ; Segment register to memory indirect
        mov     ax, ds       ; Segment register to general
                             ;  register
```

This next example shows several common types of moves that require two instructions.

```
; Move immediate to segment register
        mov     ax, DGROUP  ; Load immediate to general register
        mov     ds, ax       ; Store general register to segment
                             ;  register

; Move memory to memory
        mov     ax, mem1     ; Load memory to general register
        mov     mem2, ax     ; Store general register to memory

; Move segment register to segment register
        mov     ax, ds       ; Load segment register to general
                             ;  register
        mov     es, ax       ; Store general register to segment
                             ;  register
```

The **MOVSX** and **MOVZX** instructions for the 80386/486 processors extend and copy values in one step. See Section 4.2.1.4, "Extending Signed and Unsigned Integers."

4.2.1.2 Exchanging Integers

The **XCHG** (Exchange) instruction exchanges the data in the source and destination operands. Data can be exchanged between registers or between registers and memory, but not from memory to memory:

```
        xchg    ax, bx       ; Put AX in BX and BX in AX
        xchg    memory, ax   ; Put "memory" in AX and AX in "memory"
;       xchg    mem1, mem2   ; Illegal- can't exchange between
                             ;  memory location
```

In some circumstances, register-to-register moves are faster with **XCHG** than with **MOV**. If speed is important in your programs, check the *Reference* to find the fastest clock speeds for various operand combinations allowed with **MOV** and **XCHG**.

4.2.1.3 Translating Integers from Tables

The **XLAT** (Translate) instruction loads data from a table into memory. The instruction is useful for translating bytes from one coding system to another. The syntax is

XLAT[[**B**]] [[[*segment*:]]*memory*]]

XLAT and XLATB are synonyms.

The BX register must contain the address of the start of the table. By default, the DS register contains the segment of the table, but you can use a segment override to specify a different segment. Also, you need not give the operand except when specifying a segment override. (See Section 3.2.3, "Direct Memory Operands," for information about the segment override operator.)

Before the **XLAT** instruction executes, the AL register should contain a value that points into the table (the start of the table is position 0). After the instruction executes, AL contains the table value pointed to. For example, if AL contains 7, the assembler puts the eighth byte of the table in the AL register.

This example, illustrating **XLAT**, looks up hexadecimal characters in a table to convert an eight-bit binary number to a string representing a hexadecimal number.

```
; Table of hexadecimal digits
hex     BYTE    "0123456789ABCDEF"
convert BYTE    "You pressed the key with ASCII code "
key     BYTE    ?,?,"h",13,10,"$"
        .CODE
        .
        .
        .
        mov     ah, 8               ; Get a key in AL
        int     21h                 ; Call DOS
        mov     bx, OFFSET hex      ; Load table address
        mov     ah, al              ; Save a copy in high byte
        and     al, 00001111y       ; Mask out top character
        xlat                        ; Translate
        mov     key[1], al          ; Store the character
        mov     cl, 12              ; Load shift count
        shr     ax, cl              ; Shift high character into
                                    ;  position
        xlat                        ; Translate
        mov     key, al             ; Store the character
        mov     dx, OFFSET convert  ; Load message
        mov     ah, 9               ; Display character
        int     21h                 ; Call DOS
```

4.2.1.4 Extending Signed and Unsigned Integers

Since moving data to a different-sized register is illegal, you must "sign-extend" integers to convert signed data to a larger register or register pair.

Sign-extending means copying the sign bit of the unextended operand to all bits of the extended operand. The instructions in the following list sign-extend values as shown. They work only on signed values in the accumulator register.

Instruction	Function
CBW	Convert byte to word
CWD	Convert word to doubleword
CWDE	Convert word to doubleword extended (80386/486 only)
CDQ	Convert doubleword to quadword (80386/486 only)

On the 80386/486, the **CWDE** instruction converts a signed 16-bit value in AX to a signed 32-bit value in EAX. The **CDQ** instruction converts a signed 32-bit value in EAX to a signed 64-bit value in the EDX:EAX register pair.

This example converts signed integers using **CBW**, **CWD**, **CWDE**, and **CDQ**.

```
        .DATA
mem8    SBYTE   -5
mem16   SWORD   -5
mem32   SDWORD  -5
        .CODE
        .
        .
        .
        mov     al, mem8        ; Load 8-bit -5 (FBh)
        cbw                     ; Convert to 16-bit -5 (FFFBh) in AX

        mov     ax, mem16       ; Load 16-bit -5 (FFFBh)
        cwd                     ; Convert to 32-bit -5 (FFFF:FFFBh)
                                ;  in DX:AX
        mov     ax, mem16       ; Load 16-bit -5 (FFFBh)
        cwde                    ; Convert to 32-bit -5 (FFFFFFFBh)
                                ;  in EAX
        mov     eax, mem32      ; Load 32-bit -5 (FFFFFFFBh)
        cdq                     ; Convert to 64-bit -5
                                ;  (FFFFFFFF:FFFFFFFBh) in EDX:EAX
```

Conversion instructions do not operate on unsigned numbers.

The procedure is different for unsigned values. Unsigned values are extended by filling the upper bits with zeros rather than by sign extension. Because the sign-extend instructions do not work on unsigned integers, you must set the value of the higher register to zero.

This example shows sign extension for unsigned numbers.

```
        .DATA
mem8    BYTE    251
mem16   WORD    251
        .CODE
        .
        .
        .
        mov     al, mem8  ; Load 251 (FBh) from 8-bit memory
        sub     ah, ah    ; Zero upper half (AH)

        mov     ax, mem16 ; Load 251 (FBh) from 16-bit memory
        sub     dx, dx    ; Zero upper half (DX)
```

The 80386/486 processors provide instructions that move and extend a value to a larger data size in a single step. **MOVSX** moves a signed value into a register and sign-extends it. **MOVZX** moves an unsigned value into a register and zero-extends it.

```
; 80386/486 instructions
        movzx   dx, bl      ; Load unsigned 8-bit value into
                            ;  16-bit register and zero-extend
```

These special 80386 and 80486 instructions usually execute much faster than the equivalent 8086-80286 instructions.

4.2.2 Pushing and Popping Stack Integers

A stack is an area of memory for storing data temporarily. Unlike other segments that store data starting from low memory, the stack stores data in reverse order—starting from high memory. Data is always pushed or popped from the top of the stack. The data on the stack can be the calling addresses of procedures or interrupts, procedure arguments, or any operands, flags, or registers your program needs to store temporarily.

At first, the stack is an uninitialized segment of a finite size. As data is added to the stack at run time, the stack grows downward from high memory to low memory. When items are removed from the stack, it shrinks upward from low to high memory.

4.2.2.1 Saving Operands on the Stack

PUSH and POP always operate on word-sized data.

The **PUSH** instruction stores a two-byte operand on the stack. The **POP** instruction retrieves a previously pushed value. When a value is pushed onto the stack, the assembler decreases the SP (Stack Pointer) register by 2. On 8086-based processors, the SP register always points to the top of the stack. The **PUSH** and **POP** instructions use the SP register to keep track of the current position.

When a value is popped off the stack, the assembler increases the SP register by 2. Although the stack always contains word values, the SP register points to byte addresses. Thus, SP changes in multiples of two. When a **PUSH** or **POP** instruction executes in a 32-bit code segment (one with **USE32** use type), the assembler transfers a four-byte value, and ESP changes in multiples of four.

NOTE The 8086 and 8088 processors differ from later Intel processors in how they push and pop the SP register. If you give the statement `push sp` with the 8086 or 8088, the word pushed is the word in SP after the push operation.

Figure 4.2 illustrates how pushes and pops change the SP register.

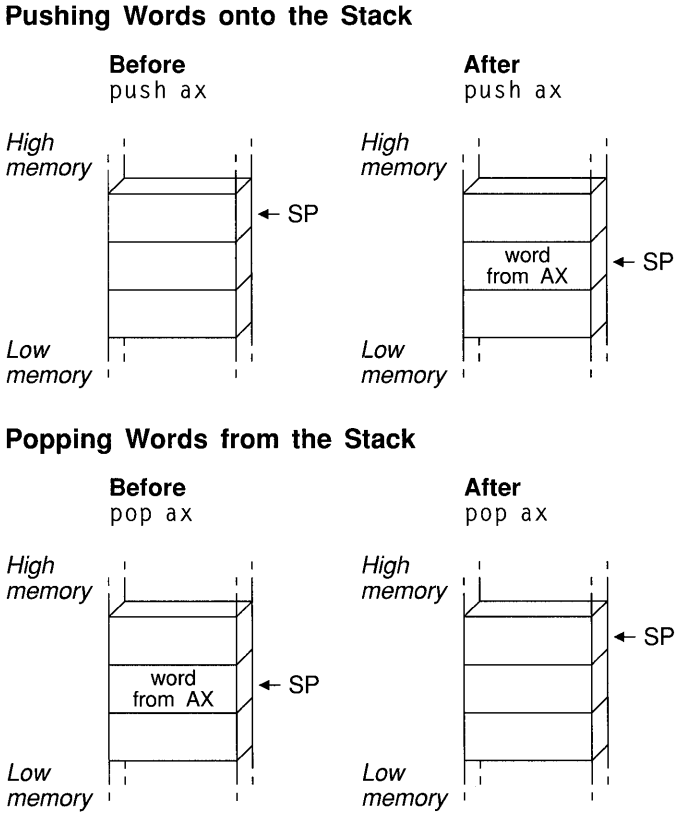


Figure 4.2 Stack Status before and after Pushes and Pops

On the 8086, **PUSH** and **POP** take only registers or memory expressions as their operands. The other processors allow an immediate value to be an operand for **PUSH**. For example, the following statement is legal on the 80186–80486 processors:

```
push     7              ; 3 clocks on 80286
```

That statement is faster than these equivalent statements, which are required on the 8088 or 8086:

```
mov      ax, 7          ; 2 clocks plus
push     ax             ; 3 clocks on 80286
```

There are two ways to clean up the stack.

Words are popped off the stack in reverse order: the last item pushed is the first popped. To return the stack to its original status, you can do the same number of pops as pushes. You can subtract the correct number of words from the SP register if you want to restore the stack without using the values on it.

To reference operands on the stack, keep in mind that the values pointed to by the BP (Base Pointer) and SP registers are relative to the SS (Stack Segment) register. The BP register is often used to point to the base of a frame of reference (a stack frame) within the stack.

This example shows how you can access values on the stack using indirect memory operands with BP as the base register.

```
push     bp             ; Save current value of BP
mov      bp, sp         ; Set stack frame
push     ax             ; Push first;  SP = BP - 2
push     bx             ; Push second; SP = BP - 4
push     cx             ; Push third;  SP = BP - 6
.
.
.
mov      ax, [bp-6]     ; Put third in AX
mov      bx, [bp-4]     ; Put second in BX
mov      cx, [bp-2]     ; Put first in CX
.
.
.
add      sp, 6          ; Restore stack pointer
                        ;  two bytes per push
pop      bp             ; Restore BP
```

Creating labels for stack variables makes code easier to read.

If you use these stack values often in your program, you may want to give them labels. For example, you can use **TEXTEQU** to create a label such as `count TEXTEQU <bp-6>`. Now you can replace the `mov ax, [bp - 6]` statement in the example above with `mov ax, count`. Section 9.1, "Text Macros," gives more information about the **TEXTEQU** directive.

4.2.2.2 Saving Flags on the Stack

Flags can be pushed and popped onto the stack with the **PUSHF** and **POPF** instructions. You can use these instructions to save the status of flags before a procedure call and then to restore the original status after the procedure. You can also use them within a procedure to save and restore the flag status of the caller. The 32-bit versions of these instructions are **PUSHFD** and **POPFD.**

This example saves the flags register before calling the `systask` procedure:

```
pushf
call    systask
popf
```

If you do not need to store the entire flag register, you can use the **LAHF** instruction to manually load and store the status of the lower byte of the flag register in the AH register. (You need to save AH before making a procedure call.) **SAHF** restores the value.

4.2.2.3 Saving Registers on the Stack (80186–80486 Only)

Starting with the 80186 processor, the **PUSHA** and **POPA** instructions push or pop all the general-purpose registers with only one instruction. These instructions save the status of all registers before a procedure call and then restore them after the return. Using **PUSHA** and **POPA** is significantly faster and takes fewer bytes of code than pushing and popping each register individually.

The processor pushes the registers in the following order: AX, CX, DX, BX, SP, BP, SI, and DI. The SP word pushed is the value before the first register is pushed.

The processor pops the registers in the opposite order. The 32-bit versions of these instructions are **PUSHAD** and **POPAD.**

4.2.3 Adding and Subtracting Integers

You can use the **ADD, ADC, INC, SUB, SBB,** and **DEC** instructions for adding, incrementing, subtracting, and decrementing values in single registers. You can also combine them to handle larger values that require two registers for storage.

4.2.3.1 Adding and Subtracting Integers Directly

The **ADD, INC** (Increment), **SUB,** and **DEC** (Decrement) instructions operate on 8- and 16-bit values on the 8086–80286 processors, and on 8-, 16-, and 32-bit values on the 80386/486 processors. They can be combined with the **ADC** and **SBB** instructions to work on 32-bit values on the 8086 and 64-bit values on the 80386/486 processors (see Section 4.2.3.2).

These instructions have two requirements:

1. If there are two operands, only one operand can be a memory operand.
2. If there are two operands, both must be the same size.

PTR allows you to operate on data in sizes different from its declared type.

To meet the second requirement, you can use the **PTR** operator to force an operand to the size required (see Section 4.2, "Integer Operations"). For example, if `Buffer` is an array of bytes and BX points to an element of the array, you can add a word from `Buffer` with

```
        add     ax, WORD PTR Buffer[bx] ; Adds a word from the
                                        ;  byte variable
```

The next example shows 8-bit signed and unsigned addition and subtraction.

```
        DATA
mem8    BYTE    39
        .CODE

; Addition

                        ;                       signed    unsigned
        mov     al, 26  ; Start with register    26        26
        inc     al      ; Increment               1         1
        add     al, 76  ; Add immediate          76      + 76
                        ;                       ----      ----
                        ;                        103       103
        add     al, mem8; Add memory             39      + 39
                        ;                       ----      ----
        mov     ah, al  ; Copy to AH            -114       142
                                                +overflow
        add     al, ah  ; Add register                    142
                        ;                                 ----
                        ;                                  28+carry

; Subtraction

                        ;                     signed    unsigned
        mov     al, 95  ; Load register         95        95
        dec     al      ; Decrement             -1        -1
        sub     al, 23  ; Subtract immediate    -23       -23
                        ;                       ----      ----
                        ;                         71        71
        sub     al, mem8; Subtract memory       -122      -122
                        ;                       ----      ----
                        ;                        -51       205+sign

        mov     ah, 119 ; Load register          119
        sub     al, ah  ;  and subtract          -51
                        ;                       ----
                        ;                         86+overflow
```

The **INC** and **DEC** instructions treat integers as unsigned values and do not update the carry flag for signed carries and borrows.

Your programs must include error-recovery for overflows and carries.

When the sum of eight-bit signed operands exceeds 127, the processor sets the overflow flag. (The overflow flag is also set if both operands are negative and the sum is less than or equal to -128.) Placing a **JO** (Jump on Overflow) or **INTO** (Interrupt on Overflow) instruction in your program at this point can transfer control to error-recovery statements. When the sum exceeds 255, the processor sets the carry flag. A **JC** (Jump on Carry) instruction at this point can transfer control to error-recovery statements.

In the subtraction example above, the processor sets the sign flag if the result goes below 0. At this point, you can use a **JS** (Jump on Sign) instruction to transfer control to error-recovery statements.

4.2.3.2 Adding and Subtracting in Multiple Registers

You can add and subtract numbers larger than the register size on your processor with the **ADC** (Add with Carry) and **SBB** (Subtract with Borrow) instructions. If the operations prior to an **ADC** or **SBB** instruction do not set the carry flag, these instructions are identical to **ADD** and **SUB**. When you operate on large values in more than one register, use **ADD** and **SUB** for the least significant part of the number and **ADC** or **SBB** for the most significant part.

The following example illustrates multiple-register addition and subtraction. You can also use this technique with 64-bit operands on the 80386/486 processors.

```
        .DATA
mem32   DWORD   316423
mem32a  DWORD   316423
mem32b  DWORD   156739
        .CODE
        .
        .
        .
; Addition
        mov     ax, 43981               ; Load immediate     43981
        sub     dx, dx                  ;  into DX:AX
        add     ax, WORD PTR mem32[0]   ; Add to both     + 316423
        adc     dx, WORD PTR mem32[2]   ;  memory words     ------
                                        ; Result in DX:AX   360404

; Subtraction
        mov     ax, WORD PTR mem32a[0]  ; Load mem32        316423
        mov     dx, WORD PTR mem32a[2]  ;  into DX:AX
        sub     ax, WORD PTR mem32b[0]  ; Subtract low    - 156739
        sbb     dx, WORD PTR mem32b[2]  ;  then high        ------
                                        ; Result in DX:AX   159684
```

For 32-bit registers on the 80386/486, only two steps are necessary. If your program needs to be assembled for more than one processor, you can assemble the statements conditionally, as shown in this example:

```
        .DATA
mem32   DWORD   316423
mem32a  DWORD   316423
mem32b  DWORD   156739
p386    TEXTEQU (@Cpu AND 08h)
        .CODE
        .
        .
        .
; Addition
        IF      p386
        mov     eax, 43981  ; Load immediate
        add     eax, mem32  ; Result in EAX
        ELSE
        .
        .       ; do steps in previous example
        .
        ENDIF

; Subtraction
        IF      p386
        mov     eax, mem32a ; Load memory
        sub     eax, mem32b ; Result in EAX
        ELSE
        .
        .       ; do steps in previous example
        .
        ENDIF
```

Since the status of the carry flag affects the results of calculations with **ADC** and **SUB**, be sure to turn off the carry flag with the **CLC** (Clear Carry Flag) instruction or use **ADD** for the first calculation when appropriate.

4.2.4 Multiplying and Dividing Integers

The 8086 family of processors uses different multiplication and division instructions for signed and unsigned integers. Multiplication and division instructions also have special requirements depending on the size of the operands and the processor the code runs on.

4.2.4.1 Using Multiplication Instructions

The **MUL** instruction multiplies unsigned numbers. **IMUL** multiplies signed numbers. For both instructions, one factor must be in the accumulator register (AL for 8-bit numbers, AX for 16-bit numbers, EAX for 32-bit numbers). The other factor can be in any single register or memory operand. The result overwrites the contents of the accumulator register.

Multiplying two 8-bit numbers produces a 16-bit result returned in AX. Multiplying two 16-bit operands yields a 32-bit result in DX:AX. The 80386/486 processor handles 64-bit products in the same way in the EDX:EAX pair.

This example illustrates multiplication of signed 16- and 32-bit integers.

```
        .DATA
mem16   SWORD   -30000
        .CODE
        .
        .
        .
; 8-bit signed multiply
        mov     al, 23      ; Load AL                    23
        mov     bl, 24      ; Load BL                  * 24
        mul     bl          ; Multiply BL              -----
                            ; Product in AX              552
                            ;  overflow and carry set

; 16-bit unsigned multiply
        mov     ax, 50      ; Load AX                    50
                            ;                         -30000
        imul    mem16       ; Multiply memory          -----
                            ; Product in DX:AX      -1500000
                            ;  overflow and carry set
```

A nonzero number in the upper half of the result (AH for byte, DX or EDX for word) sets the overflow and carry flags.

On the 80186–80486 processors, the **IMUL** instruction supports three different operand combinations. The first syntax option allows for 16-bit multipliers producing a 16-bit product or 32-bit multipliers for 32-bit products on the 80386/486. The result overwrites the destination. The syntax for this operation is

IMUL *register16*, *immediate*

Multiplication by an immediate operand is possible on the 80386/486.

The second syntax option specifies three operands for **IMUL**. The first operand must be a 16-bit *register* operand, the second a 16-bit *memory* or *register* operand, and the third a 16-bit *immediate* operand. **IMUL** multiplies the memory (or register) and immediate operands and stores the product in the register operand with this syntax:

IMUL *register16*, *memory16* | *register16*, *immediate*

For the 80386/486 only, a third option for **IMUL** allows an additional operand for multiplication of a register value by a register or memory value. This is the syntax:

IMUL *register*,{*register* | *memory*}

The destination can be any 16-bit or 32-bit register. The source must be the same size as the destination.

In all of these options, products too large to fit in 16 or 32 bits set the overflow and carry flags. The following examples show these three options for **IMUL**.

```
        imul    dx, 456      ; Multiply DX times 456 on 80186-80486
        imul    ax, [bx],6   ; Multiply the value pointed to by BX
                             ;  by 6 and put the result in AX

        imul    dx, ax       ; Multiply DX times AX on 80386
        imul    ax, [bx]     ; Multiply AX by the value pointed to
                             ;  by BX on 80386
```

The **IMUL** instruction with multiple operands can be used for either signed or unsigned multiplication, since the 16-bit product is the same in either case. To get a 32-bit result, you must use the single-operand version of **MUL** or **IMUL**.

4.2.4.2 Using Division Instructions

The **DIV** instruction divides unsigned numbers, and **IDIV** divides signed numbers. Both return a quotient and a remainder.

Table 4.1 summarizes the division operations. The dividend is the number to be divided, and the divisor is the number to divide by. The quotient is the result. The divisor can be in any register or memory location except the registers where the quotient and remainder are returned.

Table 4.1 Division Operations

Size of Operand	Dividend Register	Size of Divisor	Quotient	Remainder
16 bits	AX	8 bits	AL	AH
32 bits	DX:AX	16 bits	AX	DX
64 bits (80386 and 80486)	EDX:EAX	32 bits	EAX	EDX

Unsigned division does not require careful attention to flags. The following examples illustrate signed division, which can be more complex.

```
        .DATA
mem16   SWORD   -2000
mem32   SDWORD  500000
        .CODE
        .
        .
        .
; Divide 16-bit unsigned by 8-bit
        mov     ax, 700              ; Load dividend      700
        mov     bl, 36               ; Load divisor DIV    36
        div     bl                   ; Divide BL       ------
                                     ; Quotient in AL      19
                                     ; Remainder in AH          16
```

```
; Divide 32-bit signed by 16-bit
        mov     ax, WORD PTR mem32[0] ; Load into DX:AX
        mov     dx, WORD PTR mem32[2] ;                 500000
        idiv    mem16                 ;              DIV -2000
                                      ; Divide memory   ------
                                      ; Quotient in AX    -250
                                      ; Remainder in DX          0

; Divide 16-bit signed by 16-bit
        mov     ax, WORD PTR mem16    ; Load into AX     -2000
        cwd                           ; Extend to DX:AX
        mov     bx,-421               ;               DIV -421
        idiv    bx                    ; Divide by BX     -----
                                      ; Quotient in AX       4
                                      ; Remainder in DX       -316
```

If the dividend and divisor are the same size, sign-extend or zero-extend the dividend so that it is the length expected by the division instruction. See Section 4.2.1.4, "Extending Signed and Unsigned Integers."

4.3 Manipulating Integers at the Bit Level

The instructions introduced so far in this chapter accessed integers at the byte or word level. The logical, shift, and rotate instructions described in this section, however, access the individual bits of the integers. You can use logical instructions to evaluate characters and do other text and screen operations. The shift and rotate instructions do similar tasks by shifting and rotating bits through registers. This section discusses some applications of these bit-level operations.

4.3.1 Logical Operations

The logical instructions—**AND**, **OR**, **XOR**, and **NOT**—operate on each bit in one operand and on the corresponding bit in the other. The following list shows how each instruction works. Except for **NOT**, these instructions require two integers of the same size.

Instruction	Sets a Bit to 1 under These Conditions
AND	Both corresponding bits in the operands have the value 1.
OR	Either of the corresponding bits in the operands has the value 1.
XOR	Either, but not both, of the corresponding bits in the operands has the value 1.
NOT	The corresponding bit in the operand is 0. (This instruction takes only one operand.)

NOTE Do not confuse logical instructions with the logical operators, which perform these operations at assembly time, not run time. Although the names are the same, the assembler recognizes the difference from context.

The following example shows the result of the **AND**, **OR**, **XOR**, and **NOT** instructions operating on a value in the AX register and in a mask. A mask is a binary or hexadecimal number with appropriate bits set for the intended operation.

```
mov     ax, 035h    ; Load value                    00110101
and     ax, 0FBh    ; Clear bit 2               AND 11111011
                    ;                               --------
                    ; Value is now 31h             00110001
or      ax, 016h    ; Set bits 4,2,1            OR  00010110
                    ;                               --------
                    ; Value is now 37h             00110111
xor     ax, 0ADh    ; Toggle bits 7,5,3,2,0     XOR 10101101
                    ;                               --------
                    ; Value is now 9Ah             10011010
not     ax          ; Value is now 65h             01100101
```

Use AND, OR, and XOR to set or clear specific bits.

You can use the **AND** instruction to clear the value of specific bits regardless of their current settings. To do this, put the target value in one operand and a mask of the bits you want to clear in the other. The bits of the mask should be 0 for any bit positions you want to clear and 1 for any bit positions you want to remain unchanged.

You can use the **OR** instruction to force specific bits to 1 regardless of their current settings. The bits of the mask should be 1 for any bit positions you want to set and 0 for any bit positions you want to remain unchanged.

You can use the **XOR** instruction to toggle the value of specific bits (reverse them from their current settings). This instruction sets a bit to 1 if the corresponding bits are different or to 0 if they are the same. The bits of the mask should be 1 for any bit positions you want to toggle and 0 for any bit positions you want to remain unchanged.

The following examples show an application for each of these instructions. The code illustrating the **AND** instruction converts a "y" or "n" read from the keyboard to uppercase, since bit 5 is always clear in uppercase letters. In the example for **OR**, the first statement is faster and uses fewer bytes than `cmp bx, 0`. When the operands for XOR are identical, each bit cancels itself, producing 0.

```
; Converts characters to uppercase
        mov     ah, 7           ; Get character without echo
        int     21h
        and     al, 11011111y   ; Convert to uppercase by clearing
                                ;  bit 5
        cmp     al, 'Y'         ; Is it Y?
        je      yes             ; If so, do Yes actions
        .                       ;  else do No actions
        .
yes:    .

; Compares operand to 0
        or      bx, bx          ; Compare to 0
                                ;  2 bytes, 2 clocks on 8088
        jg      positive        ; BX is positive
        jl      negative        ; BX is negative
                                ; else BX is zero

; Sets a register to 0
        xor     cx, cx          ; 2 bytes, 3 clocks on 8088
        sub     cx, cx          ; 2 bytes, 3 clocks on 8088
        mov     cx, 0           ; 3 bytes, 4 clocks on 8088
```

On the 80386 and 80486, the **BSF** (Bit Scan Forward) and the **BSR** (Bit Scan Reverse) instructions perform operations similar to those of the logical instructions. They scan the contents of a register to find the first-set or last-set bit. You can use **BSF** or **BSR** to find the position of a set bit in a mask or to check if a register value is 0.

4.3.2 Shifting and Rotating Bits

The 8086-based processors provide a complete set of instructions for shifting and rotating bits. Shift instructions move bits a specified number of places to the right or left. The last bit in the direction of the shift goes into the carry flag, and the first bit is filled with 0 or with the previous value of the first bit.

Rotate instructions also move bits a specified number of places to the right or left. For each bit rotated, the last bit in the direction of the rotate operation moves into the first bit position at the other end of the operand. With some variations, the carry bit is used as an additional bit of the operand. Figure 4.3 illustrates the eight variations of shift and rotate instructions for eight-bit operands. Notice that **SHL** and **SAL** are identical.

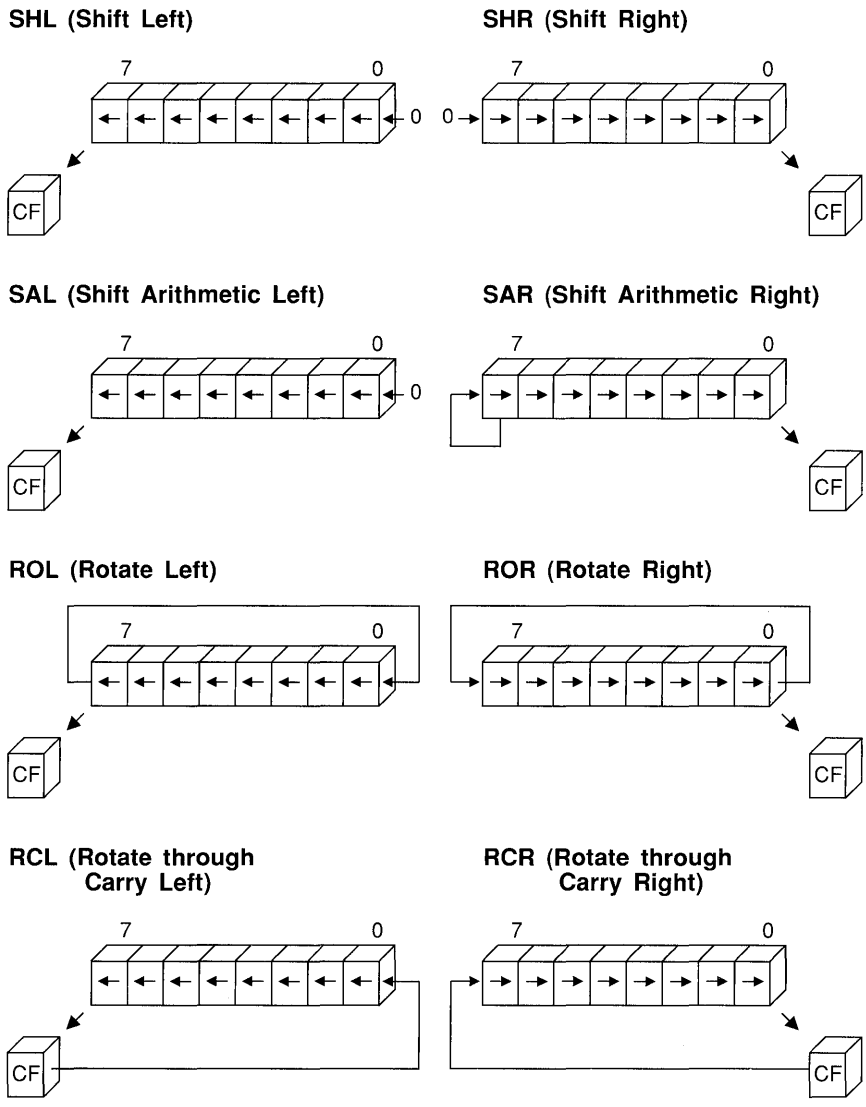


Figure 4.3 Shifts and Rotates

All shift instructions use the same format. Before the instruction executes, the destination operand contains the value to be shifted; after the instruction executes, it contains the shifted operand. The source operand contains the number of bits to shift or rotate. It can be the immediate value 1 or the CL register. The 8088 and 8086 processors do not accept any other values or registers with these instructions.

The shift instruction allows you to change masks during program execution.

Masks for logical instructions can be shifted to new bit positions. For example, an operand that masks off a bit or group of bits can be shifted to move the mask to a different position, allowing you to mask off a different bit each time the mask is used. This technique, illustrated in the following example, is useful only if the mask value is unknown until run time.

```
        .DATA
masker  BYTE    00000010y   ; Mask that may change at run time
        .CODE
        .
        .
        .
        mov     cl, 2       ; Rotate two at a time
        mov     bl, 57h     ; Load value to be changed 01010111y
        rol     masker, cl  ; Rotate two to left       00001000y
        or      bl, masker  ; Turn on masked values    ---------
                            ; New value is 05Fh        01011111y
        rol     masker, cl  ; Rotate two more          00100000y
        or      bl, masker  ; Turn on masked values    ---------
                            ; New value is 07Fh        01111111y
```

Starting with the 80186 processor, you can use eight-bit immediate values larger than 1 as the source operand for shift or rotate instructions, as shown below:

```
        shr     bx, 4   ;  9 clocks, 3 bytes on 80286
```

The following statements are equivalent if the program must run on the 8088 or 8086 processor:

```
        mov     cl, 4   ;  2 clocks, 3 bytes on 80286
        shr     bx, cl  ;  9 clocks, 2 bytes on 80286
                        ; 11 clocks, 5 bytes
```

4.3.3 Multiplying and Dividing with Shift Instructions

You can use the shift and rotate instructions (**SHR**, **SHL**, **SAR**, and **SAL**) for multiplication and division. Shifting an integer right by one bit has the effect of dividing by two; shifting left by one bit has the effect of multiplying by two. You can take advantage of shifts to do fast multiplication and division by powers of two. For example, shifting left twice multiplies by four, shifting left three times multiplies by eight, and so on.

Use **SHR** (Shift Right) to divide unsigned numbers. You can use **SAR** (Shift Arithmetic Right) to divide signed numbers, but **SAR** rounds numbers down—**IDIV** always rounds up. Division using **SAR** must adjust for this difference. Multiplication by shifting is the same for signed and unsigned numbers, so you can use either **SAL** or **SHL**.

Use shifts instead of MUL or DIV to optimize your code.

Since the multiply and divide instructions are very slow on the 8088 and 8086 processors, using shifts instead can often speed operations by a factor of 10 or more. For example, on the 8088 or 8086 processor, these statements take only four clocks:

```
        sub     ah, ah      ; Clear AH
        shl     ax, 1       ; Multiply byte in AL by 2
```

The following statements produce the same results, but take between 74 and 81 clocks on the 8088 or 8086. The same statements take 15 clocks on the 80286 and between 11 and 16 clocks on the 80386.

```
        mov     bl, 2       ; Multiply byte in AL by 2
        mul     bl
```

You can put multiplication and division operations in macros so they can be changed if the constants in a program change, as shown in the two macros below.

```
mul_10  MACRO   factor      ; Factor must be unsigned
        mov     ax, factor  ; Load into AX
        shl     ax, 1       ; AX = factor * 2
        mov     bx, ax      ; Save copy in BX
        shl     ax, 1       ; AX = factor * 4
        shl     ax, 1       ; AX = factor * 8
        add     ax, bx      ; AX = (factor * 8) + (factor * 2)
        ENDM                ; AX = factor * 10

div_512 MACRO   dividend    ; Dividend must be unsigned
        mov     ax, dividend ; Load into AX
        shr     ax, 1       ;  AX = dividend / 2 (unsigned)
        xchg    al, ah      ; xchg is like rotate right 8
                            ;  AL = (dividend / 2) / 256
        cbw                 ; Clear upper byte
        ENDM                ;  AX = (dividend / 512)
```

Since RCR and RCL use the carry flag, clear it before multiple-register shifts.

If you need to shift a value that is too large to fit in one register, you can shift each part separately. The **RCR** (Register Carry Right) and **RCL** (Register Carry Left) instructions carry values from the first register to the second by passing the leftmost or rightmost bit through the carry flag.

This example shifts a multiword value.

```
        .DATA
mem32    DWORD  500000
        .CODE

; Divide 32-bit unsigned by 16
        mov     cx, 4                   ; Shift right 4        500000
again:  shr     WORD PTR mem32[2], 1 ; Shift into carry  DIV    16
        rcr     WORD PTR mem32[0], 1 ; Rotate carry in      ------
        loop    again                   ;                       31250
```

Since the carry flag is treated as part of the operand (it's like using a nine-bit or 17-bit operand), the flag value before the operation is crucial. The carry flag can be set by a previous instruction, but you can also set it directly by using the **CLC** (Clear Carry Flag), **CMC** (Complement Carry Flag), and **STC** (Set Carry Flag) instructions.

On the 80386 and 80486, an alternate method for multiplying quickly by constants takes advantage of the **LEA** (Load Effective Address) instruction and the scaling of indirect memory operands. By using a 32-bit value as both the index and the base register in an indirect memory operand, you can multiply by the constants 2, 3, 4, 5, 8, and 9 more quickly than you can by using the **MUL** instruction. **LEA** calculates the offset of the source operand and stores it into the destination register, EBX, as this example shows:

```
lea     ebx, [eax*2]        ; EBX = 2 * EAX
lea     ebx, [eax*2+eax]    ; EBX = 3 * EAX
lea     ebx, [eax*4]        ; EBX = 4 * EAX
lea     ebx, [eax*4+eax]    ; EBX = 5 * EAX
lea     ebx, [eax*8]        ; EBX = 8 * EAX
lea     ebx, [eax*8+eax]    ; EBX = 9 * EAX
```

Section 3.2.4.3, "Indirect Memory Operands with 32-Bit Registers," discusses scaling of 80386 indirect memory operands, and Section 3.3.3.2, "Loading Addresses into Registers," introduces **LEA**.

This chapter has covered the integer operations you use in your MASM programs. The next chapter looks at more complex data types—arrays, strings, structures, unions, and records. Many of the operations presented in this chapter can also be applied to the data structures discussed in Chapter 5, "Defining and Using Complex Data Types."

4.4 Related Topics in Online Help

Online help features additional information about the topics discussed in this chapter. From the "MASM 6.0 Contents" screen for MASM online help, select the following topics:

Topic	Access
BYTE, WORD, ...	Choose "Directives" and then "Data Allocation"
Bitwise logical operations	Choose "Operators" and then from the list of operators, choose "Logical and Shift"
Location counter	Choose "Predefined Symbols" for information on the $ symbol

Topic	Access
BSF, **BSR**, **SHLD**, **SHRD**, and **SET***condition*	From the "Processor Instructions" categories, choose "Logical and Shift"
LES, **LFS**, **LGS**	From the "Processor Instructions" categories, choose "Data Transfer"
.RADIX directive	Choose "Directives" and then choose "Miscellaneous"
MOD	Choose "Operators," and then "Arithmetic"
OPATTR, **.TYPE**, **HIGH**, **LOW**, **HIGHWORD**, and **LOWWORD**	Choose "Operators," then "Miscellaneous"
OPTION EXPR32, **OPTION EXPR16**,	Choose "Directives," and then "OPTION"

Chapter 5
Defining and Using Complex Data Types

With the complex data types available in MASM 6.0—arrays, strings, records, structures, and (new to version 6.0) unions—you can access data either as a unit or as individual elements that make up the unit. The individual elements of complex data types are often the integer types discussed in Chapter 4, "Defining and Using Integers."

Section 5.1 first discusses how to declare, reference, and initialize arrays and strings. This section summarizes the general steps needed to process arrays and strings and describes the MASM instructions for moving, comparing, searching, loading, and storing operations.

Section 5.2 covers similar information for structures and unions: how to declare structure and union types, how to define structure and union variables, and how to reference structures and unions and their fields.

Section 5.3 explains how to declare record types, define record variables, and use record operators.

All three sections also describe how to use the **LENGTHOF**, **SIZEOF**, and **TYPE** operators with each complex data type.

5.1 Arrays and Strings

An assembly-language array is a sequence of fixed-size variables. A string is an array of characters. You can access the elements in an array or string relative to the first element.

This section explains and illustrates the essential ways to handle arrays and strings in your programs. It covers arrays first, beginning with the two ways to declare an array and continuing with how to reference it. The section then explains the special requirements for declaring and initializing a string. Finally, it describes the processing of arrays and strings.

5.1.1 Declaring and Referencing Arrays

You can declare an array in two ways: you can specify a list of array elements, or you can use the **DUP** operator to specify a group of identical elements.

To declare an array, you must supply a label name, a type, and a series of elements separated by commas. You can access each element of an array relative to the first. In the examples below, `warray` and `xarray` are arrays.

```
warray  WORD    1, 2, 3, 4
xarray  DWORD   OFFFh, OAAAh
```

The assembler stores the elements consecutively in memory, with the first address referenced by the label name.

Initializer lists can be longer than one line.

Beginning with MASM 6.0, initializer lists of array declarations can span multiple lines. The first initializer must appear on the same line as the data type, all entries must be initialized, and, if you want the array to continue to the new line, the line must end with a comma. These examples show legal multiple-line array declarations:

```
big             BYTE    21, 22, 23, 24, 25,
                        26, 27, 28

somelist        WORD    10,
                        20,
                        30
```

If you do not want to use the new **LENGTHOF** and **SIZEOF** operators discussed later in this section, then an array may span more than one logical line, although a separate type declaration is needed on each logical line:

```
var1    BTYE    10, 20, 30
        BYTE    40, 50, 60
        BYTE    70, 80, 90
```

The DUP Operator

You can also declare an array with the **DUP** operator. This operator can be used with any of the data allocation directives described in Section 4.1.1. In the syntax

count **DUP** (*initialvalue* [[, *initialvalue*]]...)

the *count* value sets the number of times to repeat the last *initialvalue*. Each initial value is evaluated only once and can be any expression that evaluates to an integer value, a character constant, or another **DUP** operator. The initial value (or values) must always be placed within parentheses. For example, the statement

```
barray  BYTE    5 DUP (1)
```

allocates the integer `1` five times for a total of five bytes.

The following examples show various ways to use the **DUP** operator to allocate data elements.

```
array   DWORD   10 DUP (1)                    ; 10 doublewords
                                               ;  initialized to 1
buffer  BYTE    256 DUP (?)                    ; 256-byte buffer

masks   BYTE    20 DUP (040h, 020h, 04h, 02h) ; 80-byte buffer
                                               ;  with bit masks
three_d DWORD   5 DUP (5 DUP (5 DUP (0)))      ; 125 doublewords
                                               ;  initialized to 0
```

Referencing Arrays

Once an array is defined, you can refer to its first element by typing the array name (no brackets required). The array name refers to the first object of the given type in the list of initial values.

If `warray` has been defined as

```
warray WORD 2, 4, 6, 8, 10
```

then referencing `warray` in your program refers to the first word—the word containing `2`.

To refer to the next element (in an array of words), use either of these two forms, each of which refers to the array element two bytes past the beginning of `warray`:

```
warray+2
warray[2]
```

This element can be used as you would any data item:

```
mov     ax, warray[2]
push    warray+2
```

When used with a variable name, brackets only add a number to the address. If `warray` refers to the address `2400h`, then `warray[2]` refers to the address `2402h`. The **BOUND** instruction (80186–80486 only) can be used to verify that an index value is within the bounds of an array.

Array indexes are not scaled. The index is a distance in bytes.

In assembly language, array indexes are zero-based and unscaled. The number within brackets always represents an absolute distance in bytes. In practical terms, the fact that indexes are unscaled means that if an element is larger than one byte, you must multiply the index of the element by its size (in the example above, 2), and then add the result to the address of the array. Thus, the expression `warray[4]` represents the third element, which is four bytes past the beginning of the array. Similarly, the expression `warray[6]` represents the fourth element.

You can also determine an index at run time:

```
mov     si, cx          ; CX holds index value
shl     si, 7           ; Scale for word referencing
mov     ax, warray[si]  ; Move element into AX
```

The offset required to access an array element can be calculated with the following formula:

*n*th element of array = array[(*n*-1) * size of element]

LENGTHOF, SIZEOF, and TYPE for Arrays

When applied to arrays, the **LENGTHOF**, **SIZEOF**, and **TYPE** operators return information about the length and size of the array and about the type of the initializers.

The **LENGTHOF** operator returns the number of items in the definition. It can be applied only to an integer label. This is useful for determining the number of elements you need to process in an array of integers. For an array or string label, **SIZEOF** returns the number of bytes used by the initializers in the definition. **TYPE** returns the size of the elements of the array. These examples illustrate these operators:

```
array   WORD    40 DUP (5)

larray  EQU     LENGTHOF array    ; 40 elements
sarray  EQU     SIZEOF   array    ; 80 bytes
tarray  EQU     TYPE     array    ;  2 bytes per element

num     DWORD   4, 5, 6, 7,
                8, 9, 10, 11

lnum    EQU     LENGTHOF num      ;  8 elements
snum    EQU     SIZEOF   num      ; 32 bytes
tnum    EQU     TYPE     num      ;  4 bytes per element

warray  WORD    40 DUP (40 DUP (5))

len     EQU     LENGTHOF warray   ; 1600 elements
siz     EQU     SIZEOF   warray   ; 3200 bytes
typ     EQU     TYPE     warray   ;    2 bytes per element
```

5.1.2 Declaring and Initializing Strings

A string is an array of bytes. Initializing a string like `"Hello, there"` allocates and initializes one byte for each character in the string. An initialized string can be no longer than 255 characters.

Strings declared with types other than BYTE must fit the memory space allocated.

For data directives other than **BYTE**, a string may initialize only a single element. This element must be short enough to fit into the specified size and conform to the expression word size in effect (see Section 1.2.4,"Integer Constants and Constant Expressions"), as shown in these examples:

```
wstr    WORD    "OK"
dstr    DWORD   "ADCD"  ; Legal under EXPR32 only
```

As with arrays, string initializers can span multiple lines. The line must end with a comma if you want the string to continue to the next line.

```
str1    BYTE    "This is a long string that does not ",
                "fit on one line."
```

You can also have an array of pointers to strings. For example:

```
PBYTE   TYPEDEF PTR BYTE
        .DATA
msg1    BYTE    "Operation completed successfully."
msg2    BYTE    "Unknown command"
msg3    BYTE    "File not found"
pmsg1   PBYTE   msg1
pmsg2   BPBYTE  msg2
pmsg3   PBYTE   msg3

errors  WORD    pmsg1, pmsg2, pmsg3     ; An array of pointers
                                        ;  to strings
```

Strings must be enclosed in single (') or double (") quotation marks. To put a single quotation mark inside a string enclosed by single quotation marks, use two single quotation marks. Likewise, if you need quotation marks inside a string enclosed by double quotation marks, use two sets. These examples show the various uses of quotation marks:

```
char    BYTE    'a'
message BYTE    "That's the message."       ; That's the message.
warn    BYTE    'Can''t find file.'         ; Can't find file.
string  BYTE    "This ""value"" not found." ; This "value"
                                              not found.
```

You can always use single quotation marks inside a string enclosed by double quotation marks, as the initialization for `message` shows, and vice versa.

The ? Initializer

The actual values stored when you use ? depend on the other data in your program.

You do not have to initialize all elements in an array to a value. If there is no initial value, you can initialize the array elements with the **?** operator. The **?** operator either is treated as a zero or causes a byte to be left unspecified in the object file. Object files contain records for initialized data. An unspecified byte left in the object file means that no records contain initialized data for that address.

The actual values stored in arrays allocated with **?** depend on certain conditions. The **?** initializer is treated as a zero in a **DUP** statement that contains initializers in addition to the **?** initializer. An unspecified byte is left in the object file if the **?** initializer does not appear in a **DUP** statement, or if the **DUP** statement contains only **?** initializers for nested **DUP** statements.

Length-Specified Strings

Often there are reasons to know the length of a string. To use the DOS functions for writing to a file, for example, CX must contain the length of the string before the interrupt is called, as shown in this example.

```
msg     BYTE    "This is a length-specified string"
        .
        .
        .
        mov     ah, 40h
        mov     bx, 1
        mov     cx, LENGTHOF msg
        mov     dx, OFFSET msg
        int     21h
```

Some high-level languages also expect strings passed to procedures to have a certain format. For example, Pascal procedures require the first byte of a string passed as a parameter to contain the length of the string. You can write this length into the first byte with

```
msg     BYTE    LENGTHOF msg - 1, "This is a Pascal string"
```

Interfacing with high-level languages requires special techniques with strings.

Other languages such as Basic have string descriptions—a kind of structure containing both the length and the address of the string. For example, this structure `DESC` could be used in a procedure accessed from Basic:

```
DESC    STRUCT
  len   WORD    ?       ; Length of string1
  off   WORD    ?       ; Offset of string1
DESC    ENDS

string1 BYTE    "This string goes in a string descriptor"
msg     DESC    {LENGTHOF string1, string1}
```

See Section 5.2, "Structures and Unions."

Null-Terminated and $-Terminated Strings

Null-terminated and $-terminated strings have a special use with DOS functions. Strings in modules shared with C need to end with a null character (0).

```
str1    BYTE    "This string ends with a null character", 0
```

DOS file names also require a null character at the end. This example opens a file named `"MYFILE.ASM"`.

```
name1   BYTE    "MYFILE.ASM", 0
        .
        .
        .
        mov     ah, 3Dh
        mov     dx, OFFSET name1
        int     21h
```

DOS function 9 requires a string to end with a dollar sign ($) so that it can recognize the end of the string to write to the screen, as shown in this example.

```
msg     BYTE    "This is a dollar-terminated string$"
        .
        .
        .
        mov     ah, 09h
        mov     dx, OFFSET msg
        int     21h
```

LENGTHOF, SIZEOF, and TYPE for Strings

Because the assembler considers strings as simply arrays of byte elements, the **LENGTHOF** and **SIZEOF** operators return the same values for strings as they do for arrays, as illustrated in this example. The **TYPE** operator considers `msg` to be one data unit and returns 1.

```
msg     BYTE    "This string extends ",
                "over three ",
                "lines."

lmsg    EQU     LENGTHOF msg      ; 37 elements
smsg    EQU     SIZEOF   msg      ; 37 bytes
tmsg    EQU     TYPE     msg      ;  1 byte per element
```

5.1.3 Processing Arrays and Strings

The 8086-family instruction set has seven string instructions for fast and efficient processing of entire strings and arrays. The term "string" in "string instructions" refers to a sequence of elements, not just character strings. These instructions work directly only on arrays of bytes and words on the 8086–80486 and on arrays of bytes, words, and doublewords on the 80386 and 80486. Processing larger elements must be done indirectly with loops.

The following list gives capsule descriptions of the five instructions discussed in this section. Two additional instructions not described here are the **INS** and **OUTS** instructions that transfer values to and from a memory port.

Instruction	Description
MOVS	Copies a string from one location to another
STOS	Stores values from the accumulator register to a string
CMPS	Compares values in one string with values in another
LODS	Loads values from a string to the accumulator register
SCAS	Scans a string for a specified value

All of these instructions use registers in a similar way and have a similar syntax. Most are used with the repeat instruction prefixes **REP**, **REPE** (or **REPZ**), and **REPNE** (or **REPNZ**). **REPZ** is a synonym for **REPE** (Repeat While Equal) and **REPNZ** is a synonym for **REPNE** (Repeat While Not Equal).

This section first explains the general procedures for using all string instructions. It then illustrates each instruction with an example.

5.1.3.1 Overview of String Operations

The string instructions have specific requirements for the location of strings and the use of registers. To operate on any string, follow these three steps:

All string operations follow three basic steps.

1. Set the direction flag to indicate the direction in which you want to process the string. The **STD** instruction sets the flag, while **CLD** clears it.

   If the direction flag is clear, the string is processed upward (from low addresses to high addresses, which is from left to right through the string). If the direction flag is set, the string is processed downward (from high addresses to low addresses, or from right to left). Under DOS, the direction flag is normally clear if your program has not changed it.

2. Load the number of iterations for the string instruction into the CX register.

   If you want to process a 100-byte string, move 100 into CX. If you wish the string instruction to terminate conditionally (for example, during a search when a match is found), load the maximum number of iterations that can be performed without an error.

3. Load the starting offset address of the source string into DS:SI and the starting address of the destination string into ES:DI. Some string instructions take only a destination or source, not both (see Table 5.1).

   Normally, the segment address of the source string should be DS, but you can use a segment override to specify a different segment for the source operand. You cannot override the segment address for the destination string. Therefore, you may need to change the value of ES. See Section 3.1 for information on changing segment registers.

NOTE Although you can use a segment override on the source operand, a segment override combined with a repeat prefix can cause problems in certain situations on all processors except the 80386/486. If an interrupt occurs during the string operation, the segment override is lost and the rest of the string operation processes incorrectly. Segment overrides can be used safely when interrupts are turned off or with an 80386/486 processor.

You can adapt these steps to the requirements of any particular string operation. The syntax for the string instructions is:

[[*prefix*]] **CMPS** [[*segmentregister*:]] *source*, [[**ES**:]] *destination*
LODS [[*segmentregister*:]] *source*
[[*prefix*]] **MOVS** [[**ES**:]] *destination*, [[*segmentregister*:]] *source*
[[*prefix*]] **SCAS** [[**ES**:]] *destination*
[[*prefix*]] **STOS** [[**ES**:[[*destination*

Some instructions have special forms for byte, word, or doubleword operands. If you use the form of the instruction that ends in **B** (**BYTE**), **W** (**WORD**), or **D** (**DWORD**) with **LODS**, **SCAS**, and **STOS**, the assembler knows whether the element is in the AL, AX, or EAX register. Therefore, these instruction forms do not require operands.

Table 5.1 lists each string instruction with the type of repeat prefix it uses and indicates whether the instruction works on a source, a destination, or both.

Table 5.1 Requirements for String Instructions

Instruction	Repeat Prefix	Source/Destination	Register Pair
MOVS	**REP**	Both	DS:SI, ES:DI
SCAS	**REPE/REPNE**	Destination	ES:DI
CMPS	**REPE/REPNE**	Both	DS:SI, ES:DI
LODS	None	Source	DS:SI
STOS	**REP**	Destination	ES:DI
INS	**REP**	Destination	ES:DI
OUTS	**REP**	Source	DS:SI

The instruction automatically increments DI or SI.

The repeat prefix causes the instruction that follows it to repeat for the number of times specified in the count register or until a condition becomes true. After each iteration, the instruction increments or decrements SI and DI so that it points to new array elements. The string instructions work on these elements. The direction flag determines whether SI and DI are incremented (flag clear) or decremented (flag set). The size of the instruction determines whether SI and DI are altered by one, two, or four bytes each time.

These are the conditions that determine the number of repetitions specified by a prefix.

Prefix	Description
REP	Repeats instruction CX times
REPE, REPZ	Repeats instruction CX times, or as long as elements are equal, whichever is fewer
REPNE, REPNZ	Repeats instruction CX times, or as long as elements are not equal, whichever is fewer

The prefixes apply to only one string instruction at a time. To repeat a block of instructions, use a loop construction (see Section 7.2, "Loops").

At run time, if a string instruction is preceded by a repeat sequence, the processor takes the following steps:

1. Checks the CX register and exits if CX is 0. If the **REPE** prefix is used, the loop exits if the zero flag is set; if **REPNE** is used, the loop exits if the zero flag is clear.
2. Performs the string operation once.
3. Increases SI and/or DI if the direction flag is clear. Decreases SI and/or DI if the direction flag is set. The amount of increase or decrease is 1 for byte operations, 2 for word operations, and 4 for doubleword operations (80386/486 only).
4. Decrements CX (no flags are modified).
5. Checks the zero flag at this point if the **REPE** or **REPNE** prefix is used (for **SCAS** or **CMPS**). If the repeat condition does not hold, execution proceeds to the next instruction.
6. Proceeds to the next iteration and repeats from step 1.

At loop end, SI and DI point to the element immediately after the match.

When the repeat loop ends, SI (or DI) points to the position following a match (when using **SCAS** or **CMPS**), so you need to decrement or increment DI or SI to point to the element where the match occurred.

Although string instructions (except **LODS**) are most often used with repeat prefixes, they can also be used by themselves. In this case, the SI and/or DI registers are adjusted as specified by the direction flag and the size of operands. However, you must decrement the CX register and set up a loop for the repeated action.

5.1.3.2 String Instructions

To use the 8086-family string instructions, apply the steps outlined in the previous section. Examples in this section illustrate each instruction.

You can also use the techniques in this section with structures and unions, since arrays and strings can be fields in structures and unions (see Section 5.2).

Moving Array Data The **MOVS** instruction copies data from one area of memory to another. To move data, first load the count and the source and destination addresses into the appropriate registers. Then use **REP** with the **MOVS** instruction.

```
        .MODEL  small
        .DATA
source  BYTE    10 DUP ('0123456789')
destin  BYTE    100 DUP (?)
        .CODE
        mov     ax, @data           ; Load same segment
        mov     ds, ax              ;  to both DS
        mov     es, ax              ;  and ES
        .
        .
        .
        cld                         ; Work upward
        mov     cx, LENGTHOF source ; Set iteration count to 100
        mov     si, OFFSET source   ; Load address of source
        mov     di, OFFSET destin   ; Load address of destination
        rep     movsb               ; Move 100 bytes
```

Storing Data in Arrays The STOS instruction stores a specified value in each position of a string. The string is the destination, so it must be pointed to by ES:DI. The value to store must be in the accumulator.

This example stores the character `'a'` in each byte of a 100-byte string. Notice that it does this by storing 50 words rather than 100 bytes. This makes the code faster by reducing the number of iterations. To fill an odd number of bytes, you would have to adjust for the last byte.

```
        .MODEL  small, C
        .DATA
destin  BYTE    100 DUP (?)
ldestin EQU     (LENGTHOF destin) / 2
        .CODE
        .                           ; Assume ES = DS
        .
        .
        cld                         ; Work upward
        mov     ax, 'aa'            ; Load character to fill
        mov     cx, ldestin         ; Load length of string
        mov     di, OFFSET destin   ; Load address of destination
        rep     stosw               ; Store 'aa' into array
```

Comparing Arrays The CMPS instruction compares two strings and points to the address after which a match or nonmatch occurs. If the values are the same, the zero flag is set. Either string can be considered as the destination or the source unless a segment override is used.

This example using **CMPSB** assumes that the strings are in different segments. Both segments must be initialized to the appropriate segment register.

```
        .MODEL  large, C
        .DATA
string1 BYTE    "The quick brown fox jumps over the lazy dog"
        .FARDATA
string2 BYTE    "The quick brown dog jumps over the lazy fox"
lstring EQU     LENGTHOF string2
        .CODE
        mov     ax, @data           ; Load data segment
        mov     ds, ax              ;  into DS
        mov     ax, @fardata        ; Load far data segment
        mov     es, ax              ;  into ES
        .
        .
        .
        cld                         ; Work upward
        mov     cx, lstring         ; Load length of string
        mov     si, OFFSET string1  ; Load offset of string1
        mov     di, OFFSET string2  ; Load offset of string2
        repe    cmpsb               ; Compare
        jcxz    allmatch            ; CX is 0 if no nonmatch
        .
        .
        .
allmatch:                           ; Special case for all match
```

Loading Data from Arrays The **LODS** instruction loads a value from a string into a register. The string is the source; the value is in the accumulator. This instruction normally is not used with a repeat instruction prefix, since something must be done with each element before going on to the next.

The code in this example loads, processes, and displays each byte in a string of bytes.

```
        .DATA
info    BYTE    0, 1, 2, 3, 4, 5, 6, 7, 8, 9
linfo   WORD    LENGTHOF info
        .CODE
        .
        .
        .
        cld                     ; Work upward
        mov     cx, linfo       ; Load length
        mov     si, OFFSET info ; Load offset of source
        mov     ah, 2           ; Display character function
```

```
get:
        lodsb                       ; Get a character
        add     al, '0'             ; Convert to ASCII
        mov     dl, al              ; Move to DL
        int     21h                 ; Call DOS to display character
        loop    get                 ; Repeat
```

Searching Arrays The SCAS instruction scans a string for a specified value. As the loop executes, this instruction compares the value pointed to by DI with the value in the accumulator. If values are the same, the zero flag is set.

After a **REPNE SCAS**, the zero flag is cleared if no match was found. After a **REPE SCAS**, the zero flag is set if all values matched.

This example assumes that ES is not the same as DS and that the address of the string is stored in a pointer variable. The **LES** instruction loads the far address of the string into ES:DI.

```
        .DATA
string  BYTE    "The quick brown fox jumps over the lazy dog"
pstring PBYTE   string              ; Far pointer to string
lstring EQU     LENGTHOF string     ; Length of string
        .CODE
        .
        .
        .
        cld                         ; Work upward
        mov     cx, lstring         ; Load length of string
        les     di, pstring         ; Load address of string
        mov     al, 'z'             ; Load character to find
        repne   scasb               ; Search
        jcxz    notfound            ; CX is 0 if not found
        .                           ; ES:DI points to character
        .                           ;  after first 'z'
        .
notfound:                           ; Special case for not found
```

5.2 Structures and Unions

A structure is a group of possibly dissimilar data types and variable declarations that can be accessed as a unit or by any of its components. The fields within the structure can have different sizes and data types.

Unions are identical to structures, except that the fields of a union overlap in memory, which allows you to define different data formats for the same memory space. Unions can store different types of data depending on the situation. They can also store data as one data type and retrieve it as another data type.

Whereas each field in a structure has an offset relative to the first byte of the structure, all the fields in a union start at the same offset. The size of a structure

is the sum of its components, while the size of a union is the length of the longest field.

A MASM structure is similar to a **struct** in the C language, a **STRUCTURE** in FORTRAN, and a **RECORD** in Pascal. Unions in MASM are similar to unions in C and FORTRAN, and to variant records in Pascal.

Follow these steps when using structures and unions:

1. Declare a structure (or union) type.
2. Define one or more variables having that type.
3. Reference the fields directly or indirectly with the field (dot) operator.

You can use the entire structure or union variable or just the individual fields as operands in assembler statements. This section explains the allocating, initializing, and nesting of structures and unions.

MASM 6.0 extends the functionality of structures and also makes some changes to MASM 5.1 behavior. You can still retain MASM 5.1 behavior if you prefer by specifying **OPTION OLDSTRUCTS** in your program. See Section 1.3.2 for information about the **OPTION** directive, and Section 5.2.3 for information about referencing structures and unions.

5.2.1 Declaring Structure and Union Types

When you declare a structure or union type, you create a template for data that contains the sizes and, optionally, the initial values for fields in the structure or union but that allocates no memory.

The **STRUCT** keyword marks the beginning of a type declaration for a structure. (**STRUCT** and **STRUC** are synonyms.) **STRUCT** and **UNION** type declarations have the following format:

name **{STRUCT | UNION}** [[*alignment*]] [[**,NONUNIQUE**]]
fielddeclarations
name **ENDS**

The *fielddeclarations* are a series of one or more variable declarations. You can declare default initial values individually or with the **DUP** operator (see Section 5.2.2, "Defining Structure and Union Variables"). Section 5.2.3, "Referencing Structures, Unions, and Fields," explains the **NONUNIQUE** keyword. Structures and unions can also be nested in MASM 6.0 (see Section 5.2.4).

Initializing Fields

If you provide initializers for the fields of a structure or union when you declare the type, these initializers become the default value for the fields when you define a variable of that type. Section 5.2.2 explains default initializers.

When you initialize the fields of a union type, the type and value of the first field become the default value and type for the union. In this example of an initialized union declaration, the default type for the union is **DWORD**:

```
DWB     UNION
  d     DWORD   00FFh
  w     WORD    ?
  b     BYTE    ?
DWB     ENDS
```

If the size of the first member is less than the size of the union, the assembler initializes the rest of the union to zeros. When initializing strings in a type, make sure the initial values are long enough to accommodate the largest possible string.

Field Names

Structure and union field names in MASM 6.0 must be unique within a given nesting level because they represent the offset from the beginning of the structure to the corresponding field.

A nested structure has its own level.

In MASM 6.0, a label and a structure field may have the same name, but not a text macro and a field name. Also, field names between structures need not be unique. Field names do need to be unique if you place **OPTION M510** or **OPTION OLDSTRUCTS** in your code or use the /Zm option from the command line, since versions of MASM prior to 6.0 require unique field names (see Appendix A).

Alignment Value and Offsets for Structures

Data access to structures is faster on aligned fields than on unaligned fields. Therefore, alignment gains speed at the cost of space. Alignment improves access on 16-bit processors but makes no difference on code executing on an 8-bit 8088 processor.

The way the assembler aligns structure fields determines the amount of space required to store a variable of that type. Each field in a structure has an offset relative to 0. If you specify an *alignment* in the structure declaration (or with the /Z*p*n command-line option), the offset for each field may be modified by the *alignment* (or *n*).

The only values accepted for *alignment* are 1, 2, and 4. The default is 1. If the type declaration includes an *alignment*, the fields are aligned to the minimum of the field's size and the *alignment*. Any padding required to reach the correct offset for the field is added prior to allocating the field. The padding consists of zeros and always precedes the field.

If the number of bytes in the field is greater than the alignment value, the element will be padded such that the offset of the element is divisible by the alignment value. If the number of bytes is greater than or equal to the alignment value, the offset of the element is padded such that it is divisible by the element size.

The size of the structure must also be evenly divisible by the structure alignment value, so zeros may be added at the end of the structure.

If neither the *alignment* nor the /Zp command-line option is used, the offset is incremented by the size of each data directive. This is the same as a default *alignment* equal to 1. The *alignment* specified in the type declaration overrides the /Zp command-line option.

These examples show how offsets are determined:

```
STUDENT2    STRUCT  2   ; Alignment value is 2
  score     WORD    1   ; Offset is  0
  id        BYTE    2   ; Offset is  2
  year      DWORD   3   ; Offset is  4; one byte padding added
  sname     BYTE    4   ; Offset is  8
STUDENT2    ENDS
```

One byte of padding is added at the end of the first byte-sized field. Otherwise the offset of the `year` field would be 3, which is not divisible by the alignment value of 2. The size of this structure is now 9 bytes. Since 9 is not evenly divisible by 2, one byte of padding is added at the end of `student2`.

```
STUDENT4    STRUCT  4             ; Alignment value is 4
  sname     BYTE    1             ; Offset is  0
  score     WORD    10 DUP (100)  ; Offset is  2
  year      BYTE    2             ; Offset is 22; 1 byte padding
                                  ;  added so offset of next field
                                  ;  is divisible by 4
  id        DWORD   3             ; Offset is 24
STUDENT4    ENDS
```

The alignment value affects memory allocation of structure variables.

The alignment value affects the alignment of structure variables, so adding an alignment value affects memory usage. This feature provides compatibility with structures in Microsoft C.

With MASM 6.0, C programmers can use the H2INC utility to translate C structures to MASM (see Chapter 16).

5.2.2 Defining Structure and Union Variables

Once you have declared a structure or union type, variables of that type can be defined. For each variable defined, memory is allocated in the current segment in the format declared by the type. The syntax for defining a structure or union variable is:

[[*name*]] *typename* < [[*initializer* [[,*initializer*]]...]] >

[[*name*]] *typename* { [[*initializer* [[,*initializer*]]...]] }

[[*name*]] *typename constant* **DUP** ({ [[*initializer* [[,*initializer*]]...]] })

The *name* is the label assigned to the variable. If no name is given, the assembler allocates space for the variable but does not give it a symbolic name. The *typename* is the name of a previously declared structure or union type.

An *initializer* can be given for each field. The type of each initializer must be the type of the corresponding field defined in the type declaration. For unions, the type of the initializer must be the same as the type for the first field. An initialization list can also be repeated using the **DUP** operator.

The list of initializers can be broken only after a comma unless you use a line continuation character (\) at the end of the line. The last curly brace or angle bracket must appear on the same line as the last initializer. You can also use the line continuation character to extend a line as shown in the `Item4` declaration below. Angle brackets and curly braces can be intermixed in an initialization as long as they match. This example using the `ITEMS` structure illustrates the options for initializing lists:

```
ITEMS       STRUCT
  Iname     BYTE      'Item Name'
  Inum      WORD      ?
  ITYPE     UNION
    oldtype BYTE      0
    newtype WORD      ?
  ENDS
ITEMS       ENDS
        .
        .
        .
        .DATA
Item1   ITEMS   < >              ; Accepts default initializers
Item2   ITEMS   { }              ; Accepts default initializers
Item3   ITEMS   <'Bolts', 126>   ; Overrides default value of first
                                 ;  2 fields; use default of
                                 ;  the third field
Item4   ITEMS   { \
                 'Bolts',        ; Item name
                 126 \           ; Part number
                }
```

The angle brackets or curly braces are required even if no initial value is given, as in `Item1` and `Item2` in the example. If initial values are given for more than one field, the values must be separated by commas, as shown in `Item3`.

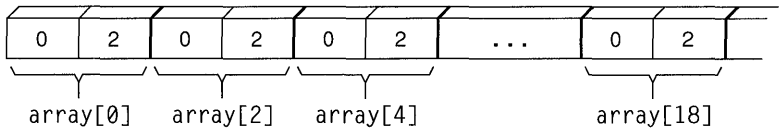
You need not initialize all fields in a structure. If an initial value is blank, the assembler automatically uses the default initial value of the field, which was originally provided in the structure type declaration. If there is no default value, the field is undefined.

For nested structures or unions (see Section 5.2.4), however, these are equivalent:

```
Item5   ITEMS   {'Bolts', ,     }
Item6   ITEMS   {'Bolts', , { } }
```

A variable and an array of union type `WB` look like this:

```
WB      UNION
  w     WORD    ?
  b     BYTE    ?
WB      ENDS

num     WB      {0Fh}                       ; Store 0Fh
array   WB      (40 / SIZEOF WB) DUP ({2})  ; Allocates and
                                            ;  initializes 10 unions
```



In MASM 6.0, control structures (such as **IF**, macros, and directives) are also allowed within structure and union declarations.

Arrays as Field Initializers

Default initializers for string or array fields set the size for the field.

The length of the array that can override the contents of a field in a variable definition is fixed by the size of the initializer. The override cannot contain more elements than the default. Specifying fewer override array elements changes the first *n* values of the default where *n* is the number of values in the override. The rest of the array elements take their default values from the initializer.

Strings as Field Initializers

If the override is shorter, the assembler pads the override with spaces to equal the length of the initializer. If the initializer is a string and the override value is not a string, the override value must be enclosed in angle brackets or curly braces.

A string may be used to override any member of type **BYTE** (or **SBYTE**). The string does not need to be enclosed in angle brackets or curly braces unless mixed with other override methods.

The string fields for structure variables are the length defined by the type declaration.

If a structure has an initialized string field or an array of bytes, any new string assigned to a variable of the field that is smaller than the default is padded with spaces. The assembler adds four spaces at the end of `'Bolts'` in the variables of type `ITEMS` above. The `Iname` field in the `ITEMS` structure cannot contain a field initializer longer than `'Item Name'`.

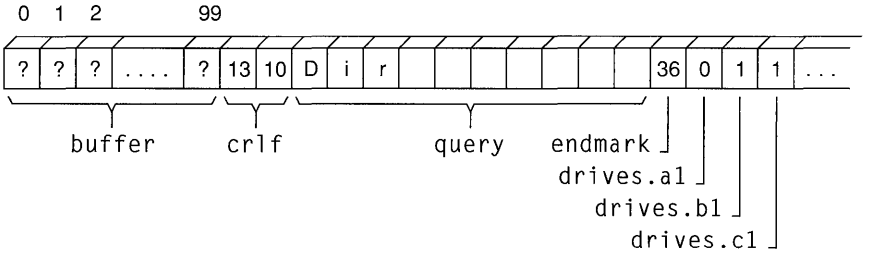
Structures as Field Initializers

Initializers for structure variables must be enclosed in curly braces or angle brackets, but you can specify overrides with fewer elements than the defaults.

This example illustrates the use of default values with structures as field initializers:

```
DISKDRIVES      STRUCT
  a1            BYTE ?
  b1            BYTE ?
  c1            BYTE ?
DISKDRIVES      ENDS

INFO            STRUCT
  buffer        BYTE    100 DUP (?)
  crlf          BYTE    13, 10
  query         BYTE    'Filename: ' ; String <= can override
  endmark       BYTE    36
  drives        DISKDRIVES <0, 1, 1>
INFO            ENDS

info1   INFO    { , , 'Dir' }

; Illegal since name in query field is too long
; and a string cannot initialize a field defined with DUP:
; info2  INFO    {"TESTFILE", , "DirectoryName",}

lotsof  INFO    { , , 'file1', , {0,0,0} },
                { , , 'file2', , {0,0,1} },
                { , , 'file3', , {0,0,2} }
```

The diagram below shows how the assembler stores `info1`.



The initialization for `drives` gives default values for all three fields of the structure. The fields left blank in `info1` use the default values for those fields. The `info2` declaration is illegal since `"DirectoryName"` is longer than the initial string for that field, and the `"TESTFILE"` string cannot initialize a field defined with **DUP**.

Arrays of Structures and Unions

You can define an array of structures using the **DUP** operator (see Section 5.1.1, "Declaring and Referencing Arrays") or by creating a list of structures. For example, you can define an array of structure variables like this:

```
Item7   ITEMS    30 DUP ({,,{10}})
```

The `Item7` array defined here has 30 elements of type `ITEMS`, with the third field of each element (the union) initialized to `10`.

You can also list array elements as shown in this example:

```
Item8   ITEMS    {'Bolts', 126, 10},
                 {'Pliers',139, 10},
                 {'Saws',  414, 10}
```

Structure Redefinition

The assembler generates an error for a structure redefinition unless all of the following are the same:

- Field names
- Offsets of named fields
- Initialization lists
- Field alignment value

Additionally, all fields must be present and at the same offset.

LENGTHOF, SIZEOF, and TYPE for Structures

The size of a structure determined by **SIZEOF** is the offset of the last field, plus the size of the last field, plus any padding required for proper alignment (see Section 5.2.1 for information about alignment). This example, using the data declarations above, shows how to use the **LENGTHOF**, **SIZEOF**, and **TYPE** operators with structures:

```
INFO            STRUCT
  buffer        BYTE    100 DUP (?)
  crlf          BYTE    13, 10
  query         BYTE    'Filename: '
  endmark       BYTE    36
  drives        DISKDRIVES <0, 1, 1>
INFO            ENDS

info1   INFO    { , , 'Dir' }
lotsof  INFO    { , , 'file1', , {0,0,0} },
                { , , 'file2', , {0,0,1} },
                { , , 'file3', , {0,0,2} }
```

```
sinfo1  EQU     SIZEOF    info1  ; 116 = number of bytes in
                                 ;  initializers
linfo1  EQU     LENGTHOF  info1  ; 1 = number of items
tinfo1  EQU     TYPE      info1  ; 116 = same as size

slotsof EQU     SIZEOF    lotsof ; 116 * 3 = number of bytes in
                                 ;  initializers
llotsof EQU     LENGTHOF  lotsof ; 3 = number of items
tlotsof EQU     TYPE      lotsof ; 116 = same as size for structure
                                 ;  of type INFO
```

LENGTHOF, SIZEOF, and TYPE for Unions

The size of a union determined by **SIZEOF** is the size of the longest field plus any padding required. The length of a union variable determined by **LENGTHOF** equals the number of initializers defined inside angle brackets or curly braces. **TYPE** returns a value indicating the type of the longest field.

```
DWB     UNION
  d     DWORD   ?
  w     WORD    ?
  b     BYTE    ?
DWB     ENDS

num     DWB     {0FFFFh}
array   DWB     (100 / SIZEOF DWB) DUP ({0})

snum    EQU     SIZEOF   num      ; = 4
lnum    EQU     LENGTHOF num      ; = 1
tnum    EQU     TYPE     num      ; = 4
sarray  EQU     SIZEOF   array    ; = 100 (4*25)
larray  EQU     LENGTHOF array    ; = 25
tarray  EQU     TYPE     array    ; = 4
```

5.2.3 Referencing Structures, Unions, and Fields

Like other variables, structure variables can be accessed by name. You can access fields within structure variables with this syntax:

variable.field

In MASM 6.0, references to fields must always be fully qualified, with both the structure or union name and the dot operator preceding the field name. Also, in MASM 6.0, the dot operator can be used only with structure fields, not as an alternative to the plus operator; nor can the plus operator be used as an alternative to the dot operator.

This example shows several ways to reference the fields of a structure called `date`.

```
DATE    STRUCT                              ; Defines structure type
  month BYTE    ?
  day   BYTE    ?
  year  WORD    ?
DATE    ENDS

yesterday       DATE    {9, 30, 1987}       ; Declare structure
                                            ;  variable
        .
        .
        .
        mov     al, yesterday.day           ; Use structure variables
        mov     bx, OFFSET yesterday        ; Load structure address
        mov     al, (DATE PTR [bx]).month   ; Use as indirect operand
        mov     al, [bx].date.month         ; This is necessary if
                                            ;  month were already a
                                            ;  field in a different
                                            ;  structure
```

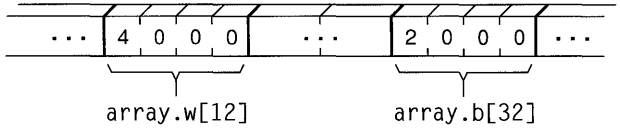
Under **OPTION M510** or **OPTION OLDSTRUCTS**, unique structure names do not need to be qualified. See Section 1.3.2 for information on the **OPTION** directive.

If the **NONUNIQUE** keyword appears in a structure definition, all fields of the structure must be fully qualified when referenced, even if the **OPTION OLDSTRUCTS** directive appears in the code. Also, in MASM 6.0, all references to a field must be qualified.

Even if the initialized union is the size of a **WORD** or **DWORD**, members of structures or unions are accessible only through the field's names.

In the following example, the two **MOV** statements show how you can access the elements of an array of structures.

```
WB      UNION
  w     WORD    ?
  b     BYTE    ?
WB      ENDS

array   WB      (100 / SIZEOF WB) DUP ({0})

        mov     array[12].w, 40
        mov     array[32].b,  2
```



The WB union cannot be used directly as a **WORD** variable. However, you can define a union containing both the structure and a **WORD** variable and access either field. (The next section discusses nested structures and unions.)

You can use unions to access the same data in more than one form. For example, one application of structures and unions is to simplify the task of reinitializing a far pointer. If you have a far pointer declared as

```
FPWORD  TYPEDEF FAR PTR WORD

        .DATA
BoxB    FPWORD ?
BoxA    FPWORD ?
BoxB2   uptr   < >
```

you must follow these steps to point BoxB to BoxA:

```
        mov     bx,  OFFSET BoxA
        mov     WORD PTR BoxB[2], ds
        mov     WORD PTR BoxB, bx
```

When you do this, you must remember whether the segment or the offset is stored first. However, if your program contains this union:

```
uptr      UNION
  dwptr   FPWORD   0
  STRUCT
    offs  WORD     0
    segm  WORD     0
  ENDS
uptr      ENDS
```

you can initialize a far pointer with these steps:

```
        mov     BoxB2.segm, ds
        mov     BoxB2.offs, bx
        lds     si, BoxB2.dwptr
```

This code moves the segment and the offset into the pointer and then moves the pointer into a register with the other field of the union. Although this technique does not reduce the code size, it avoids confusion about the order for loading the segment and offset.

5.2.4 Nested Structures and Unions

Structures and unions in MASM 6.0 can be nested in several ways. This section explains how to refer to the fields in a nested structure or union. The example below illustrates the four techniques for nesting and how to reference the fields. Note the syntax for nested structures. The discussion of these techniques follows the example.

```
ITEMS           STRUCT
  Inum          WORD    ?
  Iname         BYTE    'Item Name'
ITEMS           ENDS

INVENTORY       STRUCT
  UpDate        WORD    ?
  oldItem       ITEMS   { \
                        ?,
                        'AF8' \         ; Named variable of
                        }               ;  existing structure
                ITEMS   { ?, '94C' }    ; Unnamed variable of
                                        ;  existing type
  STRUCT ups                            ; Named nested structure
    source      WORD    ?
    shipmode    BYTE    ?
  ENDS
  STRUCT                                ; Unnamed nested structure
    f1          WORD    ?
    f2          WORD    ?
  ENDS
INVENTORY       ENDS

        .DATA

yearly  INVENTORY       { }

; Referencing each type of data in the yearly structure:

        mov     ax, yearly.oldItem.Inum
        mov     yearly.ups.shipmode, 'A'
        mov     yearly.Inum, 'C'
        mov     ax, yearly.f1
```

To nest structures and unions, you can use any of these techniques:

- The field of a structure or union can be a named variable of an existing structure or union type, as in the `oldItem` field. The field names in `oldItem` are not unique, so the full field names must be used when referencing those fields in the statement

```
mov     ax, yearly.oldItem.Inum
```

- To declare a named structure or union inside another structure or union, give the **STRUCT** or **UNION** keyword first and then define a label for it. Fields of the nested structure or union must always be qualified, as shown in this example:

```
mov     yearly.ups.shipmode, 'A'
```

- As shown in the `Items` field of `Inventory`, you can also use unnamed variables of existing structures or unions inside another structure or union. In this case you can reference its fields directly, as shown in this example:

```
mov     yearly.Inum, 'C'
mov     ax, yearly.f1
```

Offsets of nested structures are relative to the nested structure, not the root structure. In the example above, the offset of `yearly.ups.shipmode` is `(current address of yearly) + 8 + 2`. It is relative to the `ups` structure, not the `yearly` structure.

5.3 Records

Records are similar to structures, except that fields in records are bit strings. Each bit field in a record variable can be used separately in constant operands or expressions. The processor cannot access bits individually at run time, but it can access bit fields with instructions that manipulate bits.

Record fields are bits, not bytes or words.

Records are bytes, words, or doublewords in which the individual bits or groups of bits are considered fields. In general, the three steps for using record variables are the same as those for other complex data types:

1. Declare a record type.
2. Define one or more variables having the record type.
3. Reference record variables using shifts and masks.

Once defined, the record variable can be used as an operand in assembler statements.

This section explains the record declaration syntax and the use of the **MASK** and **WIDTH** operators. It also shows a few applications of record variables and constants.

5.3.1 Declaring Record Types

A record type creates a template for data with the sizes and, optionally, the initial values for bit fields in the record, but it does not allocate memory space for the record.

The **RECORD** directive declares a record type for an 8-bit, 16-bit, or 32-bit record that contains one or more bit fields. The maximum size is based on the expression word size. See **OPTION EXPR16** and **OPTION EXPR32** in Section 1.3.2. The syntax is

recordname **RECORD** *field* [[,*field*]]...

The *field* declares the name, width, and initial value for the field. The syntax for each *field* is:

fieldname:*width*[[=*expression*]]

Global labels, macro names, and record field names must all be unique, but record field names can have the same names as structure field names or global labels. *Width* is the number of bits in the field, and *expression* is a constant giving the initial (or default) value for the field. Record definitions can span more than one line if the continued lines end with commas.

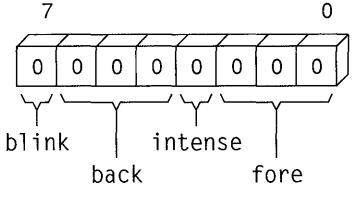
If *expression* is given, it declares the initial value for the field. The assembler generates an error message if an initial value is too large for the width of its field.

The assembler shifts bits in a record to the right if all bits are not used.

The first field in the declaration always goes into the most significant bits of the record. Subsequent fields are placed to the right in the succeeding bits. If the fields do not total exactly 8, 16, or 32 bits as appropriate, the entire record is shifted right, so the last bit of the last field is the lowest bit of the record. Unused bits in the high end of the record are initialized to 0.

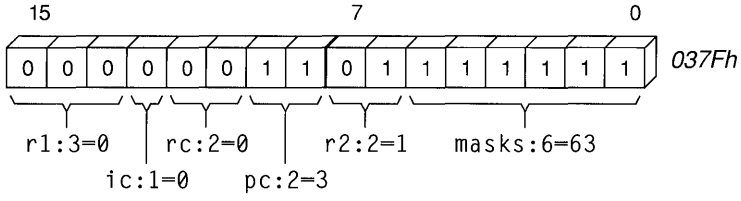
The following example creates a byte record type `color` having four fields: `blink`, `back`, `intense`, and `fore`. The contents of the record type are shown after the example. Since no initial values are given, all bits are set to 0. Note that this is only a template maintained by the assembler. No data is created.

```
COLOR   RECORD  blink:1, back:3, intense:1, fore:3
```



The next example creates a record type `cw` having six fields. Each record declared with this type occupies 16 bits of memory. Initial (default) values are given for each field. They can be used when data is declared for the record. The bit diagram after the example shows the contents of the record type.

```
CW      RECORD  r1:3=0, ic:1=0, rc:2=0, pc:2=3, r2:2=1, masks:6=63
```



5.3.2 Defining Record Variables

Once you have declared a record type, you can define record variables of that type. For each variable, memory is allocated to the object file in the format declared by the type. The syntax is

[[*name*]] *recordname* <[[*initializer* [[,*initializer*]]...]] >

[[*name*]] *recordname* {[[*initializer* [[,*initializer*]]...]] }

[[*name*]] *recordname constant* **DUP** ([[*initializer* [[,*initializer*]]...]])

The *recordname* is the name of a record type that was previously declared by using the **RECORD** directive.

A *fieldlist* for each field in the record can be a list of integers, character constants, or expressions that correspond to a value compatible with the size of the field. Curly braces or angle brackets are required even if no initial value is given.

If you use the **DUP** operator (see Section 5.1.1, "Declaring and Referencing Arrays") to initialize multiple record variables, only the angle brackets and initial values, if given, need to be enclosed in parentheses. For example, you can define an array of record variables with

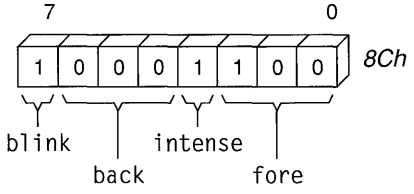
```
xmas    COLOR   50 DUP ( <1, 2, 0, 4> )
```

You do not have to initialize all fields in a record. If an initial value is blank, the assembler automatically stores the default initial value of the field. If there is no default value, the assembler clears each bit in the field.

The definition in the example below creates a variable named `warning` whose type is given by the record type `color`. The initial values of the fields in the

variable are set to the values given in the record definition. The initial values override any default record values, had any been given in the declaration.

```
COLOR   RECORD  blink:1,back:3,intense:1,fore:3 ; Record
                                               ;  declaration
warning COLOR   <1, 0, 1, 4>                   ; Record
                                               ;  definition
```



LENGTHOF, SIZEOF, and TYPE with Records

The **SIZEOF** and **TYPE** operators applied to a record name return the number of bytes used by the record. **SIZEOF** for a record variable returns the number of bytes used by the variable. You cannot use **LENGTHOF** with record types, but you can with the variables of that type. **LENGTHOF** returns the number of items in an initializer. The record can be used as an operand. The value of the operand is a bit mask of the defined record. This example illustrates these points.

```
; Record definition
; 9 bits stored in 2 bytes
RGBCOLOR        RECORD  red:3,  green:3,  blue:3

        mov     ax, RGBCOLOR            ; Equivalent to "mov ax,
                                        ;  01FFh"
;       mov     ax, LENGTHOF RGBCOLOR   ; Illegal since LENGTHOF can
                                        ;  apply only to data label
        mov     ax, SIZEOF   RGBCOLOR   ; Equivalent to "mov ax, 2"
        mov     ax, TYPE     RGBCOLOR   ; Equivalent to "mov ax, 2"

; Record instance
; 8 bits stored in 1 byte
RGBCOLOR2       RECORD red:3, green:3, blue:2
rgb             RGBCOLOR2 <1, 1, 1>     ; Initialize to 025h

        mov     ax, RGBCOLOR2           ; Equivalent to "mov ax,
                                        ;  00FFhh"
        mov     ax, LENGTHOF rgb        ; Equivalent to "mov ax, 1"
        mov     ax, SIZEOF   rgb        ; Equivalent to "mov ax, 1"
        mov     ax, TYPE     rgb        ; Equivalent to "mov ax, 1"
```

5.3.3 Record Operators

The **WIDTH** operator (which is used only with records) returns the width in bits of a record or record field. The **MASK** operator returns a bit mask for the bit positions occupied by the given record field. A bit in the mask contains a 1 if that bit corresponds to a bit field. The example below shows how to use **MASK** and **WIDTH**.

```
        .DATA
COLOR           RECORD  blink:1, back:3, intense:1, fore:3
message         COLOR   <1, 5, 1, 1>
wblink  EQU     WIDTH blink             ; "wblink"      = 1
wback   EQU     WIDTH back              ; "wback"       = 3
wintense EQU    WIDTH intense           ; "wintense"    = 1
wfore   EQU     WIDTH fore              ; "wfore"       = 3
wcolor  EQU     WIDTH color             ; "wcolor"      = 8
        .CODE
        .
        .
        .
        mov     ah, message             ; Load initial   0101 1001
        and     ah, NOT MASK back       ; Turn off   AND 1000 1111
                                        ; "back"         ---------
                                        ;                0000 1001
        or      ah, MASK blink          ; Turn on     OR 1000 0000
                                        ; "blink"        ---------
                                        ;                1000 1001
        xor     ah, MASK intense        ; Toggle     XOR 0000 1000
                                        ; "intense"      ---------
                                        ;                1000 0001
        .
        IF      (WIDTH color) GE 8      ; If color is 16 bit, load
        mov     ax, message             ;  into 16-bit register
        ELSE                            ; else
        mov     al, message             ;  load into low 8-bit register
        xor     ah, ah                  ;  and clear high 8-bits
        ENDIF
```

This example illustrates several ways in which record fields can be used as operands and in expressions.

```
; Rotate "back" of "cursor" without changing other values

mov     al, cursor       ; Load value from memory
mov     ah, al           ; Save a copy for work       1101 1001=ah/al
and     al, NOT MASK back; Mask out old bits     AND 1000 1111=mask
                         ;  to save old cursor       ---------
                         ;                          1000 1001=al
mov     cl, back         ; Load bit position
shr     ah, cl           ; Shift to right             0000 1101=ah
inc     ah               ; Increment                  0000 1110=ah

shl     ah, cl           ; Shift left again           1110 0000=ah
and     ah, MASK back    ; Mask off extra bits    AND 0111 0000=mask
                         ;  to get new cursor        ---------
                         ;                          0110 0000 ah
or      ah, al           ; Combine old and new     OR 1000 1001 al
                         ;                          ---------
mov     cursor, ah       ; Write back to memory       1110 1001 ah
```

Record variables are often used with the logical operators to perform logical operations on the bit fields of the record, as in the previous example using the **MASK** operator.

5.4 Related Topics in Online Help

In addition to information on all the instructions and directives mentioned in this chapter, information on the following topics can be found in online help, starting at the "MASM 6.0 Contents" screen:

Topic	Access
INS, OUTS	Choose "Processor Instructions" and then "System and I/O Access"
LABEL	Choose "Directives" and then "Code Labels"
RECORD, UNION, STRUCT, MASK, ORG, WIDTH, and **ALIGN**	Choose "Directives" and then choose "Complex Data Types"
SHRD, SHLD, BSF, and **BSR**	From "Processor Instructions," choose "Logical and Shifts"
BOUND	From "Processor Instructions," choose "Data Transfer"

Chapter 6

Using Floating-Point and Binary Coded Decimal Numbers

MASM requires different techniques for handling floating-point (real) numbers and binary coded decimal (BCD) numbers than for handling integers. You have two choices for working with real numbers—a math coprocessor or emulation routines.

Math coprocessors—the 8087, 80287, and 80387 chips—work with the main processor to handle real-number calculations. The 80486 processor performs floating-point operations directly. All information in this chapter pertaining to the 80387 coprocessor applies to the 80486 processor as well.

This chapter begins with a summary of the directives and formats of floating-point data; you need to use these to allocate memory storage and initialize variables before you can work with floating-point numbers.

The chapter then explains how to use a math coprocessor for floating-point operations. It covers these areas:

- The architecture of the registers
- The operands for the coprocessor instruction formats
- The coordination of coprocessor and main processor memory access
- The basic groups of coprocessor instructions—for loading and storing data, doing arithmetic calculations, and controlling program flow

The next main section describes emulation libraries. With the emulation routines provided with all Microsoft high-level languages, you can use coprocessor instructions as though your computer had a math coprocessor. However, some coprocessor instructions are not handled by emulation, as this section explains.

Finally, because math coprocessor and emulation routines can also operate on BCD numbers, this chapter discusses the instruction set for these numbers.

6.1 Using Floating-Point Numbers

Before using floating-point data in your program, you need to allocate the memory storage for the data. You can then initialize variables either as real numbers in decimal form or as encoded hexadecimals. The assembler stores allocated data in 10-byte IEEE format. This section looks at floating-point declarations and floating-point data formats.

6.1.1 Declaring Floating-Point Variables and Constants

You can allocate real constants using the **REAL4**, **REAL8**, and **REAL10** directives. The list below shows the size of the floating-point number each of these directives allocates.

Directive	Size
REAL4	Short (32-bit) real numbers
REAL8	Long (64-bit) real numbers
REAL10	10-byte (80-bit) real numbers and BCD numbers

The possible ranges for floating-point variables are given in Table 6.1.

Table 6.1 Ranges of Floating-Point Variables

Data Type	Bits	Significant Digits	Approximate Range
Short real	32	6–7	$\pm 1.18 \times 10^{-38}$ to $\pm 3.40 \times 10^{38}$
Long real	64	15–16	$\pm 2.23 \times 10^{-308}$ to $\pm 1.79 \times 10^{308}$
10-byte real	80	19	$\pm 3.37 \times 10^{-4932}$ to $\pm 1.18 \times 10^{4932}$

With previous versions of MASM, the **DD**, **DQ**, and **DT** directives could be used to allocate real constants. These directives are still supported by MASM 6.0, but this means that the variables are integers rather than floating-point values. Although this makes no difference in the assembly code, CodeView displays the values incorrectly.

There are two forms for specifying floating-point numbers.

You can specify floating-point constants either as decimal constants or as encoded hexadecimal constants. You can express decimal real-number constants in the form

[[+ | –]] *integer*.[[*fraction*]][[**E**]][[[+ | –]]*exponent*]]

For example, the numbers `2.523E1` and `-3.6E-2` are written in the correct decimal format. These numbers can be used as initializers for real-number variables.

Digits of real numbers are always evaluated as base 10. During assembly, the assembler converts real-number constants given in decimal format to a binary format. The sign, exponent, and mantissa of the real number are encoded as bit fields within the number.

You can also specify the encoded format directly with hexadecimal digits (0–9 plus A–F). The number must begin with a decimal digit (0–9) and a leading zero if necessary, and end with the real-number designator (R). It cannot be signed.

For example, the hexadecimal number `3F800000r` can be used as an initializer for a doubleword-sized variable.

The maximum range of exponent values and the number of digits required in the hexadecimal number depend on the directive. The number of digits for encoded numbers used with **REAL4**, **REAL8**, and **REAL10** must be 8, 16, and 20 digits, respectively. If the number has a leading zero, the number must be 9, 17, or 21 digits.

Examples of decimal constant and hexadecimal specifications are shown here:

```
; Real numbers
short   REAL4    25.23              ; IEEE format
double  REAL8    2.523E1            ; IEEE format
tenbyte REAL10   2523.0E-2          ; 10-byte real format

; Encoded as hexadecimals
ieeeshort       REAL4    3F800000r             ; 1.0 as IEEE short
ieeedouble      REAL8    3FF0000000000000r     ; 1.0 as IEEE long
temporary       REAL10   3FFF8000000000000000r ; 1.0 as 10-byte
                                                ;  real
```

Section 6.1.2, "Storing Numbers in Floating-Point Format," explains the IEEE formats—the way the assembler actually stores the data.

Pascal or C programmers may prefer to create language-specific **TYPEDEF** declarations, as illustrated in this example:

```
; C-language specific
float           TYPEDEF REAL4
double          TYPEDEF REAL8
long_double     TYPEDEF REAL10
; Pascal-language specific
SINGLE          TYPEDEF REAL4
DOUBLE          TYPEDEF REAL8
EXTENDED        TYPEDEF REAL10
```

For applications of **TYPEDEF** other than aliasing, see Section 3.3.1, "Defining Pointer Types with TYPEDEF."

6.1.2 Storing Numbers in Floating-Point Format

The assembler stores real numbers in the IEEE format.

The assembler stores the floating-point variables in the IEEE format. MASM 6.0 does not support **.MSFLOAT** and Microsoft binary format, which are available in previous versions.

Figure 6.1 illustrates the IEEE format for encoding short (four-byte), long (eight-byte), and 10-byte real numbers. Although this figure places the most-significant bit first for illustration, low bytes actually appear first in memory.

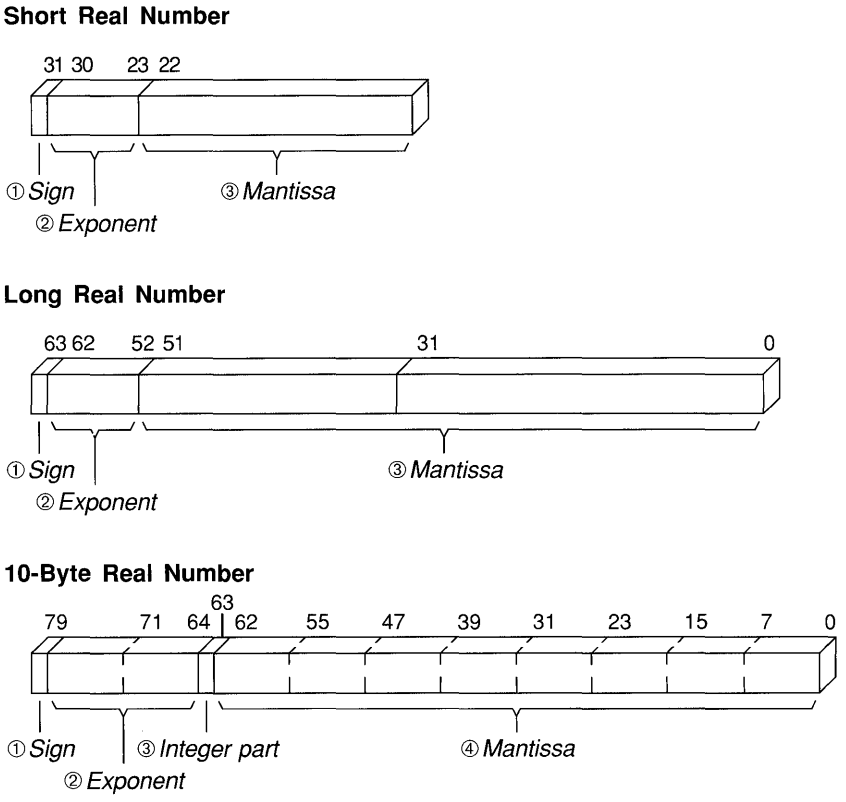


Figure 6.1 Encoding for Real Numbers in IEEE Format

This is how the parts of a real number are stored in the IEEE format:

1. Sign bit (0 for positive or 1 for negative) in the upper bit of the first byte.
2. Exponent in the next bits in sequence (8 bits for a short real number, 11 bits for a long real number, and 15 bits for a 10-byte real number).

3. Mantissa in the remaining bits. The first bit is always assumed to be 1. The length is 23 bits for short real numbers, 52 bits for long real numbers, and 63 bits for 10-byte reals.

The exponent field represents a multiplier 2^n. To accommodate negative exponents (such as 2^{-6}), the value in the exponent field is biased; that is, the actual exponent is determined by subtracting the appropriate bias value from the value in the exponent field. For example, the bias for short reals is 127. If the value in the exponent field is 130, the exponent represents a value of $2^{130-127}$, or 2^3. The bias for long reals is 1,023. The bias for 10-byte reals is 16,383.

Notice that the 10-byte real format stores the integer part of the mantissa. This differs from the 4-byte and 8-byte formats, in which the integer part is implicit.

Once you have declared floating-point data for your program, you can use coprocessor or emulator instructions to access the data. The next section focuses on the coprocessor architecture, instructions, and operands required for floating-point operations.

6.2 Using a Math Coprocessor

When used with real numbers, packed BCD numbers, or long integers, coprocessors (the 8087, 80287, 80387, and 80486) calculate many times faster than the 8086-based processors. The coprocessor handles data with its own registers. The organization of these registers reflects four possible formats for using operands (as explained in Section 6.2.2, "Instruction and Operand Formats").

This section also describes how the coprocessor performs various tasks: transferring data to and from the coprocessor, coordinating processor and coprocessor operations, and controlling program flow.

6.2.1 Coprocessor Architecture

The coprocessor accesses memory as the CPU does, but it has its own data and control registers—eight data registers organized as a stack and seven control registers similar to the 8086 flag registers. The coprocessor's instruction set provides direct access to these registers.

The eight coprocessor data registers form a stack.

The eight 80-bit data registers of the 8087-based coprocessors are organized as a stack although they need not be used as a stack. As data items are pushed into the top register, previous data items move into higher-numbered registers, which are lower on the stack. Register 0 is the top of the stack; register 7 is the bottom. The syntax for specifying registers is shown below:

ST [[(*number*)]]

The *number* must be a digit between 0 and 7 or a constant expression that evaluates to a number from 0 to 7. **ST** is another way to refer to **ST(0)**.

All coprocessor data is stored in registers in the 10-byte real format. The registers and the register format are shown in Figure 6.2.

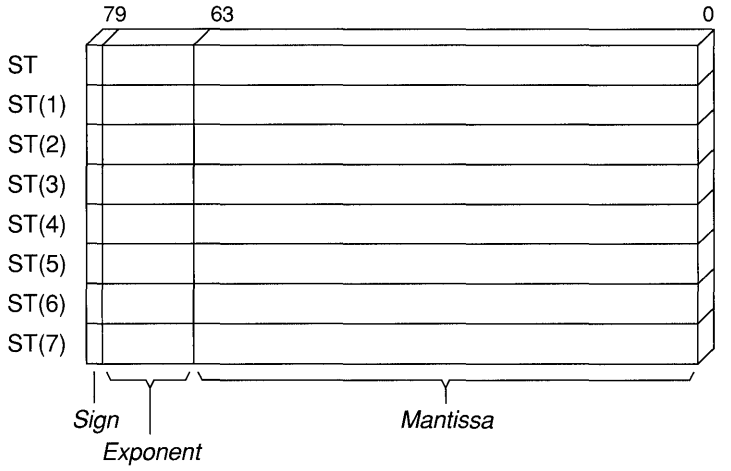


Figure 6.2 Coprocessor Data Registers

Internally, all calculations are done on numbers of the same type. Since 10-byte real numbers have the greatest precision, lower-precision numbers are guaranteed not to lose precision as a result of calculations. The instructions that transfer values between the main memory and the coprocessor automatically convert numbers to and from the 10-byte real format.

6.2.2 Instruction and Operand Formats

Because of the stack organization of registers, you can consider registers either as elements on a stack or as registers much like 8086-family registers. Table 6.2 lists the four main groups of coprocessor instructions and the general syntax for each. The names given to the instruction format reflect the way the instruction uses the coprocessor registers. The instruction operands are placed in the coprocessor data registers before the instruction executes.

Table 6.2 Coprocessor Operand Formats

Instruction Format	Syntax	Implied Operands	Example
Classical stack	F*action*	ST, ST(1)	`fadd`
Memory	F*action memory*	ST	`fadd memloc`
Register	F*action* **ST(***num***), ST** F*action* **ST, ST(***num***)**	—	`fadd st(5), st` `fadd st, st(3)`
Register pop	F*action*P **ST(***num***), ST**	—	`faddp st(4), st`

All coprocessor instructions begin with F.

You can easily recognize coprocessor instructions because, unlike all 8086-family instruction mnemonics, they start with the letter **F**. Coprocessor instructions can never have immediate operands and, with the exception of the **FSTSW** instruction, they cannot have processor registers as operands.

6.2.2.1 Classical-Stack Format

Instructions in the classical-stack format treat the coprocessor registers like items on a stack—thus its name. Items are pushed onto or popped off the top elements of the stack. Since only the top item can be accessed on a traditional stack, there is no need to specify operands. The first (top) register (and the second if the instruction needs two operands) is always assumed.

In coprocessor arithmetic operations, the top of the stack (ST) is the source operand and the second register [ST(1)] is the destination. The result of the operation goes into the destination operand, and the source is popped off the stack. The result is left at the top of the stack.

Instructions that load constants are one example of instructions that require the classical-stack format. In this case, the constant created by the instruction is the implied source, and the top of the stack is the destination.

This example illustrates the classical-stack format, and Figure 6.3 shows the status of the register stack after each instruction:

```
fld1                ; Push 1 into first position
fldpi               ; Push pi into first position
fadd                ; Add pi and 1 and pop
```

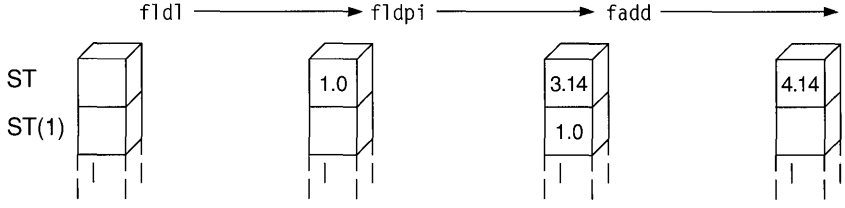


Figure 6.3 Status of the Register Stack

6.2.2.2 Memory Format

Instructions using the memory format, such as data transfer instructions, also treat coprocessor registers like items on a stack. However, with this format, items are pushed from memory onto the top element of the stack or popped from the top element to memory. You must specify the memory operand.

Some coprocessor instructions operate on integers or BCDs.

Some instructions that use the memory format specify how a memory operand is to be interpreted—as an integer (**I**) or as a binary coded decimal (**B**). The letter **I** or **B** follows the initial **F** in the syntax. For example, **FILD** interprets its operand as an integer and **FBLD** interprets its operand as a BCD number. If the instruction name does not include a type letter, the instruction works on real numbers.

You can also use memory operands in calculation instructions that operate on two values (see Section 6.2.4, "Using Coprocessor Instructions"). The memory operand is always the source. The stack top (ST) is always the implied destination. The result of the operation replaces the destination without changing its stack position, as shown in this example and Figure 6.4:

```
            .DATA
m1          REAL4  1.0
m2          REAL4  2.0
            .CODE
            .
            .
            .
            fld    m1       ; Push m1 into first position
            fld    m2       ; Push m2 into first position
            fadd   m1       ; Add m2 to first position
            fstp   m1       ; Pop first position into m1
            fst    m2       ; Copy first position to m2
```

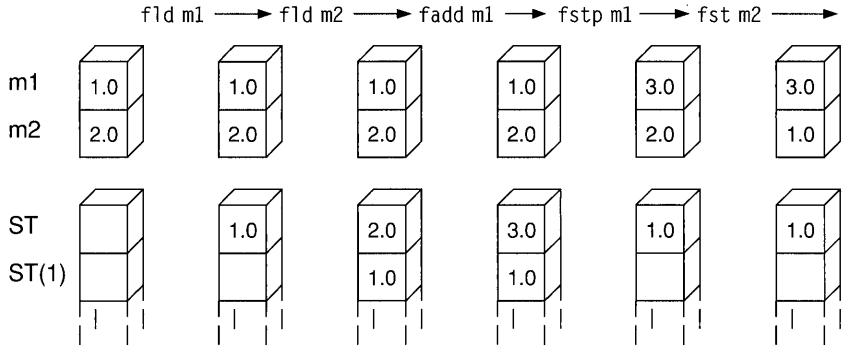


Figure 6.4 Status of the Register Stack and Memory Locations

6.2.2.3 Register Format

Instructions using the register format treat coprocessor registers as registers rather than as stack elements. Instructions that use this format require two register operands; one of them must be the stack top (ST).

In the register format, specify all operands by name. The first operand is the destination; its value is replaced with the result of the operation. The second operand is the source; it is not affected by the operation. The stack position of the operands does not change.

The only instructions using the register operand format are the **FXCH** instruction and the arithmetic instructions that do calculations on two values. With the **FXCH** instruction, the stack top is implied and need not be specified, as shown in this example and Figure 6.5:

```
fadd    st(1), st   ; Add second position to first -
                    ;  result goes in second position
fadd    st, st(2)   ; Add first position to third -
                    ;  result goes in first position
fxch    st(1)       ; Exchange first and second positions
```

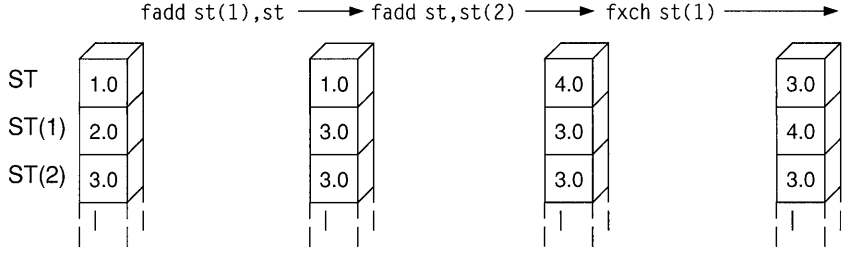


Figure 6.5 Status of the Previously Initialized Register Stack

6.2.2.4 Register-Pop Format

The register-pop format treats coprocessor registers as a modified stack. The source register must always be the stack top. Specify the destination with the register's name.

Instructions with this format place the result of the operation into the destination operand, and the stack top pops off the stack. The effect is that both values being operated on are lost and the result of the operation is saved in the specified destination register. The register-pop format is used only for instructions that do calculations on two values, as in this example and Figure 6.6:

```
faddp   st(2), st ; Add first and third positions and pop -
                  ;  first position destroyed;
                  ;  third moves to second and holds result
```

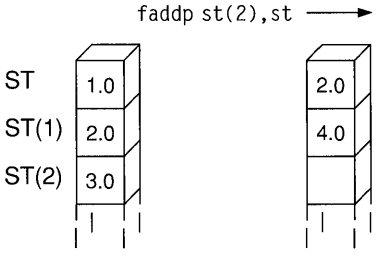


Figure 6.6 Status of the Already Initialized Register Stack

6.2.3 Coordinating Memory Access

The math coprocessor works simultaneously with the main processor. However, since the coprocessor cannot handle device input or output, data originates in the main processor.

The processor and coprocessor exchange data through memory.

The main processor and the coprocessor have their own registers, which are completely separate and inaccessible to each other. They usually exchange data through memory, since memory is available to both.

When using the coprocessor, follow these three steps:

1. Load data from memory to coprocessor registers.
2. Process the data.
3. Store the data from coprocessor registers back to memory.

Step 2, processing the data, can occur while the main processor is handling other tasks. Steps 1 and 3 must be coordinated with the main processor so that the processor and coprocessor do not try to access the same memory at the same time; otherwise, problems of coordinating memory access can occur. Since the processor and coprocessor work independently, they may not finish working on memory in the order in which you give instructions. Two potential timing conflicts can occur; they are handled in different ways.

One timing conflict results if a coprocessor instruction follows a processor instruction. The processor may have to wait until the coprocessor finishes if the next processor instruction requires the result of the coprocessor's calculation. You do not have to write your code to avoid this conflict, however. The assembler coordinates this timing automatically for the 8088 and 8086 processors, and the processor coordinates it automatically on the 80186–80486 processors. This is the first case shown in the example later in this section.

Another conflict results if a processor instruction that accesses memory follows a coprocessor instruction that accesses the same memory. The processor can try to load a variable that is still being used by the coprocessor. You need careful synchronization to control the timing, and this synchronization is not automatic on the 8087 coprocessor. For code to run correctly on the 8087, you must include the **WAIT** or **FWAIT** instruction (they are mnemonics for the same instruction) to ensure that the coprocessor finishes before the processor begins, as shown in the second example. In this situation, the processor does not generate the **FWAIT** instruction automatically.

```
; Processor instruction first - No wait needed
        mov     WORD PTR mem32[0], ax   ; Load memory
        mov     WORD PTR mem32[2], dx
        fild    mem32                   ; Load to register

; Coprocessor instruction first - Wait needed (for 8087)
        fist    mem32                   ; Store to memory
        fwait                           ; Wait until coprocessor
                                        ;  is done
        mov     ax, WORD PTR mem32[0]   ; Move to register
        mov     dx, WORD PTR mem32[2]
```

When generating code for the 8087 coprocessor, the assembler automatically inserts a **WAIT** instruction before the coprocessor instruction. However, if you use the **.286** or **.386** directive, the compiler assumes that the coprocessor instructions are for the 80287 or 80387 and does not insert the **WAIT** instruction.

If your code does not need to run on an 8086 or 8088 processor, you can make your programs shorter and more efficient by using the **.286** or **.386** directive.

6.2.4 Using Coprocessor Instructions

The 8087 family of coprocessors has separate instructions for each of the following operations:

- Loading and storing data
- Doing arithmetic calculations
- Controlling program flow

The following sections explain the available instructions and show how to use them for each of the operations listed above. See Section 6.2.2, "Instruction and Operand Formats," for general syntax information.

6.2.4.1 Loading and Storing Data

Data-transfer instructions transfer data between main memory and the coprocessor registers or between different coprocessor registers. Two principles govern data transfers:

- The choice of instruction determines whether a value in memory is considered an integer, a BCD number, or a real number. The value is always considered a 10-byte real number once it is transferred to the coprocessor.
- The size of the operand determines the size of a value in memory. Values in the coprocessor always take up 10 bytes.

Load commands transfer data, and store commands remove data.

You can transfer data to stack registers using load commands. These commands push data onto the stack from memory or from coprocessor registers. Store commands remove data. Some store commands pop data off the register stack into memory or coprocessor registers; others simply copy the data without changing it on the stack.

If you use constants as operands, you cannot load them directly into coprocessor registers. You must allocate memory and initialize a variable to a constant value. That variable can then be loaded by using one of the load instructions listed below.

A few special instructions are provided for loading certain constants. You can load 0, 1, pi, and several common logarithmic values directly. Using these instructions is faster and often more precise than loading the values from initialized variables.

All instructions that load constants have the stack top as the implied destination operand. The constant to be loaded is the implied source operand.

The coprocessor data area, or parts of it, can also be moved to memory and later loaded back. You may want to do this to save the current state of the coprocessor before executing a procedure. After the procedure ends, restore the previous status. Saving coprocessor data is also useful when you want to modify coprocessor behavior by writing certain data to main memory, operating on the data with 8086-family instructions, and then loading it back to the coprocessor data area.

You can use the following instructions for transferring numbers to and from registers:

Instruction(s)	Description
FLD, FST, FSTP	Loads and stores real numbers
FILD, FIST, FISTP	Loads and stores binary integers
FBLD	Loads BCD
FBSTP	Stores BCD
FXCH	Exchanges register values
FLDZ	Pushes 0 into ST
FLD1	Pushes 1 into ST
FLDPI	Pushes the value of pi into ST
FLDCW *mem2byte*	Loads the control word into the coprocessor
F[[N]]STCW *mem2byte*	Stores the control word in memory
FLDENV *mem14byte*	Loads environment from memory
F[[N]]STENV *mem14byte*	Stores environment in memory
FRSTOR *mem94byte*	Restores state from memory
F[[N]]SAVE *mem94byte*	Saves state in memory
FLDL2E	Pushes the value of $\log_2 e$ into ST
FLDL2T	Pushes $\log_2 10$ into ST
FLDLG2	Pushes $\log_{10} 2$ into ST
FLDLN2	Pushes $\log_e 2$ into ST

The following example and Figure 6.7 illustrate some of these instructions:

```
        .DATA
m1      REAL4   1.0
m2      REAL4   2.0
        .CODE
        fld     m1      ; Push m1 into first item
        fld     st(2)   ; Push third item into first
        fst     m2      ; Copy first item to m2
        fxch    st(2)   ; Exchange first and third items
        fstp    m1      ; Pop first item into m1
```

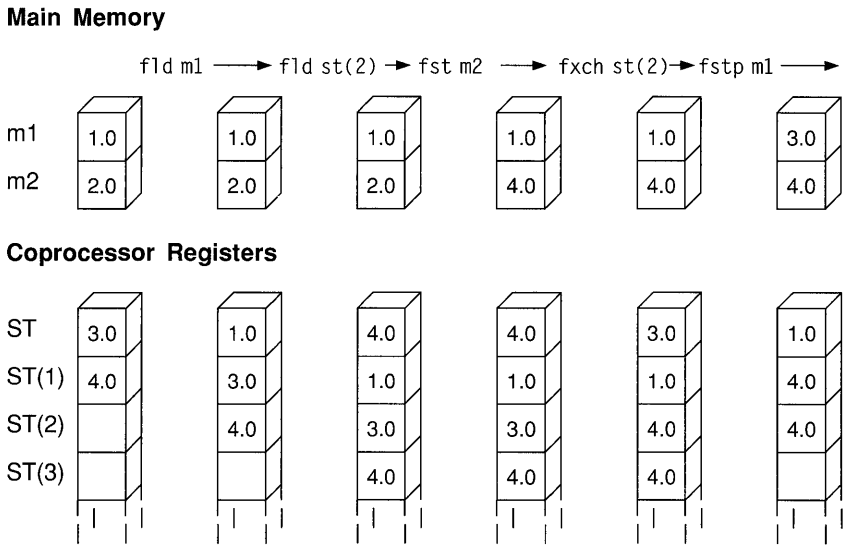


Figure 6.7 Status of the Register Stack: Main Memory and Coprocessor

6.2.4.2 Doing Arithmetic Calculations

Most of the coprocessor instructions for doing arithmetic operations have several forms, depending on the operand used. You do not need to specify the operand type in the instruction if both operands are stack registers, since register values are always 10-byte real numbers. The arithmetic instructions are listed below. In most cases, the result replaces the destination register.

Instruction	Description
FADD	Adds the source and destination
FSUB	Subtracts the source from the destination
FSUBR	Subtracts the destination from the source

Instruction	Description
FMUL	Multiplies the source and the destination
FDIV	Divides the destination by the source
FDIVR	Divides the source by the destination
FABS	Sets the sign of ST to positive
FCHS	Reverses the sign of ST
FRNDINT	Rounds ST to an integer
FSQRT	Replaces the contents of ST with its square root
FSCALE	Multiplies the stack-top value by 2 to the power contained in ST(1)
FPREM	Calculates the remainder of ST divided by ST(1)

80387 Only

Instruction	Description
FSIN	Calculates the sine of the value in ST
FCOS	Calculates the cosine of the value in ST
FSINCOS	Calculates the sine and cosine of the value in ST
FPREM1	Calculates the partial remainder by performing modulo division on the top two stack registers
FXTRACT	Breaks a number down into its exponent and mantissa and pushes the mantissa onto the register stack
F2XM1	Calculates 2^X-1
FYL2X	Calculates $Y * \log_2 X$
FYL2XP1	Calculates $Y * \log_2 (X+1)$
FPTAN	Calculates the tangent of the value in ST
FPATAN	Calculates the arctangent of the ratio Y/X
F[[N]]INIT	Resets the coprocessor and restores all the default conditions in the control and status words
F[[N]]CLEX	Clears all exception flags and the busy flag of the status word
FINCSTP	Adds 1 to the stack pointer in the status word
FDECSTP	Subtracts 1 from the stack pointer in the status word
FFREE	Marks the specified register as empty

The following example illustrating several arithmetic instructions solves quadratic equations. It does no error checking and fails for some values because it attempts to find the square root of a negative number. You could revise the code using the **FTST** (Test for Zero) instruction to check for a negative number or 0 before the square root is calculated. If $b^2 - 4ac$ is negative or 0, the code can jump to routines that handle these two special cases.

```
        .DATA
a       REAL4   3.0
b       REAL4   7.0
cc      REAL4   2.0
posx    REAL4   0.0
negx    REAL4   0.0

        .CODE
        .
        .
        .
; Solve quadratic equation - no error checking
; The formula is: -b +/- squareroot(b^2 - 4ac) / (2a)
        fld1                ; Get constants 2 and 4
        fadd    st,st       ; 2 at bottom
        fld     st          ; Copy it
        fmul    a           ; = 2a

        fmul    st(1),st    ; = 4a
        fxch                ; Exchange
        fmul    cc          ; = 4ac

        fld     b           ; Load b
        fmul    st,st       ; = b^2
        fsubr               ; = b^2 - 4ac
                            ; Negative value here produces error
        fsqrt               ; = square root(b^2 - 4ac)
        fld     b           ; Load b
        fchs                ; Make it negative
        fxch                ; Exchange

        fld     st          ; Copy square root
        fadd    st,st(2)    ; Plus version = -b + root(b^2 - 4ac)
        fxch                ; Exchange
        fsubp   st(2),st    ; Minus version = -b - root(b^2 - 4ac)

        fdiv    st,st(2)    ; Divide plus version
        fstp    posx        ; Store it
        fdivr               ; Divide minus version
        fstp    negx        ; Store it
```

The examples in online help contain an enhanced version of this procedure.

6.2.4.3 Controlling Program Flow

The math coprocessors have several instructions that set control flags in the status word. The 8087-family control flags can be used with conditional jumps to direct program flow in the same way that 8086-family flags are used. Since the coprocessor does not have jump instructions, you must transfer the status word to memory so that the flags can be used by 8086-family instructions.

An easy way to use the status word with conditional jumps is to move its upper byte into the lower byte of the processor flags, as shown in this example:

```
fstsw   mem16       ; Store status word in memory
fwait               ; Make sure coprocessor is done
mov     ax, mem16   ; Move to AX
sahf                ; Store upper word in flags
```

The **SAHF** (Store AH into Flags) instruction in the example above transfers AH into the low bits of the flags register.

You can save several steps by loading the status word directly to AX on the 80287 with the **FSTSW** and **FNSTSW** instructions. This is the only case in which data can be transferred directly between processor and coprocessor registers, as shown in this example:

```
fstsw   ax
```

The coprocessor control flags and their relationship to the status word are described in Section 6.2.4.4, "Control Registers."

The 8087-family coprocessors provide several instructions for comparing operands and testing control flags. All these instructions compare the stack top (ST) to a source operand, which may either be specified or implied as ST(1).

The compare instructions affect the C3, C2, and C0 control flags, but not the C1 flag. Table 6.3 shows the flags set for each possible result of a comparison or test.

Table 6.3 Control-Flag Settings after Comparison or Test

After FCOM	After FTEST	C3	C2	C0
ST > *source*	ST is positive	0	0	0
ST < *source*	ST is negative	0	0	1
ST = *source*	ST is 0	1	0	0
Not comparable	ST is NAN or projective infinity	1	1	1

Variations on the compare instructions allow you to pop the stack once or twice and to compare integers and zero. For each instruction, the stack top is always the implied destination operand. If you do not give an operand, ST(1) is the

implied source. With some compare instructions, you can specify the source as a memory or register operand.

All instructions summarized in the following list have implied operands: either ST as a single-destination operand or ST as the destination and ST(1) as the source. These are the instructions for comparing and testing flags.

Some instructions have a wait version and a no-wait version. The no-wait versions have **N** as the second letter.

Instruction	Description
FCOM	Compares the stack top to the source. The source and destination are unaffected by the comparison.
FTST	Compares ST to 0.
FCOMP	Compares the stack top to the source and then pops the stack.
FUCOM, FUCOMP, FUCOMPP	Compare the source to ST and set the condition codes of the status word according to the result (80386/486 only).
F[[N]]STSW *mem2byte*	Stores the status word in memory.
FXAM	Sets the value of the control flags based on the type of the number in ST.
FPREM	Finds a correct remainder for large operands. It uses the C2 flag to indicate whether the remainder returned is partial (C2 is set) or complete (C2 is clear). (If the bit is set, the operation should be repeated. It also returns the least-significant three bits of the quotient in C0, C3, and C1.)
FNOP	Copies the stack top onto itself, thus padding the executable file and taking up processing time without having any effect on registers or memory.
FDISI, FNDISI, FENI, FNENI	Enables or disables interrupts (8087 only).
FSETPM	Sets protected mode. Requires a **.286P** or **.386P** directive (80287, 80387, and 80486 only).

The following example illustrates some of these instructions. Notice how conditional blocks are used to enhance 80287 code.

```
        .DATA
down    REAL4   10.35       ; Sides of a rectangle
across  REAL4   13.07
diamtr  REAL4   12.93       ; Diameter of a circle
status  WORD    ?
P287    EQU     (@Cpu AND 00111y)
        .CODE
        .
        .
        .
; Get area of rectangle
        fld     across      ; Load one side
        fmul    down        ; Multiply by the other

; Get area of circle:  Area = PI * (D/2)2
        fld1                ; Load one and
        fadd    st, st      ;  double it to get constant 2
        fdivr   diamtr      ; Divide diameter to get radius
        fmul    st, st      ; Square radius
        fldpi               ; Load pi
        fmul                ; Multiply it

; Compare area of circle and rectangle
        fcompp              ; Compare and throw both away
        IF      p287
        fstsw   ax          ; (For 287+, skip memory)
        ELSE
        fnstsw  status      ; Load from coprocessor to memory
        mov     ax, status  ; Transfer memory to register
        ENDIF
        sahf                ; Transfer AH to flags register
        jp      nocomp      ; If parity set, can't compare
        jz      same        ; If zero set, they're the same
        jc      rectangle   ; If carry set, rectangle is bigger
        jmp     circle      ;  else circle is bigger

nocomp:                     ; Error handler
        .
        .
        .
same:                       ; Both equal
        .
        .
        .
rectangle:                  ; Rectangle bigger
        .
        .
        .
circle:                     ; Circle bigger
```

Additional instructions for the 80387/486 are **FLDENVD** and **FLDENVW** for loading the environment; **FNSTENVD**, **FNSTENVW**, **FSTENVD**, and **FSTENVW** for storing the environment state; **FNSAVED**, **FNSAVEW**, **FSAVED**, and

FSAVEW for saving the coprocessor state; and **FRSTORD** and **FRSTORW** for restoring the coprocessor state.

The size of the code segment, not the operand size, determines the number of bytes loaded or stored with these instructions. The instructions ending with **W** store the 16-bit form of the control register data, and the instructions ending with **D** store the 32-bit form. For example, in 16-bit mode **FSAVEW** saves the 16-bit control register data. If you need to store the 32-bit form of the control register data, use **FSAVED**.

6.2.4.4 Control Registers

Some of the flags of the seven 16-bit control registers control coprocessor operations, while others maintain the current status of the coprocessor. In this sense, they are much like the 8086-family flags registers (see Figure 6.8).

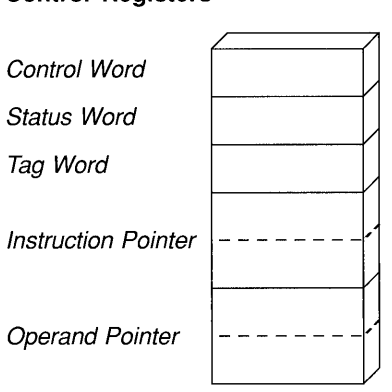


Figure 6.8 Coprocessor Control Registers

Of the control registers, only the status word register is commonly used (the others are used mostly by systems programmers). The format of the status word register is shown in Figure 6.9, which shows how the coprocessor control flags align with the processor flags. C3 overwrites the zero flag, C2 overwrites the parity flag, and C0 overwrites the carry flag. C1 overwrites an undefined bit, so it cannot be used directly with conditional jumps, although you can use the **TEST** instruction to check C1 in memory or in a register. The status word register also overwrites the sign and auxiliary-carry flags, so you cannot count on their being unchanged after the operation.

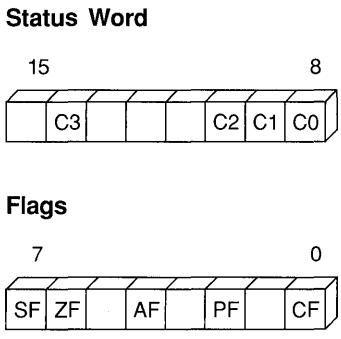


Figure 6.9 Coprocessor and Processor Control Flags

6.3 Using Emulator Libraries

If you do not have a math coprocessor or an 80486 processor, you can do most floating-point operations by writing assembly-language procedures and accessing the emulator from a high-level language. All Microsoft high-level languages come with the emulator library.

However, you cannot use a Microsoft emulator library with stand-alone assembler programs, since the library depends on the high-level-language start-up code.

With emulator libraries, you can use most floating-point instructions.

To use the emulator, first write the procedure using coprocessor instructions. Then assemble it using the /FPi option of your compiler. Finally, link it with your high-level-language modules. In MASM 6.0 you can enter options in the Programmer's WorkBench (PWB) environment, or you can use the **OPTION EMULATOR** in your source code.

In emulation mode, the assembler generates instructions for the linker that the Microsoft emulator can use. The form of the **OPTION** directive in the example below tells the assembler to use emulation mode. This option (introduced in Section 1.3.2) can be defined only once in a module.

```
OPTION EMULATOR
```

Emulator libraries do not allow for all of the coprocessor instructions. The following floating-point instructions are not emulated:

FCOS	**FRSTOR16**	**FSETPM**	**FUCOMPP**
FDECSTP	**FRSTOR32**	**FSIN**	**FXTRACT**
FINCSTP	**FSAVE**	**FSINCOS**	
FPREM1	**FSAVE16**	**FUCOM**	
FRSTOR	**FSAVE32**	**FUCOMP**	

The set of emulated instructions is different under OS/2 2.*x*. If you use a coprocessor instruction that is not emulated, your program generates a run-time error when it tries to execute the unemulated instruction.

See Chapter 20, "Mixed-Language Programming," for information about writing assembly-language procedures for high-level languages.

6.4 Using Binary Coded Decimal Numbers

Binary coded decimal (BCD) numbers allow calculations on large numbers without rounding errors. The 8087-family coprocessors can do fast calculations with packed BCD numbers. See Section 6.4.2.2 for details. The 8086-family processors can also do some calculations with packed BCD numbers, but the process is slower and more complicated. See Section 6.4.2 for details.

This section explains how to define BCD numbers and then how to use them in calculations.

6.4.1 Defining BCD Constants and Variables

Unpacked BCD numbers are made up of bytes containing a single decimal digit in the lower four bits of each byte. Packed BCD numbers are made up of bytes containing two decimal digits: one in the upper four bits and one in the lower four bits. The leftmost digit holds the sign (0 for positive, 1 for negative).

Packed BCD numbers are encoded in the 8087 coprocessor's packed BCD format. They can be up to 18 digits long, packed two digits per byte. The assembler zero-pads BCDs initialized with fewer than 18 digits. Digit 20 is the sign bit, and digit 19 is reserved.

The TBYTE directive allocates packed BCD constants.

When you define an integer constant with the **TBYTE** directive and the current radix is decimal (`t`), the assembler interprets the number as a packed BCD number.

The syntax for specifying packed BCDs is exactly the same as for other integers.

```
pos1    TBYTE   1234567890  ; Encoded as 00000000001234567890h
neg1    TBYTE   -1234567890 ; Encoded as 80000000001234567890h
```

Unpacked BCD numbers are stored one digit to a byte, with the value in the lower four bits. They can be defined using the **BYTE** directive. For example, an unpacked BCD number could be defined and initialized as shown below:

```
unpackedr       BYTE    1,5,8,2,5,2,9   ; Initialized to 9,252,851
unpackedf       BYTE    9,2,5,2,8,5,1   ; Initialized to 9,252,851
```

Least-significant digits can come either first or last, depending on how you write the calculation routines that handle the numbers.

6.4.2 Calculating with BCDs

When you use the processor to calculate with BCDs, the result is not correct unless you use the ASCII-adjust instructions to convert the result into the valid BCD integer.

6.4.2.1 Unpacked BCD Numbers

Instructions for unpacked BCDs allow accurate BCD calculations.

To do processor arithmetic on unpacked BCD numbers, you must do the eight-bit arithmetic calculations on each digit separately and assign the result to the AL register. After each operation, use the corresponding BCD instruction to adjust the result. The ASCII-adjust instructions do not take an operand. They always work on the value in the AL register.

When a calculation using two one-digit values produces a two-digit result, the **AAA**, **AAS**, **AAM**, and **AAD** instructions put the first digit in AL and the second in AH. If the digit in AL needs to carry to or borrow from the digit in AH, the instructions set the carry and auxiliary carry flags.

These instructions get their names from Intel mnemonics that use the term "ASCII" to refer to unpacked BCD numbers and "decimal" to refer to packed BCD numbers. The four ASCII-adjust instructions for unpacked BCDs are described below:

Instruction	Description
AAA	Adjusts after an addition operation.
AAS	Adjusts after a subtraction operation.
AAM	Adjusts after a multiplication operation. Always use with **MUL**, not with **IMUL**.
AAD	Adjusts before a division operation. Unlike other BCD instructions, **AAD** converts a BCD value to a binary value before the operation. After the operation, use **AAM** to adjust the quotient. The remainder is lost. If you need the remainder, save it in another register before adjusting the quotient. Then move it back to AL and adjust if necessary.

The following examples show how to use each of these instructions in BCD addition, subtraction, multiplication, and division.

```
; To add 9 and 3 as BCDs:
        mov     ax, 9       ; Load 9
        mov     bx, 3       ;  and 3 as unpacked BCDs
        add     al, bl      ; Add 09h and 03h to get 0Ch
        aaa                 ; Adjust 0Ch in AL to 02h,
                            ;  increment AH to 01h, set carry
                            ; Result 12 (unpacked BCD in AX)
```

```
; To subtract 4 from 13:
        mov     ax, 103h    ; Load 13
        mov     bx, 4       ;  and 4 as unpacked BCDs
        sub     al, bl      ; Subtract 4 from 3 to get FFh (-1)
        aas                 ; Adjust 0FFh in AL to 9,
                            ;  decrement AH to 0, set carry
                            ; Result 9 (unpacked BCD in AX)

; To multiply 9 times 3:
        mov     ax, 903h    ; Load 9 and 3 as unpacked BCDs
        mul     ah          ; Multiply 9 and 3 to get 1Bh
        aam                 ; Adjust 1Bh in AL
                            ;  to get 27 (unpacked BCD in AX)

; To divide 25 by 2:
        mov     ax, 205h    ; Load 25
        mov     bl, 2       ;  and 2 as unpacked BCDs
        aad                 ; Adjust 0205h in AX
                            ;  to get 19h in AX
        div     bl          ; Divide by 2 to get
                            ;  quotient 0Ch in AL
                            ;  remainder 1 in AH
        aam                 ; Adjust 0Ch in AL
                            ;  to 12 (unpacked BCD in AX)
                            ;  (remainder destroyed)
```

If you process multidigit BCD numbers in loops, each digit is processed and adjusted in turn.

6.4.2.2 Packed BCD Numbers

Packed BCD numbers are made up of bytes containing two decimal digits: one in the upper four bits and one in the lower four bits. The 8086-family processors provide instructions for adjusting packed BCD numbers after addition and subtraction. You must write your own routines to adjust for multiplication and division.

To do processor calculations on packed BCD numbers, you must do the eight-bit arithmetic calculations on each byte separately. The result should always be in the AL register. After each operation, use the corresponding BCD instruction to adjust the result. The decimal-adjust instructions do not take an operand. They always work on the value in the AL register.

The 8086-family processors provide **DAA** (Decimal Adjust after Addition) and **DAS** (Decimal Adjust after Subtraction) for adjusting packed BCD numbers after addition and subtraction.

These examples show **DAA** and **DAS** used for adding and subtracting BCDs.

```
;To add 88 and 33:
        mov     ax, 8833h    ; Load 88 and 33 as packed BCDs
        add     al, ah       ; Add 88 and 33 to get 0BBh
        daa                  ; Adjust 0BBh to 121 (packed BCD:)
                             ;  1 in carry and 21 in AL

;To subtract 38 from 83:
        mov     ax, 3883h    ; Load 83 and 38 as packed BCDs
        sub     al, ah       ; Subtract 38 from 83 to get 04Bh
        das                  ; Adjust 04Bh to 45 (packed BCD:)
                             ;  0 in carry and 45 in AL
```

Unlike the ASCII-adjust instructions, the decimal-adjust instructions never affect AH. The assembler sets the auxiliary carry flag if the digit in the lower four bits carries to or borrows from the digit in the upper four bits, and it sets the carry flag if the digit in the upper four bits needs to carry to or borrow from another byte.

Multidigit BCD numbers are usually processed in loops. Each byte is processed and adjusted in turn.

6.5 Related Topics in Online Help

In addition to information on the instructions and directives mentioned in this chapter, information on the following topics can be found in online help, starting from the "MASM 6.0 Contents" screen.

Topic	Access
Control registers	Choose "Language Overview," and then choose "Coprocessor Status Word," "Coprocessor Control Word," or "Coprocessor Environment"
ML options	Choose "ML Command Line"
Coprocessor instructions	Choose "Coprocessor Instructions"
MATHDEMO.ASM	Choose "Example Code" and then "Map of Demos"

Chapter 7
Controlling Program Flow

Very few programs actually execute all lines sequentially from **.STARTUP** to **.EXIT**. Rather, complex program logic and efficiency dictate that you control the flow of your program—jumping from one point to another, repeating an action until a condition is reached, and passing control to procedures. This chapter describes various means for controlling program flow and several features that simplify coding program-control constructs.

The first section covers jumps from one point in the program to another. It explains how MASM 6.0 optimizes both unconditional and conditional jumps under certain circumstances, so that you do not have to specify every attribute. The section also describes instructions you can use to test conditional jumps.

The next section describes loop and decision structures that repeat actions or evaluate conditions. They discuss some new MASM directives, such as **.WHILE** and **.REPEAT**, that generate appropriate compare, loop, and jump instructions for you, and the new **.IF**, **.ELSE**, and **.ELSEIF** directives that generate jump instructions.

A number of improvements to procedure automation are covered in Section 7.3. These include extended functionality for **PROC**, a **PROTO** directive that lets you write procedure prototypes similar to those used in C, an **INVOKE** directive that automates parameter passing, and new options for the stack-frame setup inside procedures.

Finally, the last section explains how to pass control to an interrupt routine.

7.1 Jumps

Jumps are the most direct method for changing program control from one location to another. At the processor level, jumps work by changing the value of the IP (Instruction Pointer) register from the address of the current instruction to a target address, by changing the CS register for far jumps, and by changing the CS register for far jumps. The many forms of the jump instructions handle jumps based on conditions, flags, and bit settings.

This section first describes unconditional jumps, including the new jump optimization features of MASM 6.0 and the use of indirect operands to specify the jump's destination and to construct jump tables. The section then discusses conditional jumps—extending jumps, jumps based on bit or flag status, anonymous jumps, labels for jump targets, and decision directives that generate conditional jumps.

7.1.1 Unconditional Jumps

Jumps in assembler programs are either conditional or unconditional. The assembler executes conditional jumps only when the jump condition is true. You use the **JMP** instruction to jump unconditionally to a specified address. Its single operand contains the target address, which can be short, near, or far.

Unconditional jumps are often used to skip over code that should not be executed, as shown in this example.

```
; Handle one case
label1: .
        .
        .
        jmp continue
; Handle second case
label2: .
        .
        .
        jmp continue
        .
        .
        .
continue:
```

The distance of the target from the jump instruction and the size of the operand determine the assembler's encoding of the instruction. The larger the distance, the more bytes the assembler uses to code the instruction. In previous versions of MASM, unconditional **NEAR** jumps sometimes generate inefficient code. Unspecified **FAR** jumps result in phase errors.

7.1.1.1 Jump Optimizing

Beginning with MASM 6.0, the assembler determines the smallest encoding possible for the direct unconditional jump. You do not specify a distance operator, so you do not have to determine the correct distance of the jump. If you do specify a distance, however, and it is too short, the assembler generates an error. A specified distance that is too long causes a less efficient jump to be generated than the assembler would generate if the distance had not been specified.

MASM 6.0 optimizes jumps if the following conditions are met:

- You do not specify **SHORT**, **NEAR**, **FAR**, **NEAR16**, **NEAR32**, **FAR16**, **FAR32**, or **PROC** as the distance of the target.
- The target of the jump is not external and is in the same segment as the jump instruction. If the target is in a different segment (but in the same group), it is treated as if external.

If these two conditions are met, MASM uses the instruction, distance, and size of the operand to determine how best to optimize the encoding for the jump. No syntax changes are necessary.

NOTE This information about jump optimizing also applies to conditional jumps on the 80386/486.

7.1.1.2 Indirect Operands

Indirect operands specify a register or data memory location that holds the address of the jump's destination. Indirect operands differ from the operands of direct jumps by being a memory expression instead of an immediate expression. For indirect jumps, you can specify the encoding for the instruction by giving the size (**WORD**, **DWORD**, or **FWORD**) attributes for the operand.

The default rules are based on the **.MODEL** and the default segment size.

```
        jmp     [bx]            ; Uses .MODEL and segment size
                                ;  defaults
        jmp     WORD PTR [bx] ; A NEAR16 indirect call
```

If the indirect operand is a register, the jump is always a **NEAR16** jump for a 16-bit register, and **FAR32** for a 32-bit register:

```
        jmp     bx              ; NEAR16 jump
        jmp     ebx             ; FAR32  jump
```

A **DWORD** indirect operand, however, is an ambiguous case:

```
        jmp     DWORD PTR [var] ; A NEAR32 jump in a 32-bit segment;
                                ;  a FAR16 jump in a 16-bit segment
```

In this case, you must define a type with **TYPEDEF** to specify the indirect operand.

```
NFP     TYPEDEF PTR NEAR32
FFP     TYPEDEF PTR FAR16
        jmp     NFP PTR [var] ; NEAR32 indirect jump
        jmp     FFP PTR [var] ; FAR16  indirect jump
```

You can use an unconditional jump as a form of conditional jump by specifying the address in a register or indirect memory operand. Also, you can use indirect memory operands to construct jump tables that work like C **switch** statements,

Pascal **CASE** statements, or Basic **ON GOTO**, **ON GOSUB**, or **SELECT CASE** statements, as shown in this example:

```
NPVOID  TYPEDEF NEAR PTR VOID
        .DATA
ctl_tbl NPVOID  extended,     ; Null key (extended code)
                ctrla,        ; Address of CONTROL-A key routine
                ctrlb         ; Address of CONTROL-B key routine
        .CODE
        .
        .
        .
        mov     ah, 8h        ; Get a key
        int     21h
        cbw                   ; Stretch AL into AX
        mov     bx, ax        ; Copy
        shl     bx, 1         ; Convert to address
        jmp     ctl_tbl[bx]   ; Jump to key routine

extended:
        mov     ah, 8h        ; Get second key of extended key
        int     21h
        .                     ; Use another jump table
        .                     ;  for extended keys
        .
        jmp     next
ctrla:  .                     ; CONTROL-A code here
        .
        .
        jmp     next
ctrlb:  .                     ; CONTROL-B code here
        .
        .
        jmp     next
        .
        .
next:   .                     ; Continue
```

In this example, the indirect memory operands point to addresses of routines for handling different keystrokes.

7.1.2 Conditional Jumps

The most common way to transfer control in assembly language is with a conditional jump. This is a two-step process: first test the condition, and then jump if the condition is true or continue if it is false.

The conditional jump instructions check flag status.

Conditional-jump instructions (except **JCXZ**) use the status of one or more flags as their condition. Thus, any statement that sets a flag under specified conditions can be the test statement. The most common test statements use the **CMP** or **TEST** instructions. The jump statement can be any one of 31 conditional-jump in-

structions. Conditional-jump instructions take a single operand containing the target address.

7.1.2.1 Jump Extending

In earlier versions of MASM, the **NEAR** and **FAR** operators cannot be used with conditional jumps on the 8086-80286 processors. MASM 6.0 automatically expands the jump instruction to include an unconditional jump to the destination, as long as a distance or size other than **SHORT** is specified or implicitly required from the operands. That is, MASM now generates the code that previously you had to write.

Conditional jumps cannot refer to labels more than 128 bytes away. Therefore, in versions of MASM prior to 6.0, they are often combined with unconditional jumps, which have no such limitation. For example, the following statement is valid as long as `target` is not far away:

```
; Jump to target less than 128 bytes away
        jz      target  ; If previous operation resulted in
                        ;  zero, jump to target
```

However, once `target` becomes too distant, the following sequence is necessary to enable a longer jump. Note that this sequence is logically equivalent to the example above:

```
; Jumps to distant targets previously required two steps
        jnz     skip    ; If previous operation result is
                        ;  NOT zero, jump to "skip"
        jmp     target  ;  Otherwise, jump to target
skip:
```

If the instruction is any of the conditional-jump instructions (except **JCXZ** and **JECXZ**) and the target is greater than 128 bytes or is in a far segment, then jump-extending for an instruction such as `je target` generates two instructions to replace it:

1. The logical negation of the jump instruction, with a destination that skips over the second line it generates
2. An unconditional jump to the target destination

For example, if `target` is more than 128 bytes away, MASM generates these lines of code for `je target`:

```
        jne $ + 2 + (length in bytes of the next instruction)
        jmp NEAR PTR target
```

Now the conditional jump executes correctly.

The assembler generates this same code sequence if you specify the distance with **NEAR PTR**, **FAR PTR**, or **SHORT**. Therefore,

```
        jz      NEAR PTR target
```

becomes

```
        jne     $ + 5
        jmp     NEAR PTR target
```

even if `target` is nearby.

When `skip` is more than 128 bytes away, this example

```
        mov     ax, cx
        jz      skip      ; Skip is more than 128 bytes away
        .
        .                 ; (additional code here)
        .
skip:
```

generates code that looks like this:

```
7327:0000 8BC1          MOV       AX,CX
7327:0002 7503          JNZ       0007
7327:0004 E9C000        JMP       00C7
7327:0007               (more code here)
```

MASM 6.0 enables this jump expansion feature by default, but you can turn it off with the **NOLJMP** form of the **OPTION** directive. See Section 1.3.2 for information about the **OPTION** directive.

If the assembler generates code to extend a conditional jump, it issues a level 3 warning saying that the conditional jump has been lengthened. You can set the warning level to 1 for development and to level 3 for a final optimizing pass to see if you can shorten jumps by reorganizing.

If you specify the distance for the jump and the target is out of range for that distance, a "Jump out of Range" error results.

Since the **JCXZ** and **JECXZ** instructions do not have logical negations, expansion of the jump instruction to handle targets with unspecified distances cannot be performed for those instructions. Therefore the distance must always be short.

The size and distance of the target operand determines the encoding for conditional or unconditional jumps to externals or targets in different segments. The new jump-extending and optimization features do not apply in this case.

NOTE Conditional jumps on the 80386 and 80486 processors can be to targets up to 32K bytes away, so jump extension occurs only for targets greater than that distance.

7.1.2.2 Jumps Based on Comparisons

The **CMP** instruction is specifically designed to test for conditional jumps. It does not change the destination operand—it compares two values without changing either of them. Instructions that change operands (such as **SUB** or **AND**) can also be used to test conditions.

SUB and CMP set the same flags.

Internally, the **CMP** instruction is the same as the **SUB** instruction, except that **CMP** does not change the destination operand. Both set flags according to the result that the subtraction generates.

Table 7.1 lists conditional-jump instructions for each comparison relationship and shows the flags that are tested to see if the relationship is true. Note the difference in instructions depending on the sign of the operands. Some of these are equivalent to instructions listed in the previous section.

Table 7.1 Conditional-Jump Instructions Used after Compare Instruction

Jump Condition	Signed Compare	Flags Tested (Jump if True)	Unsigned Compare	Flags Tested (Jump if True)
= (Equal)	**JE**	ZF = 1	**JE**	ZF = 1
≠ (Not equal)	**JNE**	ZF = 0	**JNE**	ZF = 0
> (Greater than)	**JG** or **JNLE**	ZF = 0 and SF = OF	**JA** or **JNBE**	CF = 0 and ZF = 0
<= (Less than or equal to)	**JLE** or **JNG**	ZF = 1 or SF ≠ OF	**JBE** or **JNA**	CF = 1 or ZF = 1
< (Less than)	**JL** or **JNGE**	SF ≠ OF	**JB** or **JNAE**	CF = 1
>= (Greater than or equal to)	**JGE** or **JNL**	SF = OF	**JAE** or **JNB**	CF = 0

In the **CMP** instruction, the mnemonic names always refer to the relationship of the first operand to the second operand. For instance, in this example **JG** tests whether the first operand is greater than the second.

```
cmp     ax, bx  ; Compares ax and bx
jg      contin  ; Equivalent to:  If ( ax > bx ) goto
                ;  contin
jl      next    ; Equivalent to:  If ( ax < bx ) goto next
```

Several conditional instructions have two names. For example, **JG** and **JNLE** (Jump if Not Less or Equal) are equivalent. You can use whichever name seems more mnemonic in context.

7.1.2.3 Testing Bits and Jumping

Using **CMP** is not the only way to check a condition prior to a jump. You can also check the status of bits in the operands using the **TEST** instruction. This instruction tests for conditions prior to jumps by comparing specific bits rather than entire operands. Jump execution depends on whether certain bits are on or off.

Pairs of operands cannot be both registers or both memory locations.

The **TEST** instruction is the same as the **AND** instruction, except that **TEST** changes neither operand. If the result of the operation is 0, the zero flag is set, but the 0 is not actually written to the destination operand. The following example shows an application of **TEST**.

```
        .DATA
bits    BYTE    ?
        .CODE
        .
        .
        .
; If bit 2 or bit 4 is set, then call task_a
                            ; Assume "bits" is 0D3h    11010011
        test    bits, 10100y ; If 2 or 4 is set     AND 00010100
        jz      skip1        ;                          --------
        call    task_a       ; Then call task_a         00010000
skip1:                       ; Jump taken
        .
        .
        .
; If bits 2 and 4 are clear, then call task_b
                            ; Assume "bits" is 0E9h    11101001
        test    bits, 10100y ; If 2 and 4 are clear AND 00010100
        jnz     skip2        ;                          --------
        call    task_b       ; Then call task_b         00000000
skip2:                       ; Jump taken
```

Generally, when you use **TEST**, one of the operands is a mask in which the bits to be tested are the only bits set. The other operand contains the value to be tested. If all the bits set in the mask are clear in the operand being tested, the zero flag is set. If any of the flags set in the mask are also set in the operand, the zero flag is cleared.

7.1.2.4 Jumping Based on Flag Status

Your code can jump based on the condition of flags rather than on the relationships of operands. Use the following conditional-jump instructions:

Instruction	Jumps if
JO	The overflow flag is set
JNO	The overflow flag is clear
JC	The carry flag is set (same as **JB**)

Instruction	Jumps if
JNC	The carry flag is clear (same as **JAE**)
JZ	The zero flag is set (same as **JE**)
JNZ	The zero flag is clear (same as **JNE**)
JS	The sign flag is set
JNS	The sign flag is clear
JP	The parity flag is set
JNP	The parity flag is clear
JPE	Parity is even (parity flag set)
JPO	Parity is odd (parity flag clear)
JCXZ	CX is 0
JECXZ (80386/486 only)	ECX is 0

The following example shows two ways to use the instructions from the list above:

```
; Uses JO to handle overflow condition
        add     ax, bx      ; Add two values
        jo      overflow    ; If value too large, adjust

; Uses JNZ to check for zero as the result of subtraction
        sub     ax, bx      ; Subtract
        jnz     skip        ; If the result is not zero, continue
        call    zhandler    ; Else do special case
```

7.1.2.5 Anonymous Labels

Anonymous labels are alternatives to named labels.

Coding jumps in assembly language requires that you invent many label names. One alternative to continually thinking up new label names is using anonymous labels, which you can use anywhere in your program. But because anonymous labels do not provide meaningful names, they are best used for conditionally testing a few lines of code. You should mark major divisions of a program with actual named labels.

Use two at signs (@@) followed by a colon (:) as an anonymous label. To jump to the nearest preceding anonymous label, use **@B** (back) in the jump instruction's operand field; to jump to the nearest following anonymous label, use **@F** (forward) in the operand field.

The jump in the example below uses an anonymous label:

```
; DX is 20, unless CX is less than -20, then make DX 30
        mov     dx, 20
        cmp     cx, -20
        jge     @F
        mov     dx, 30
@@:
```

The items `@B` and `@F` always refer to the nearest occurrences of @@:, so there is never any conflict between different anonymous labels.

7.1.2.6 Decision Directives

The high-level structures you can use for decision-making are the **.IF**, **.ELSEIF**, and **.ELSE** statements. These directives generate conditional jumps. The expression following the **.IF** directive is evaluated, and if true, the following instructions are executed until the next **.ENDIF**, **.ELSE**, or **.ELSEIF** directive is reached. The **.ELSE** statements execute if the expression is false. Using the **.ELSEIF** directive puts a new expression to be evaluated inside the alternative part of the original **.IF** statement. The syntax is

.IF *condition1*
statements
[[**.ELSEIF** *condition2*
statements]]
[[**.ELSE**
statements]]
.ENDIF

The decision structure

```
        .IF     cx = 20
        mov     dx, 20
        .ELSE
        mov     dx, 30
        .ENDIF
```

generates this code:

```
                            .IF  cx == 20
0017  83 F9 14          *         cmp    cx, 014h
001A  75 05             *         jne    @C0001
001C  BA 0014                     mov    dx, 20
                            .ELSE
001F  EB 03             *         jmp    @C0003
0021                    *@C0001:
0021  BA 001E                     mov   dx, 30
                            .ENDIF
0024                    *@C0003:
```

7.2 Loops

Loops repeat an action until a termination condition is reached. This condition can be a counter or the result of an expression's evaluation. MASM 6.0 offers many ways to set up loops in your programs. The following list compares MASM loop structures.

Instructions	Action
LOOP	Automatically decrements CX. When CX = 0, the loop ends. The top of the loop cannot be greater than 128 bytes from the **LOOP** instruction. (This is true for all **LOOP** instructions.)
LOOPE, LOOPZ, LOOPNE, LOOPNZ	Loops while equal (or not equal). Checks CX and a condition. The loop ends when the condition is true. Set CX to a number out of range if you don't want a count to control the loop.
JCXZ, JECXZ	Branches to a label only if CX = 0 (ECX on the 80386). Useful for testing condition of CX before beginning loop. If CX = 0 before entering the loop, CX decrements to –1 on the first iteration and then must be decremented 65,535 times before it reaches 0 again. Unlike conditional-jump instructions, which can jump to either a near or a short label under the 80386 or 80486, the loop instructions **JCXZ** and **JECXZ** always jump to a short label.
Conditional jumps	Acts only if certain conditions met. Necessary if several conditions must be tested. See Section 7.1.2, "Conditional Jumps."

The following examples illustrate these loop constructions.

```
; The LOOP instruction: For 200 to 0 do task
        mov     cx, 200         ; Set counter
next:   .                       ; Do the task here
        .
        .
        loop    next            ; Do again
                                ; Continue after loop
```

```
; The LOOPNE instruction: While AX is not 'Y', do task
        mov     cx, 256         ; Set count too high to interfere
wend:   .                       ; But don't do more than 256 times
        .                       ; Some statements that change AX
        .
        cmp     al, 'Y'         ; Is it Y or too many times?
        loopne  wend            ; No? Repeat
                                ; Yes? Continue

; Using JCXZ:  For 0 to CX do task
                                ; CX counter set previously
        jcxz    done            ; Check for 0
next:   .                       ; Do the task here
        .
        .
        loop    next            ; Do again
done:                           ; Continue after loop
```

7.2.1 Loop-Generating Directives

These directives are new to MASM 6.0.

The high-level control structures new to MASM 6.0 generate loop structures for you. These new directives are similar to the **while** and **repeat** loops of C or Pascal. They can make your assembly programs less repetitive and easier to code, as well as easier to read. The assembler generates the appropriate assembly code. The **.BREAK** and **.CONTINUE** directives are also implemented to interrupt loop execution. These directives are summarized in the following list:

Directives	Action
.WHILE, .ENDW	The statements between **.WHILE** *condition* and **.ENDW** execute while the condition is true.
.REPEAT, .UNTIL	The loop executes at least once and continues until the condition given after **.UNTIL** is true. Generates conditional jumps.
.REPEAT, .UNTILCXZ	Compares label to an expression and generates appropriate loop instructions.

These constructs work much as they do in a high-level language such as C or Pascal. Keep in mind the following points:

- These directives generate appropriate processor instructions. They are not new instructions.
- They require proper use of signed and unsigned data declarations.

These directives cause a set of instructions to execute based on the evaluation of some *condition*. This *condition* can be an expression that evaluates to a negative or nonnegative value, an expression using the binary operators in C (&&, ||, or !), or the state of a flag. See Section 7.2.2.1 for more information about expression operators.

The evaluation of the *condition* requires the assembler to know if the operands in the condition are signed or unsigned. To state explicitly that a named memory location contains a signed integer, use the signed data allocation directives: **SBYTE**, **SWORD**, and **SDWORD**.

7.2.1.1 .WHILE Loops

As with **while** loops in C or Pascal, the test condition for **.WHILE** is checked before the statements inside the loop execute. If the test condition is false, the loop does not execute. While the condition is true, the statements inside the loop repeat.

Use the **.ENDW** directive to mark the end of the **.WHILE** loop. When the condition becomes false, program execution begins at the first statement following the **.ENDW** directive. The **.WHILE** directive generates appropriate compare and jump statements. The syntax is

.WHILE *condition*
statements
.ENDW

For example, this loop copies one buffer to another until a '$' character (marking the end of the string) is found:

```
.DATA
buf1    BYTE "This is a string",'$'
buf2    BYTE 100 DUP (?)
.CODE
sub     bx, bx              ; Zero out bx
.WHILE  (buf1[bx] != '$')
mov     al, buf1[bx]        ; Get a character
mov     buf2[bx], al        ; Move it to buffer 2
inc     bx                  ; Count forward
.ENDW
```

7.2.1.2 .REPEAT Loops

MASM's **.REPEAT** directive allows for loop constructions like the **do** loop of C and the **REPEAT** loop of Pascal. The loop executes until the condition following the **.UNTIL** (or **.UNTILCXZ**) directive becomes true. Since the condition is checked at the end of the loop, the loop always executes at least once. The **.REPEAT** directive generates conditional jumps. The syntax is:

.REPEAT
statements
.UNTIL *condition*

.REPEAT
statements
.UNTILCXZ [[*condition*]]

A condition is optional with .UNTILCXZ.

where *condition* can also be *expr1 == expr2* or *expr1 != expr2*. When two conditions are used, *expr2* can be an immediate expression, a register, or (if *expr1* is a register) a memory location.

For example, the following code fills up a buffer with characters typed at the keyboard. The loop ends when the ENTER key (character 13) is pressed:

```
        .DATA
buffer  BYTE    100 DUP (0)
        .CODE
        sub     bx, bx              ; Zero out bx
        .REPEAT
        mov     ah, 01h
        int     21h                 ; Get a key
        mov     buffer[bx], al      ; Put it in the buffer
        inc     bx                  ; Increment the count
        .UNTIL (al == 13)           ; Continue until al is 13
```

The **.UNTIL** directive generates conditional jumps, but the **.UNTILCXZ** directive generates a **LOOP** instruction, as shown by the listing file code for these examples. In a listing file, assembler-generated code is preceded by an asterisk.

```
    ASSUME  bx:PTR SomeStruct

            .REPEAT
      *@C0001:
                inc     ax
            .UNTIL  ax==6
      *         cmp     ax, 006h
      *         jne     @C0001
```

```
     .REPEAT
*@C0003:
          mov    ax, 1
     .UNTILCXZ
*         loop   @C0003

     .REPEAT
*@C0004:
     .UNTILCXZ   [bx].field != 6
*         cmp    [bx].field, 006h
*         loope  @C0004
```

7.2.1.3 .BREAK and .CONTINUE Directives

.BREAK and .CONTINUE interrupt loop execution.

The **.BREAK** and **.CONTINUE** directives can be used to terminate a **.REPEAT** or **.WHILE** loop prematurely. These directives allow an optional **.IF** clause for conditional breaks. The syntax is

.BREAK [[**.IF** *condition*]]
.CONTINUE [[**.IF** *condition*]]

Note that **.ENDIF** is not used with the **.IF** forms of **.BREAK** and **.CONTINUE** in this context. The **.BREAK** and **.CONTINUE** directives work the same way as the **break** and **continue** instructions in C. Execution continues at the instruction following the **.UNTIL**, **.UNTILCXZ**, or **.ENDW** of the nearest enclosing loop.

Instead of causing the loop execution to end as **.BREAK** does, **.CONTINUE** causes loop execution to jump directly to the code that evaluates the loop condition of the nearest enclosing loop.

The following loop accepts only the keys in the range '0' to '9' and terminates when ENTER is pressed.

```
.WHILE 1                ; Loop forever
mov     ah, 08h         ; Get key without echo
int     21h
.BREAK .IF al == 13     ; If ENTER, break out of the loop
.CONTINUE .IF (al < '0') || (al > '9')
                        ; If not a digit, continue looping
mov     dl, al          ; Save the character for processing
mov     ah, 02h         ; Output the character
int     21h
.ENDW
```

If you assemble the source code above with the /Fl and /Sg command-line options and then view the results in the listing file, you would see this code:

```
                        .WHILE 1
0017                    *@C0001:
0017  B4 08                       mov     ah, 08h
0019  CD 21                       int     21h
                        .BREAK .IF al == 13
001B  3C 0D             *         cmp     al, 00Dh
001D  74 10             *         je      @C0002
                        .CONTINUE .IF (al  '0') || (al  '9')
001F  3C 30             *         cmp     al, '0'
0021  72 F4             *         jb      @C0001
0023  3C 39             *         cmp     al, '9'
0025  77 F0             *         ja      @C0001
0027  8A D0                       mov     dl, al
0029  B4 02                       mov     ah, 02h
002B  CD 21                       int     21h
                        .ENDW
002D  EB E8             *         jmp     @C0001
002F                    *@C0002:
```

The high-level control structures can be nested. That is, **.REPEAT** or **.WHILE** loops can contain **.REPEAT** or **.WHILE** loops as well as **.IF** statements.

If the code generated by a **.WHILE** loop, **.REPEAT** loop, or **.IF** statement generates a conditional or unconditional jump, MASM uses the jump extension and jump optimization techniques described in Sections 7.1.1, "Unconditional Jumps," and 7.1.2, "Conditional Jumps," to encode the jump appropriately.

7.2.2 Writing Loop Conditions

You can express the conditions of the **.IF**, **.REPEAT**, and **.WHILE** directives using relational operators, and you can express the attributes of the operand with the **PTR** operator. To write loop conditions, you also need to know how the assembler evaluates the operators and operands in the condition. This section explains the operators, attributes, precedence level, and expression evaluation order for the conditions used with loop-generating directives.

7.2.2.1 Expression Operators

The binary relational operators in MASM 6.0 high-level control structures are listed below. The same binary operators are used in C. These operators generate MASM compare, test, and conditional jump instructions.

Operator	Meaning
==	Equal
!=	Not equal

Operator	Meaning
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
&	Bit test
!	Logical NOT
&&	Logical AND
||	Logical OR

A condition without operators (other than **!**) tests for nonzero as it does in C. For example, `.WHILE (x)` is the same as `.WHILE (x != 0)`, and `.WHILE (!x)` is the same as `.WHILE (x == 0)`.

Flag names can be operands in a condition.

You can also use the flag names (**ZERO?**, **CARRY?**, **OVERFLOW?**, **SIGN?**, and **PARITY?**) as operands in conditions with the high-level control structures as in `.WHILE (CARRY?)`. The particular flag set determines the outcome of the condition. Use flag names when you want to generate the compare or other instructions that set the flags.

7.2.2.2 Signed and Unsigned Operands

Registers, constants, and memory locations are unsigned by default.

Expression operators generate unsigned jumps by default. However, if either side of the operation is signed, then the entire operation is considered signed. The default for the operands in registers, constants, and named memory locations is also to be unsigned.

You can use the **PTR** operator to tell the assembler that a particular operand in a register or constant is a signed number, as in these examples:

```
        .WHILE  SWORD PTR [bx] <= 0
        .IF     SWORD PTR mem1 >  0
```

Without the **PTR** operator, the assembler would treat the contents of BX as an unsigned value.

You can also specify the size attributes of operands in memory locations with **SBYTE**, **SWORD**, and **SDWORD**, for use with **.IF**, **.WHILE**, and **.REPEAT**.

```
        .DATA
mem1    SBYTE   ?
mem2    WORD    ?
        .IF     mem1 > 0
        .WHILE  mem2 < bx
        .WHILE  SWORD PTR ax < count
```

7.2.2.3 Precedence Level

As with C, you can concatenate conditions with the **&&** operator for AND, the || operator for OR, and the **!** operator for negate. The precedence level is **!**, **&&**, and ||, with **!** having the highest precedence. Like expressions in high-level languages, associativity is evaluated left to right.

7.2.2.4 Expression Evaluation

The assembler evaluates conditions created with high-level control structures according to short-circuit evaluation. If the evaluation of a particular condition automatically determines the final result (such as a condition that evaluates to false in a compound statement concatenated with **AND**), the evaluation does not continue.

For example, in this **.WHILE** statement,

```
.WHILE (ax > 0) && (WORD PTR [bx] == 0)
```

the assembler evaluates the first condition. If this condition is false (that is, if AX is less than or equal to 0), the evaluation is finished. The second condition is not checked and the loop does not execute, because a compound condition containing a **&&** requires both expressions to be true for the entire condition to be true.

7.3 Procedures

Organizing your code into procedures that execute specific tasks divides large programs into manageable units, allows for separate testing, and makes code more efficient for repetitive tasks.

Assembly-language procedures are comparable to functions in C; subprograms, functions, and subroutines in Basic; procedures and functions in Pascal; or subroutines and functions in FORTRAN.

Two instructions control the use of assembly-language procedures; **CALL** pushes the return address onto the stack and transfers control to a procedure, and **RET** pops the return address off the stack and returns control to that location.

The **PROC** and **ENDP** directives mark the beginning and end of a procedure. Additionally, **PROC** can automatically

- Preserve register values that should not change but that the procedure might otherwise alter
- Set up a local stack pointer, so that you can access parameters and local variables placed on the stack
- Adjust the stack when the procedure ends

Sections 7.3.1 through 7.3.3 give information on techniques for calling procedures and accessing parameters. Sections 7.3.4 through 7.3.5 show how to allocate and access local variables and parameters.

Sections 7.3.6 and 7.3.7 introduce new directives in MASM 6.0 to further automate calling procedures and passing arguments. The **PROTO** directive allows you to declare prototypes for your procedures. **INVOKE** handles procedure calls and stack cleanup. Section 7.3.8 describes the automatic stack setup and cleanup generated with **PROC**.

7.3.1 Defining Procedures

Procedures require a label at the start of the procedure and a return at the end. Procedures are normally defined by using the **PROC** directive at the start of the procedure and the **ENDP** directive at the end. The **RET** instruction is normally placed immediately before the **ENDP** directive. The assembler makes sure that the distance of the **RET** instruction matches the distance defined by the **PROC** directive. The basic syntax for **PROC** is

label **PROC** [[**NEAR** | **FAR**]]
    .
    .
    .
RET [[*constant*]]
label **ENDP**

The **CALL** instruction pushes the address of the next instruction in your code onto the stack and passes control to a specified address. The syntax is

CALL {*label* | *register* | *memory*}

The operand contains a value calculated at run time. Since that operand can be a register, direct memory operand, or indirect memory operand, you can write call tables similar to the jump table illustrated in Section 7.1.1.2.

Calls can be near or far. Near calls push only the offset portion of the calling address and therefore must be within the same segment or group. You can specify the type for the target operand, but if you do not, MASM uses the declared distance (**NEAR** or **FAR**) for operands that are labels and for the size of register or memory operands. Then the assembler encodes the call appropriately, as it does with unconditional jumps (see Sections 7.1.1, "Unconditional Jumps," and 7.1.2, "Conditional Jumps").

MASM 6.0 optimizes a call to a far label when the label is in the current segment by generating the code for a near call, saving one byte.

You can define procedures without **PROC** and **ENDP**, but if you do, you must make sure that the size of the **CALL** matches the size of the **RET**. You can specify the **RET** instruction as **RETN** (Return Near) or **RETF** (Return Far) to override the default size:

```
        call    NEAR PTR task ; Call is declared near
        .                     ; Return comes to here
        .
        .
task:                         ; Procedure begins with near label
        .
        .                     ; Instructions go here
        .
        retn                  ; Return declared near
```

The syntax for **RETN** and **RETF** is

label: | *label* **NEAR**
statements
RETN [[*constant*]]

label **LABEL FAR**
statements
RETF [[*constant*]]

The **RET** instruction (and its **RETF** and **RETN** variations) allows an optional constant operand that specifies a number of bytes to be added to the value of the SP register after the return. This operand adjusts for arguments passed to the procedure before the call, as shown in the example in Section 7.3.4, "Using Local Variables."

Incorrect size for RET can cause your program to fail.

When you define procedures without **PROC** and **ENDP**, you must make sure that calls have the same size as corresponding returns. For example, **RETF** pops two words off the stack. If a **NEAR** call is made to a procedure with a far return, not only is the popped value meaningless, but the stack status may cause the execution to return to a random memory location, resulting in program failure.

There is an also an extended **PROC** syntax that automates many of the details of accessing arguments and saving registers. See Section 7.3.3, "Declaring Parameters with the PROC Directive."

7.3.2 Passing Arguments on the Stack

Each time you call a procedure, you may want it to operate on different data. This data, called "arguments," can be passed in various ways. For example, arguments can be passed to a procedure in registers or in variables. However, the

most common method of passing arguments is to use the stack. Microsoft languages have specific conventions for passing arguments. Chapter 20, "Mixed-Language Programming," explains these conventions for assembly-language modules shared with modules from high-level languages.

This section describes how a procedure accesses the arguments passed to it on the stack. Each argument is accessed as an offset from BP. However, if you use the **PROC** directive to declare parameters, the assembler calculates these offsets for you and lets you refer to parameters by name. The next section, "Declaring Parameters with the PROC Directive," explains how to use **PROC** this way.

This example shows how to pass arguments to a procedure. The procedure expects to find those arguments on the stack. As this example shows, arguments must be accessed as offsets of BP.

```
; C-style procedure call and definition

        mov     ax, 10      ; Load and
        push    ax          ;  push constant as third argument
        push    arg2        ; Push memory as second argument
        push    cx          ; Push register as first argument
        call    addup       ; Call the procedure
        add     sp, 6       ; Destroy the pushed arguments
        .                   ;  (equivalent to three pops)
        .
        .
addup   PROC    NEAR        ; Return address for near call
                            ;  takes two bytes
        push    bp          ; Save base pointer - takes two bytes
                            ;  so arguments start at fourth byte
        mov     bp, sp      ; Load stack into base pointer
        mov     ax, [bp+4]  ; Get first argument from
                            ;  fourth byte above pointer
        add     ax, [bp+6]  ; Add second argument from
                            ;  sixth byte above pointer
        add     ax, [bp+8]  ; Add third argument from
                            ;  eighth byte above pointer
        mov     sp, bp
        pop     bp          ; Restore BP
        ret                 ; Return result in AX
addup   ENDP
```

Figure 7.1 shows the stack condition at key points in the process.

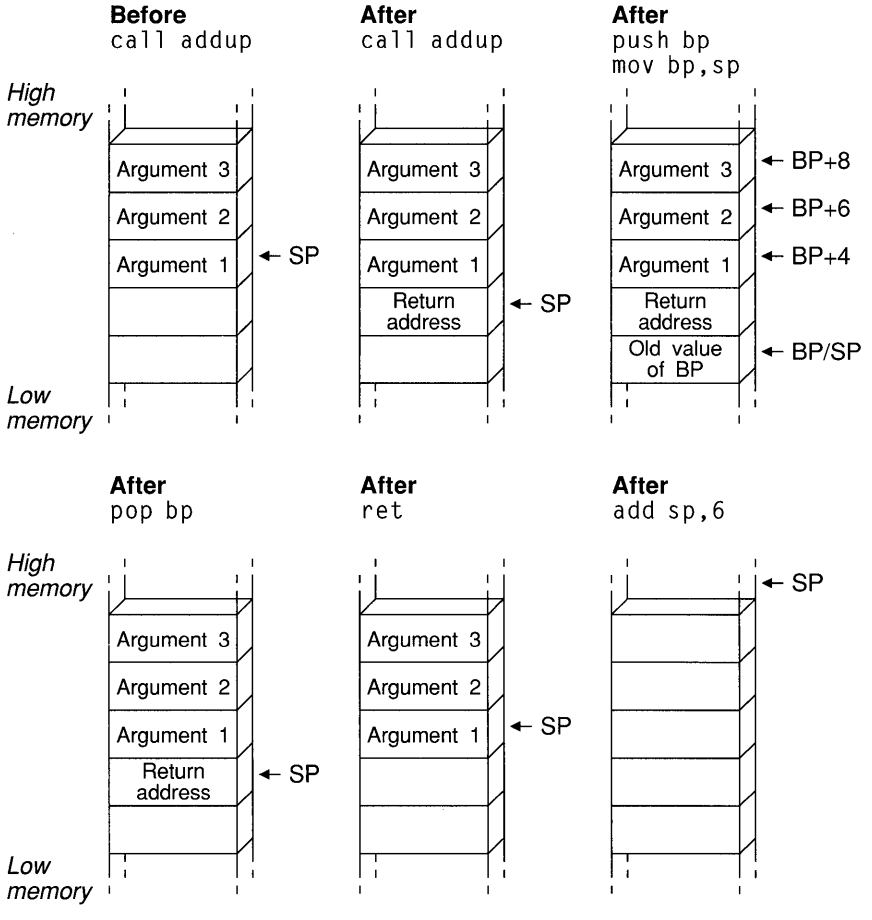


Figure 7.1 Procedure Arguments on the Stack

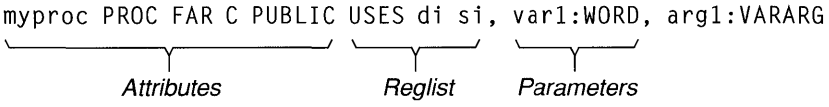
Starting with the 80186 processor, the **ENTER** and **LEAVE** instructions simplify the stack setup and restore instructions at the beginning and end of procedures.

However, **ENTER** uses a lot of time. It is necessary only with nested, statically scoped procedures. Thus, a Pascal compiler may sometimes generate **ENTER**. The **LEAVE** instruction, on the other hand, is an efficient way to do the stack cleanup. **LEAVE** reverses the effect of the last **ENTER** instruction by restoring BP and SP to their values before the procedure call.

7.3.3 Declaring Parameters with the PROC Directive

With the **PROC** directive, you can specify registers to be saved, define parameters to the procedure, and assign symbol names to parameters (rather than as offsets from BP). This section describes how to use the **PROC** directive to automate the parameter-accessing techniques described in the last section.

For example, the diagram below shows a valid **PROC** statement for a procedure called from C. It takes two parameters, `var1` and `arg1`, and uses (and must save) the DI and SI registers:



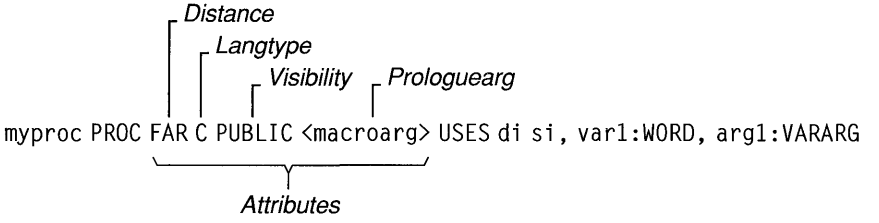
The syntax for **PROC** is

label **PROC** [[*attributes*]] [[**USES** *reglist*]] [[, *parameter*[[:*tag*]]...]]

The following list describes the parts of the **PROC** directive.

Argument	Description
label	The name of the procedure.
attributes	Any of several attributes of the procedure, including the distance, *langtype*, and *visibility* of the procedure. The syntax for *attributes* is given in Section 7.3.3.1.
reglist	A list of registers following the **USES** keyword that the procedure uses and that should be saved on entry. Registers in the list must be separated by blanks or tabs, not by commas. The assembler generates prologue code to push these registers onto the stack. When you exit, the assembler generates epilogue code to pop the saved register values off the stack.
parameter	The list of parameters passed to the procedure on the stack. The list can have a variable number of parameters. See the discussion below for the syntax of *parameter*. This list can be longer than one line if the continued line ends with a comma.

This diagram shows a valid **PROC** definition that uses several attributes:



7.3.3.1 Attributes

The syntax for the attributes field is

[[*distance*]] [[*langtype*]] [[*visibility*]] [[*<prologuearg>*]]

The list below explains each of these options.

Argument	Description
distance	Controls the form of the **RET** instruction generated. Can be **NEAR** or **FAR**. If *distance* is not specified, it is determined from the model declared with the **.MODEL** directive. For **TINY, SMALL, COMPACT**, and **FLAT, NEAR** is assumed. For **MEDIUM, LARGE**, and **HUGE, FAR** is assumed. For 80386/486 programming with 16- and 32-bit segments, **NEAR16, NEAR32, FAR16,** or **FAR32** can be specified.
langtype	Determines the calling convention used to access parameters and restore the stack. The **BASIC**, **FORTRAN**, and **PASCAL** *langtypes* convert procedure names to uppercase, place the last parameter in the parameter list lowest on the stack, and generate a **RET**, which adjusts the stack upward by the number of bytes in the argument list. The **C** and **STDCALL** *langtype* prefixes an underscore to the procedure name when the procedure's scope is **PUBLIC** or **EXPORT** and places the first parameter lowest on the stack. **SYSCALL** is equivalent to the **C** calling convention with no underscore prefixed to the procedure's name. **STDCALL** uses caller stack cleanup when **:VARARG** is specified; otherwise the called routine must clean up the stack (see Chapter 20).

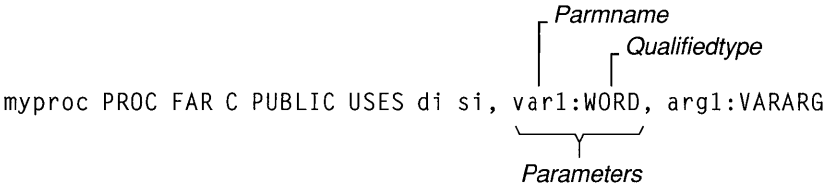
Argument	Description
visibility	Indicates whether the procedure is available to other modules. The *visibility* can be **PRIVATE, PUBLIC**, or **EXPORT**. A procedure name is **PUBLIC** unless it is explicitly declared as **PRIVATE**. If the *visibility* is **EXPORT**, the linker places the procedure's name in the export table for segmented executables. **EXPORT** also enables **PUBLIC** visibility. You can explicitly set the default *visibility* with the **OPTION** directive. **OPTION PROC:PUBLIC** sets the default to public. See Section 1.3.2 for more information.
prologuearg	Specifies the arguments that affect the generation of prologue and epilogue code (the code MASM generates when it encounters a **PROC** directive or the end of a procedure). See Section 7.3.8 for an explanation of prologue and epilogue code.

7.3.3.2 Parameters

The *parameters* are separated from the *reglist* by a comma if there is a list of registers. In the syntax:

parmname [[*:tag*]

parmname is the name of the parameter. The *tag* can be either the *qualifiedtype* or the keyword **VARARG**. However, only the last parameter in a list of parameters can use the **VARARG** keyword. The *qualifiedtype* is discussed in Section 1.2.6, "Data Types." An example showing how to reference **VARARG** parameters appears later in this section. Procedures can be nested if they do not have parameters or **USES** register lists. This diagram shows a procedure definition with one parameter definition.



The following example shows the procedure in Section 7.3.2, "Passing Arguments on the Stack," rewritten to use the extended **PROC** functionality. Prior to the procedure call, you must push the arguments onto the stack unless you use **INVOKE** (see Section 7.3.7, "Calling Procedures with INVOKE").

```
addup   PROC NEAR C,
        arg1:WORD, arg2:WORD, count:WORD
        mov     ax, arg1
        add     ax, count
        add     ax, arg2
        ret
addup   ENDP
```

If the arguments for a procedure are pointers, the assembler does not generate any code to get the value or values that the pointers reference; your program must still explicitly treat the argument as a pointer. (See Chapter 3, "Using Addresses and Pointers," for more information about using pointers.)

In the example below, even though the procedure declares the parameters as near pointers, you still must code two **MOV** instructions to get the values of the parameters—the first **MOV** gets the address of the parameters, and the second **MOV** gets the parameter.

```
; Call from C as a FUNCTION returning an integer

        .MODEL medium, c
        .CODE
myadd   PROC   arg1:NEAR PTR WORD, arg2:NEAR PTR WORD

        mov     bx, arg1      ; Load first argument
        mov     ax, [bx]
        mov     bx, arg2      ; Add second argument
        add     ax, [bx]

        ret

myadd   ENDP
        END
```

You can use conditional-assembly directives to make sure that your pointer parameters are loaded correctly for the memory model. For example, the following version of `myadd` treats the parameters as **FAR** parameters if necessary:

```
        .MODEL  medium, c       ; Could be any model
        .CODE
myadd   PROC    arg1:PTR WORD,   arg2:PTR WORD

        IF      @DataSize
        les     bx, arg1        ; Far parameters
        mov     ax, es:[bx]
        les     bx, arg2
        add     ax, es:[bx]
```

```
        ELSE
        mov     bx, arg1        ; Near parameters
        mov     ax, [bx]
        mov     bx, arg2
        add     ax, [bx]
        ENDIF

        ret
myadd   ENDP

        END
```

7.3.3.3 Using VARARG

In the **PROC** statement, you can append the **:VARARG** keyword to the last parameter to indicate that a variable number of arguments can be passed if you use the **C**, **SYSCALL**, or **STDCALL** calling conventions (see Section 20.1). A label must precede **:VARARG** so that the arguments can be accessed as offsets from the variable name given. This example illustrates **VARARG**:

```
addup3  PROTO NEAR C, argcount:WORD, arg1:VARARG

        invoke  addup3, 3, 5, 2, 4

addup3  PROC    NEAR C, argcount:WORD, arg1:VARARG
        sub     ax, ax          ; Clear work register
        sub     si, si

        .WHILE  argcount > 0    ; Argcount has number of arguments
        add     ax, arg1[si]    ; Arg1 has the first argument
        dec     arg1            ; Point to next argument
        inc     si
        inc     si
        .ENDW

        ret                     ; Total is in AX
addup3  ENDP
```

Passing non-default-sized pointers in the **VARARG** portion of the parameter list can be done by explicitly passing the segment portion and the offset portion of the address separately.

NOTE When you use the extended **PROC** features and the assembler encounters a **RET** instruction, it automatically generates instructions to pop saved registers, remove local variables from the stack, and, if necessary, remove parameters. It generates this code for each **RET** instruction it encounters. You can reduce code size by having only one return and jumping to it from various locations.

7.3.4 Using Local Variables

In high-level languages, local variables are visible only within a procedure. In Microsoft languages, these variables are usually stored on the stack. In assembly-language programs, you can also have local variables. These variables should not be confused with labels or variable names that are local to a module, as described in Chapter 8, "Sharing Data and Procedures among Modules and Libraries."

This section outlines the standard methods for creating local variables. The next section shows how to use the **LOCAL** directive to make the assembler automatically generate local variables. When you use this directive, the assembler generates the same instructions as those used in this section but handles some of the details for you.

If your procedure has relatively few variables, you can usually write the most efficient code by placing these values in registers. Local (stack) data is more efficient when you have a large amount of local data for the procedure.

Local variables are stored on the stack.

To use local variables you must save stack space for the variable at the start of the procedure. The variable can then be accessed by its position in the stack. At the end of the procedure, you need to restore the stack pointer, which restores the memory used by local variables.

This example subtracts two bytes from the SP register to make room for a local word variable. This variable can then be accessed as `[bp-2]`.

```
        push    ax                  ; Push one argument
        call    task                ; Call
        .
        .
        .

task    PROC    NEAR
        push    bp                  ; Save base pointer
        mov     bp, sp              ; Load stack into base pointer
        sub     sp, 2               ; Save two bytes for local
                                    ;  variable
        .
        .
        .
        mov     WORD PTR [bp-2], 3 ; Initialize local variable
        add     ax, [bp-2]          ; Add local variable to AX
        sub     [bp+4], ax          ; Subtract local from argument
        .                           ; Use [bp-2] and [bp+4] in
        .                           ;  other operations
        .
        mov     sp, bp              ; Clear local variables
        pop     bp                  ; Restore base
        ret     2                   ; Return result in AX and pop
task    ENDP                        ;  two bytes to clear parameter
```

Notice that the instruction `mov sp,bp` at the end of the procedure restores the original value of SP. The statement is required only if the value of SP is changed inside the procedure (usually by allocating local variables). The argument passed to the procedure is removed with the **RET** instruction. Contrast this to the example in Section 7.3.2, "Passing Arguments on the Stack," in which the calling code adjusts the stack for the argument.

Figure 7.2 shows the state of the stack at key points in the process.

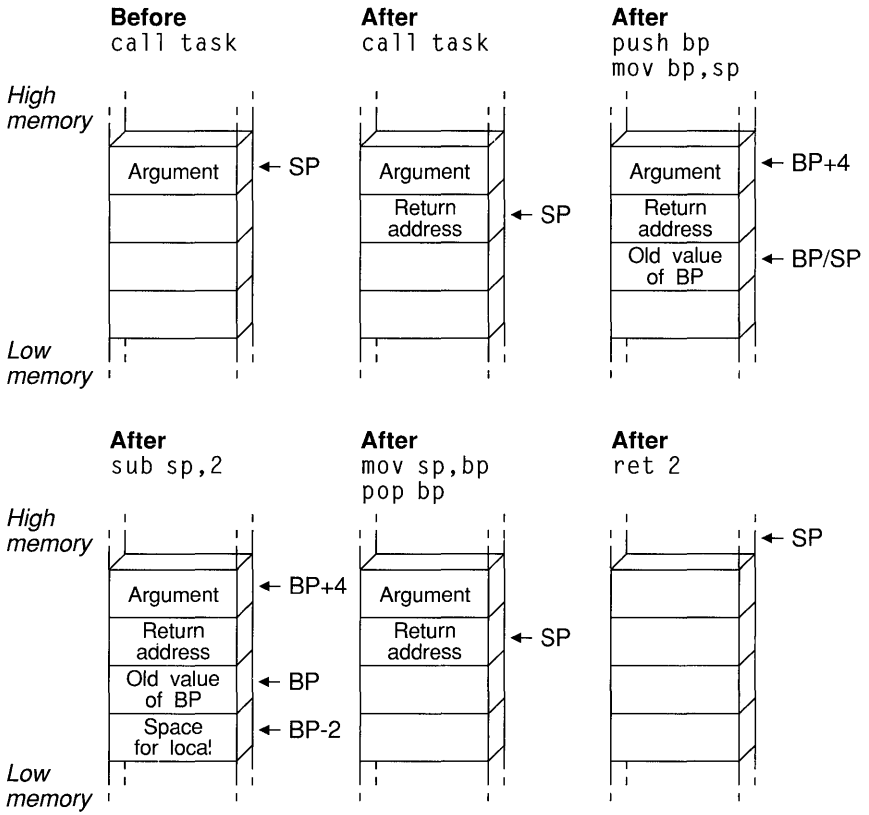


Figure 7.2 Local Variables on the Stack

7.3.5 Creating Local Variables Automatically

Section 7.3.4 described how to create local variables on the stack. This section shows you how to automate the process with the **LOCAL** directive.

The LOCAL directive generates code to set up the stack for local variables.

You can use the **LOCAL** directive to save time and effort when working with local variables. When you use this directive, simply list the variables you want to create, giving a type for each one. The assembler calculates how much space is required on the stack. It also generates instructions to properly decrement SP (as described in the previous section) and to reset SP when you return from the procedure.

When you create local variables this way, your source code can then refer to each local variable by name rather than as an offset of the stack pointer. Moreover, the assembler generates debugging information for each local variable.

The procedure in the previous section can be generated more simply with the following code:

```
task    PROC    NEAR    arg:WORD
        LOCAL   loc:WORD
        .
        .
        .
        mov     loc, 3      ; Initialize local variable
        add     ax, loc     ; Add local variable to AX
        sub     arg, ax     ; Subtract local from argument
        .                   ; Use "loc" and "arg" in other operations
        .
        .
        ret
task    ENDP
```

The **LOCAL** directive must be on the line immediately following the **PROC** statement. It cannot be used after the first instruction in a procedure. The **LOCAL** directive has the following syntax:

LOCAL *vardef* [[, *vardef*]]...

Each *vardef* defines a local variable. A local variable definition has this form:

label[[[*count*]]][[:*qualifiedtype*]]

These are the parameters in local variable definitions:

Argument	Description
label	The name given to the local variable. You can use this name to access the variable.
count	The number of elements of this name and type to allocate on the stack. You can allocate a simple array on the stack with *count*. The brackets around *count* are required. If this field is omitted, one data object is assumed.

Argument	Description
qualifiedtype	A simple MASM type or a type defined with other types and attributes. See Section 1.2.6, "Data Types," for more information.

If the number of local variables exceeds one line, you can place a comma at the end of the first line and continue the list on the next line. Another method is to use several consecutive **LOCAL** directives.

You must initialize local variables.

The assembler does not initialize local variables. Your program must include code to perform any necessary initializations. For example, the following code fragment sets up a local array and initializes it to zero:

```
arraysz EQU     20

aproc   PROC    USES di
        LOCAL   var1[arraysz]:WORD, var2:WORD
        .
        .
        .
; Initialize local array to zero
        push    ss
        pop     es              ; Set ES=SS
        lea     di, var1        ; ES:DI now points to array
        mov     cx, arraysz     ; Load count
        sub     ax, ax
        rep     stosw           ; Store zeros
; Use the array...
        .
        .
        .
        ret
aproc   ENDP
```

Even though you can reference stack variables by name, the assembler treats them as offsets from BP, and they are not visible outside the procedure. In this procedure, `array` is a local variable.

```
index   EQU   10
test    PROC  NEAR
LOCAL   array[index]:WORD
        .
        .
        .
        mov     bx, index
;       mov     array[bx], 5          ; Not legal!
```

The second **MOV** statement may appear to be legal, but since `array` is an offset of BP, this statement is the same as

```
;       mov [bp + bx + arrayoffset], 5   ; Not legal!
```

BP and BX can be added only to SI and DI. This example would be legal, however, if the index value were moved to SI or DI. This type of error in your program can be difficult to find unless you keep in mind that local variables in procedures are offsets of BP.

7.3.6 Declaring Procedure Prototypes

MASM 6.0 provides a new directive, **INVOKE**, to handle many of the details important to procedure calls, such as pushing parameters according to the correct calling conventions. In order to use **INVOKE**, the procedure called must have previously been declared with a **PROC** statement, an **EXTERNDEF** (or **EXTERN**) statement, or a **TYPEDEF**. You can also place a prototype defined with **PROTO** before the **INVOKE** if the procedure type does not appear before the **INVOKE**. Procedure prototypes defined with **PROTO** inform the assembler of types and numbers of arguments so the assembler can check for errors and provide automatic conversions when **INVOKE** calls the procedure.

Place prototypes after data declarations or in a separate include file.

Prototypes in MASM perform the same function as prototypes in the C language and other high-level languages. A procedure prototype includes the procedure name, the types, and (optionally) the names of all parameters the procedure expects. Prototypes are usually placed at the beginning of an assembly program or in a separate include file. They are especially useful for procedures called from other modules and other languages, enabling the assembler to check for unmatched parameters. If you write routines for a library, you may want to put prototypes into an include file for all the procedures used in that library. See Chapter 8, "Sharing Data and Procedures among Modules and Libraries," for more information about using include files.

Declaring procedure prototypes is optional. You can use the **PROC** directive and the **CALL** instruction, as shown in the previous section.

In MASM 6.0, using the **PROTO** directive is one way to define procedure prototypes. The syntax for a prototype definition is the same as for a procedure declaration (see Section 7.3.3, "Declaring Parameters with the PROC Directive"), except that you do not include the list of registers, *prologuearg* list, or the scope of the procedure.

Also, the **PROTO** keyword precedes the *langtype* and *distance* attributes. The attributes (like **C** and **FAR**) are optional, but if not specified, the defaults are based on any **.MODEL** or **OPTION LANGUAGE** statement. The names of the parameters are also optional, but you must list parameter types. A label preceding **:VARARG** is also optional in the prototype but not in the **PROC** statement.

If a **PROTO** and a **PROC** for the same function appear in the same module, they must match in attribute, number of parameters, and parameter types. The easiest way to create prototypes with **PROTO** for your procedures is to write the procedure and then copy the first line (the line that contains the **PROC** keyword) to a location in your program that follows the data declarations. Change **PROC** to **PROTO** and remove the **USES** *reglist*, the *prologuearg* field, and the *visibility* field. It is important that the prototype follow the declarations for any types used in it to avoid any forward references used by the parameters in the prototype.

The prototype defined with **PROTO** statement and the **PROC** statement for two procedures are given below.

```
;  Procedure prototypes

addup     PROTO NEAR C argcount:WORD, arg2:WORD, arg3:WORD

myproc    PROTO FAR C, argcount:WORD, arg2:VARARG

; Procedure declarations

addup     PROC NEAR C, argcount:WORD, arg2:WORD, arg3:WORD

myproc    PROC FAR C PUBLIC <callcount> USES di si,
          argcount:WORD,
          arg2:VARARG
```

When you call a procedure with **INVOKE**, the assembler checks the arguments given by **INVOKE** against the parameters expected by the procedure. If the data types of the arguments do not match, MASM either reports an error or converts the type to the expected type. These conversions are explained in the next section.

7.3.7 Calling Procedures with INVOKE

INVOKE generates a sequence of instructions that push arguments and call a procedure. This helps maintain code if arguments or *langtype* for a procedure is changed. **INVOKE** generates procedure calls and automatically handles the following tasks:

- Converts arguments to the expected types
- Pushes arguments on the stack in the correct order
- Cleans up the stack when the procedure returns

If arguments do not match in number or if the type is not one the assembler can convert, an error results.

If **VARARG** is an option in a procedure, **INVOKE** can pass arguments in addition to those in the parameter list without generating an error or warning. The extra

arguments must be at the end of the **INVOKE** argument list. All other arguments must match in number and type.

The syntax for **INVOKE** is

INVOKE *expression* [[, *arguments*]]

where *expression* can be the procedure's label or an indirect reference to a procedure, and *arguments* can be an expression, a register pair, or an expression preceded with **ADDR**. (The **ADDR** operator is discussed below.)

Procedures that have these procedure prototypes

```
addup   PROTO NEAR C argcount:WORD, arg2:WORD, arg3:WORD

myproc  PROTO FAR C, argcount:WORD, arg2:VARARG
```

and these procedure declarations

```
addup   PROC NEAR C, argcount:WORD, arg2:WORD, arg3:WORD

myproc  PROC FAR C PUBLIC <callcount> USES di si,
        argcount:WORD,
        arg2:VARARG
```

may have **INVOKE** statements that look like this:

```
        INVOKE  addup,   ax, x,  y
        INVOKE  myproc,  bx, cx, 100, 10
```

The assembler can convert some arguments and parameter type combinations so that the correct type can be passed. The signed or unsigned qualities of the arguments in the **INVOKE** statements determine how the assembler converts them to the types expected by the procedure.

The `addup` procedure, for example, expects parameters of type **WORD**, but the arguments passed by **INVOKE** to the `addup` procedure can be any of these types:

- **BYTE**, **SBYTE**, **WORD**, or **SWORD**
- An expression whose type is specified with the **PTR** operator to be one of those types
- An 8-bit or 16-bit register
- An immediate expression in the range –32K to +64K
- A **NEAR PTR**

If the type is smaller than that expected by the procedure, MASM widens the argument to match.

7.3.7.1 Widening Arguments

For **INVOKE** to correctly handle type conversions, you must use the signed data types for any signed assignments. This list shows the cases in which MASM widens an argument to match the type expected by a procedure's parameters.

Type Passed	Type Expected
BYTE, SBYTE	WORD, SWORD, DWORD, SDWORD
WORD, SWORD	DWORD, SDWORD

When possible, MASM widens arguments to match parameter types.

The assembler generates instructions such as **XOR** and **CBW** to perform the conversion. You can see these generated instructions in the listing file by using the /Sg command-line option. The assembler can extend a segment if far data is expected, and it can convert the type given in the list to the types expected. If the assembler cannot convert the type, however, it generates an error.

7.3.7.2 Detecting Errors

When the assembler widens arguments, it may require the use of a register that could overwrite another argument.

For example, if a procedure with the C calling convention is called with this **INVOKE** statement,

```
INVOKE  myprocA, ax, cx, 100, arg
```

where `arg` is a **BYTE** variable and `myproc` expects four arguments of type **WORD**, the assembler widens and then pushes the variable with this code:

```
mov     al, DGROUP:arg
xor     ah, ah
push    ax
```

As a result, the assembler generates code that also uses the AX register and therefore overwrites the first argument passed to the procedure in AX. The assembler generates an error in this case, requiring you to rewrite the **INVOKE** statement for this procedure.

The **INVOKE** directive uses as few registers as possible. However, widening arguments or pushing constants on the 8088 and 8086 requires the use of the AX register, and sometimes the DX register or the EAX and EDX on the 80386/486. This means that the content of AL, AH, AX, and EAX must frequently be overwritten, so you should avoid using these registers to pass arguments. As an alternative you can use DL, DH, DX, and EDX, since these registers are rarely used.

7.3.7.3 Invoking Far Addresses

You can pass a **FAR** pointer in a *segment*::*offset* pair, as shown below. Note the use of double colons to separate the register pair. The registers could be any other register pair, including a pair that a DOS call uses to return values.

```
FPWORD   TYPEDEF FAR PTR WORD
SomeProc PROTO var1:DWORD, var2:WORD, var3:WORD

        pfaritem    FPWORD     faritem
        .
        .
        .
        les         bx, pfaritem
        INVOKE      SomeProc, ES::BX, arg1, arg2
```

However, you cannot give **INVOKE** two arguments, one for the segment and one for the offset, and have **INVOKE** combine the two for an address.

7.3.7.4 Passing an Address

You can use the **ADDR** operator to pass the address of an expression to a procedure that is expecting a **NEAR** or **FAR** pointer. This example generates code to pass a far pointer (to `arg1`) to the procedure `proc1`.

```
PBYTE   TYPEDEF FAR PTR BYTE
arg1    BYTE    "This is a string"
proc1   PROTO   NEAR C fparg:PBYTE
        .
        .
        .
INVOKE  proc1, ADDR arg1
```

See Section 3.3.1 for information on defining pointers with **TYPEDEF**.

7.3.7.5 Invoking Procedures Indirectly

You can make an indirect procedure call such as `call [bx + si]` by using a pointer to a function prototype with **TYPEDEF**, as shown in this example:

```
FUNCPROTO       TYPEDEF PROTO NEAR ARG1:WORD, ARG2:WORD
FUNCPTR         TYPEDEF PTR FUNCPROTO

        .DATA
pfunc   FUNCPTR OFFSET proc1, OFFSET proc2

        .CODE
        mov     si, Num             ; Num contains 0 or 2
        INVOKE  FUNCPTR PTR [si]    ; Selects proc1 or proc2
```

You can also use **ASSUME** to accomplish the same task. The **ASSUME** statement associates the type PFUNC with the BX register.

```
ASSUME  BX:FUNCPTR
mov     si, Num
INVOKE  FUNCPTR PTR [bx+si]
```

7.3.7.6 Checking the Code Generated

The **INVOKE** directive generates code that may vary depending on the processor mode and calling conventions in effect. You can check your listing files to see the code generated by the **INVOKE** directive if you use the /Sg command-line option.

7.3.8 Generating Prologue and Epilogue Code

When you use the **PROC** directive with its extended syntax and argument list, the assembler automatically generates the prologue and epilogue code in your procedure. "Prologue code" is generated at the start of the procedure; it sets up a stack pointer so you can access parameters from within the procedure. It also saves space on the stack for local variables, initializes registers such as DS, and pushes registers that the procedure uses. Similarly, "epilogue code" is the code at the end of the procedure that pops registers and returns from the procedure.

The assembler automatically generates the prologue code when it encounters the first instruction after the **PROC** directive. It generates the epilogue code when it encounters a **RET** or **IRET** instruction. Using the assembler-generated prologue and epilogue code saves you time and decreases the number of repetitive lines of code in your procedures.

The generated prologue or epilogue code depends on the

- Local variables defined
- Arguments passed to the procedure
- Current processor selected (affects epilogue code only)
- Current calling convention
- Options passed in the *prologuearg* of the **PROC** directive
- Registers being saved

The *prologuearg* list contains options specifying how the prologue or epilogue code should be generated. The next section explains how to use these options, gives the standard prologue and epilogue code, and explains the techniques for defining your own prologue and epilogue code.

7.3.8.1 Using Automatic Prologue and Epilogue Code

The standard prologue and epilogue code handles parameters and local variables. If a procedure does not have any parameters or local variables, the prologue and epilogue code that sets up and restores a stack pointer is omitted, unless **FORCEFRAME** is included in the *prologuearg* list. (**FORCEFRAME** is discussed later in this section.) Prologue and epilogue code also generates a push and pop for each register in the register list unless the register list is empty.

RETN and RETF suppress epilogue code generation.

When a **RET** is used without an operand, the assembler generates the standard epilogue code. If you do not want the standard epilogue generated, you can use **RETN** or **RETF** with or without operands. **RET** with an integer operand does not generate epilogue code, but it does generate the right size of return.

In the examples below showing standard prologue and epilogue code, `localbytes` is a variable name used in this example to represent the number of bytes needed on the stack for the locals declared, `parmbytes` represents the number of bytes that the parameters take on the stack, and `registers` represents the list of registers to be pushed or popped.

The standard prologue code is the same in any processor mode:

```
push bp
mov bp, sp
sub sp, localbytes  ; if localbytes is not 0
push registers
```

The standard epilogue code is:

```
pop registers
mov sp, bp    ; if localbytes is not 0
pop bp
ret parmbytes ; use parmbytes only if lang is not C
```

The standard prologue and epilogue code recognizes two operands passed in the *prologuearg* list, **LOADDS** and **FORCEFRAME**. These operands modify the prologue code. Specifying **LOADDS** saves and initializes DS. Specifying **FORCEFRAME** as an argument generates a stack frame even if no arguments are sent to the procedure and no local variables are declared. If your procedure has any parameters or locals, you do not need to specify **FORCEFRAME**.

Specifying **LOADDS** generates this prologue code:

```
push bp
mov bp, sp
sub sp, localbytes  ; if localbytes is not 0
push ds
mov ax, DGROUP
mov ds, ax
push registers
```

Specifying **LOADDS** generates the following epilogue code:

```
pop registers
pop ds
mov sp, bp
pop bp
ret parmbytes ; use parmbytes only if lang is not C
```

7.3.8.2 User-Defined Prologue and Epilogue Code

If you want a different set of instructions for prologue and epilogue code in your procedures, you can write macros that are executed instead of the standard prologue and epilogue code. For example, while you are debugging your procedures, you may want to include a stack check or track the number of times a procedure is called. You can write your own prologue code to do these things whenever a procedure executes. Different prologue code may also be necessary if you are writing applications for Microsoft Windows or any other environment application for DOS. User-defined prologue macros will respond correctly if you specify **FORCEFRAME** in the *prologuearg* of a procedure.

To write your own prologue or epilogue code, the **OPTION** directive must appear in your program. It disables automatic prologue and epilogue code generation. When you specify

OPTION PROLOGUE : *macroname*

OPTION EPILOGUE : *macroname*

the assembler calls the macro specified in the **OPTION** directive instead of generating the standard prologue and epilogue code. The prologue macro must be a macro function, and the epilogue macro must be a macro procedure.

The assembler expects your prologue or epilogue macro to have this form:

macroname **MACRO** *procname*, /
flag, /
parmbytes, /
localbytes, /
<reglist>, /
userparms

The following list explains the arguments passed to your macro. Your macro must have formal parameters to match all the actual arguments passed.

Argument	Description
procname	The name of the procedure.
flag	A 16-bit flag containing the following information:

Bit = Value	Description
Bit 0, 1, 2	For calling conventions (000=unspecified language type, 001=**C**, 010=**SYSCALL**, 011=**STDCALL**, 100=**PASCAL**, 101=**FORTRAN**, 110=**BASIC**)
Bit 3	Undefined (not necessarily zero)
Bit 4	Set if the caller restores the stack (Use **RET**, not **RET***n*)
Bit 5	Set if procedure is **FAR**
Bit 6	Set if procedure is **PRIVATE**
Bit 7	Set if procedure is **EXPORT**
Bit 8	Set if the epilogue was generated as a result of an **IRET** instruction and cleared if the epilogue was generated as a result of a **RET** instruction
Bits 9–15	Undefined (not necessarily zero)

parmbytes	The byte count of all the parameters given in the **PROC** statement.
localbytes	The count in bytes of all locals defined with the **LOCAL** directive.
reglist	A list of the registers following the **USES** operator in the procedure declaration. This list is enclosed by angle brackets (< >), and each item is separated by commas. This list is reversed for epilogues.
userparms	Any argument you want to pass to the macro. The prologuearg (if there is one) specified in the **PROC** directive is passed to this argument.

Your macro function must return the *parmbytes* parameter. However, if the prologue places other values on the stack after pushing BP and these values are not referenced by any of the local variables, the exit value must be the number of

bytes for procedure locals plus any space between BP and the locals. Therefore *parmbytes* is not always equal to the bytes occupied by the locals.

The following macro is an example of a user-defined prologue that counts the number of times a procedure is called.

```
ProfilePro      MACRO procname,       \
                      flag,          \
                      bytecount,     \
                      numlocals,     \
                      regs,          \
                      macroargs

                .DATA
procname&count  WORD 0
                .CODE
        inc     procname&count  ; Accumulates count of times the
                                ;  procedure is called
        push    bp
        mov     bp, sp
                                ; Other BP operations
        IFNB <regs>
           FOR r, regs
              push r
           ENDM
        ENDIF
        EXITM %bytecount
ENDM
```

Your program must also include this statement before any procedures are called that use the prologue:

```
OPTION PROLOGUE:ProfilePro
```

If you define only a prologue or an epilogue macro, the standard prologue or epilogue code is used for the one you do not define. The form of the code generated depends on the **.MODEL** and **PROC** options used.

If you want to revert to the standard prologue or epilogue code, use `PROLOGUEDEF` or `EPILOGUEDEF` as the *macroname* in the **OPTION** statement.

```
OPTION EPILOGUE:EPILOGUEDEF
```

You can completely suppress prologue or epilogue generation with

```
OPTION PROLOGUE:None
OPTION EPILOGUE:None
```

In this case, no user-defined macro is called, and the assembler does not generate a default code sequence. This state remains in effect until the next **OPTION PROLOGUE** or **OPTION EPILOGUE** is encountered.

See Chapter 9 for additional information about writing macros. The PROLOGUE.INC file provided in the MASM 6.0 distribution disks can be used to create the prologue and epilogue sequences for the Microsoft C Professional Development System, version 6.0.

7.4 DOS Interrupts

In addition to jumps, loops, and procedures that alter program execution, interrupt routines transfer execution to a different location. In this case, control goes to an interrupt routine.

You can write your own interrupt routines, either to replace an existing routine or to use an undefined interrupt number. You may want to replace the processor's divide-overflow (0h) interrupts or DOS interrupts, such as the critical-error (24h) and CONTROL+C (23h) handlers. The **BOUND** instruction checks array bounds and calls interrupt 5 when an error occurs. If you use this instruction, you need to write an interrupt handler for it.

This section summarizes the following:

- How to call interrupts
- How the processor handles interrupts
- How to redefine an existing interrupt routine

The example routine in this section handles addition or multiplication overflow and illustrates the steps necessary for writing an interrupt routine. See Chapter 19, "Writing Memory-Resident Software" for additional information about DOS and BIOS interrupts.

NOTE Under OS/2, system access is made through calls to the Applications Program Interface (API), not through interrupts. Microsoft Windows applications use both interrupts and API calls.

7.4.1 Calling DOS and ROM-BIOS Interrupts

Interrupts are the only way to access DOS from assembly language. They are called with the **INT** instruction, which takes one operand—an immediate value between 0 and 255.

When calling DOS and ROM-BIOS interrupts, you usually need to place a function number in the AH register. You can use other registers to pass arguments to functions. Some interrupts and functions return values in certain registers, although register use varies for each interrupt. This code writes the text of `msg` to the screen.

```
        .DATA
msg     BYTE    "This writes to the screen",$
        .CODE
        mov     dx, offset msg
        mov     ah, 09h
        int     21h
```

When the **INT** instruction executes, the processor takes the following six steps:

1. Looks up the address of the interrupt routine in the interrupt descriptor table (also called the "interrupt vector"). This table starts at the lowest point in memory (segment 0, offset 0) and consists of four bytes (two segment and two offset) for each interrupt. Thus, the address of an interrupt routine equals the number of the interrupt multiplied by 4.
2. Clears the trap flag (TF) and interrupt enable flag (IF).
3. Pushes the flags register, the current code segment (CS), and the current instruction pointer (IP).
4. Jumps to the address of the interrupt routine, as specified in the interrupt descriptor table.
5. Executes the code of the interrupt routine until it encounters an **IRET** instruction.
6. Pops the instruction pointer, code segment, and flags.

Figure 7.3 illustrates how interrupts work.

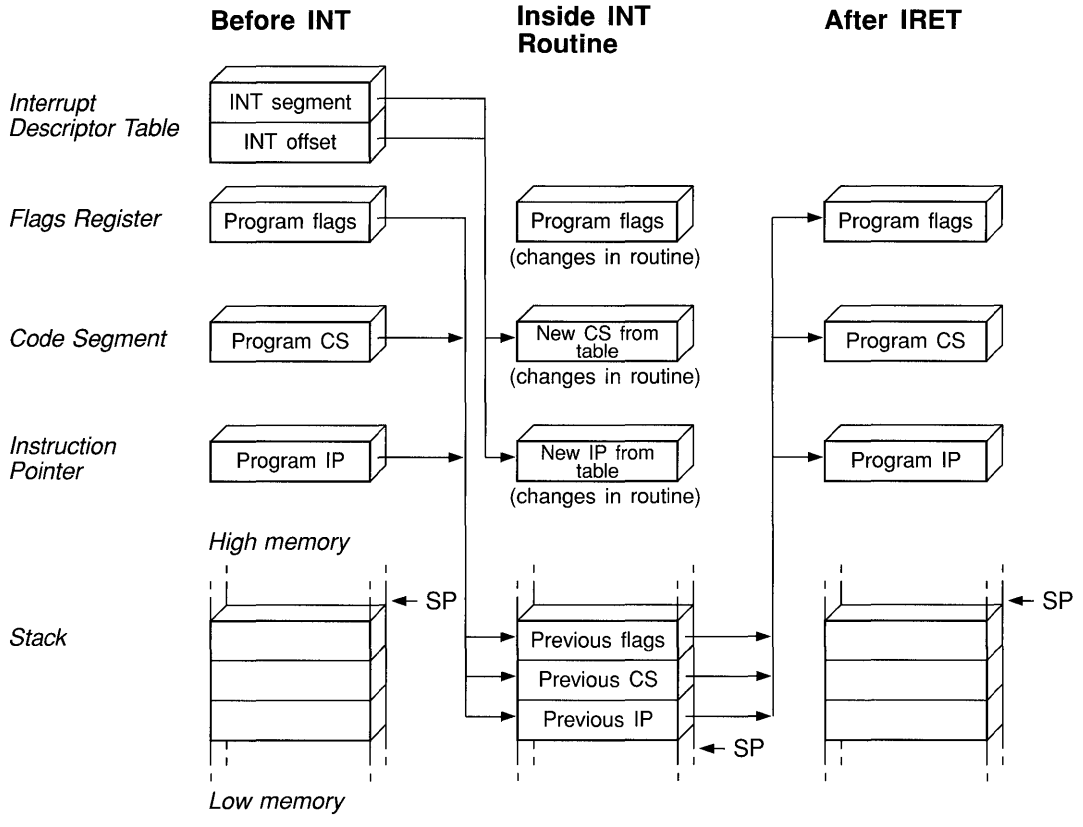


Figure 7.3 Operation of Interrupts

Some DOS interrupts should not normally be called. Some (such as 20h and 27h) have been replaced by other DOS interrupts. Others are used internally by DOS.

7.4.2 Replacing or Redefining Interrupt Routines

One interrupt routine you may want to redefine is the routine called by **INTO**. The **INTO** (Interrupt on Overflow) instruction is a variation of the **INT** instruction. It calls interrupt 04h when the overflow flag is set. By default, the routine for interrupt 4 simply consists of an **IRET**, so it returns without doing anything. Using **INTO** is an alternative to using **JO** (Jump on Overflow) to jump to an overflow routine.

To replace or redefine an existing interrupt, your routine must

- Replace the address in the interrupt descriptor table with the address of your new routine and save the old address
- Provide new instructions to handle the interrupt
- Restore the old address when your routine ends

An interrupt routine can be written like a procedure by using the **PROC** and **ENDP** directives. The routine should always be defined as **FAR** and should end with an **IRET** instruction instead of a **RET** instruction.

NOTE Since the assembler doesn't know whether you are going to terminate with **RET** or **IRET**, you can use the full extended **PROC** syntax (described in Section 7.3.3, "Declaring Parameters with the PROC Directive") to write interrupt procedures. However, you should not make interrupt procedures **NEAR** or specify arguments for them. You can use the **USES** keyword, however, to correctly generate code to save and to restore a register list in interrupt procedures.

The **STI** (Set Interrupt Flag) and **CLI** (Clear Interrupt Flag) instructions turn interrupts on or off. You can use **CLI** to turn off interrupt processing so that an important routine cannot be stopped by a hardware interrupt. After the routine has finished, use **STI** to turn interrupt processing back on. Interrupts received while interrupt processing was turned off by **CLI** are saved and executed when **STI** turns interrupts back on.

MASM 6.0 provides two new forms of the **IRET** instruction that suppress epilogue sequences. This allows an interrupt to have local variables or use a user-defined prologue. **IRETF** pops a **FAR16** return address, and **IRETFD** pops a **FAR32** return address.

The following example uses DOS functions to save the address of the initial interrupt routine in a variable and to put the address of the new interrupt routine in the interrupt descriptor table. Once the new address has been set, the new routine is called any time the interrupt is called. This new routine prints a message and sets AX and DX to 0.

To replace the address in the interrupt descriptor table with the address of your procedure, AL needs to be loaded with 04h and AH loaded with 35, the Get Interrupt Vector function. The Set Interrupt Vector function requires 25 in AH.

Follow this example to replace an existing interrupt routine. To write an interrupt handler for an unused interrupt, see online help for available vectors.

```
        .MODEL LARGE, C, DOS
FPFUNC  TYPEDEF FAR PTR
        .DATA
msg     BYTE    "Overflow - result set to 0",13,10,"$"
vector  FPFUNC  ?
        .CODE
        .STARTUP

        mov     ax, 3504h       ; Load interrupt 4 and call DOS
        int     21h             ;  Get Interrupt Vector function
        mov     WORD PTR vector[2],es ; Save segment
        mov     WORD PTR vector[0],bx ;  and offset

        push    ds              ; Save DS
        mov     ax, cs          ; Load segment of new routine
        mov     ds, ax
        mov     dx, OFFSET ovrflow   ; Load offset of new routine
        mov     ax, 2504h       ; Load interrupt 4 and call DOS
        int     21h             ;  Set Interrupt Vector function
        pop     ds              ; Restore
        .
        .
        .
        add     ax, bx          ; Do addition (or multiplication)
        into                    ; Call interrupt 4 if overflow
        .
        .
        .
        lds     dx, vector      ; Load original interrupt address
        mov     ax, 2504h       ; Restore interrupt number 4
        int     21h             ;  with DOS set vector function
        mov     ax, 4C00h       ; Terminate function
        int     21h

ovrflow         PROC    FAR
                sti             ; Enable interrupts
                                ;  (turned off by INT)
                mov     ah, 09h ; Display string function
                mov     dx, OFFSET msg     ; Load address
                int     21h     ; Call DOS
                sub     ax, ax  ; Set AX to 0
                sub     dx, dx  ; Set DX to 0
                iret            ; Return
ovrflow         ENDP
                END
```

Before your program ends, you should restore the original address by loading DX with the original interrupt address and using the DOS set vector function to store the original address at the correct location.

7.5 Related Topics in Online Help

Other information available online which relates to topics in this chapter is given in the list below:

Topic	Access
OPTION directive	From the "MASM 6.0 Contents" screen, choose "Directives," then choose "Miscellaneous"
DOS and ROM-BIOS interrupts	From the list of System Resources on the "MASM 6.0 Contents" screen, choose "DOS Calls" or "BIOS Calls"
BT, BTC, BTR, BTS	From the "MASM 6.0 Contents" screen, choose "Processor Instructions" and then "Logical and Shifts"
Other forms of the **LOOP** instruction	From the "MASM 6.0 Contents" screen, choose "Processor Instructions" and then "Control Flow"
Processor Flag Summary	From the "MASM 6.0 Contents" screen, choose "Processor Instructions"

Chapter 8

Sharing Data and Procedures among Modules and Libraries

To use symbols and procedures in more than one module, the assembler must be able to recognize the shared data as global to all the modules where they are used. MASM 6.0 provides new techniques to simplify data-sharing and give a high-level interface to multiple-module programming. With these techniques, you can place shared symbols in include files. This makes the data declarations in the file available to all modules that use the include file.

After an overview of the data-sharing methods, the next section of this chapter focuses on organizing modules and using the include file to simplify data-sharing. The first method allows you to create a single include file that works in the modules where the symbol is used as well as where it is defined.

Sharing procedures and data items using the **PUBLIC** and **EXTERN** directives in the appropriate modules is the other method of data-sharing. The third section of this chapter explains how to use **PUBLIC** and **EXTERN**.

You may also want to place commonly used routines in libraries. Section 8.4 explains how to create program libraries and access their routines.

8.1 Selecting Data-Sharing Methods

If data defined in one module is to be used in the other modules of a multiple-module program, the data must be made public and external. MASM provides several methods for doing this.

One method is to declare a symbol public (with the **PUBLIC** directive) in the module where it is defined. This makes the symbol available to other modules. Then place an **EXTERN** statement for that symbol in the rest of the modules that use the public symbol. This statement informs the assembler that the symbol is external—defined in another module.

As an alternative, you can use the **COMM** directive instead of **PUBLIC** and **EXTERN**. However, communal variables have some limitations. You cannot depend on their location in memory because they are allocated by the linker, and they cannot be initialized.

These two data-sharing methods are still available, but MASM 6.0 introduces a new directive, **EXTERNDEF**, that declares a symbol either public or external, as

appropriate. **EXTERNDEF** simplifies the declarations for global (public and external) variables and encourages the use of include files.

The next section provides further details on using include files. Section 8.3, "Using Alternatives to Include Files," provides more information on **PUBLIC** and **EXTERN**.

8.2 Sharing Symbols with Include Files

Place statements common to all modules in include files.

Include files can contain any valid MASM statement but typically consist of type and symbol declarations. The assembler inserts the contents of the include file into a module at the location of the **INCLUDE** directive. Include files can simplify project organization by eliminating the need to physically insert common declarations into more than one program or module. Include files are always optional. See Section 8.3 for alternatives to using include files.

The first part of this section explains how to organize symbol definitions and the declarations that make the symbols global (available to all modules). It then shows how to make both variables and procedures public with **EXTERNDEF**, **PROTO**, and **COMM**. The last part of this section tells where to place these directives in the modules and include files.

8.2.1 Organizing Modules

This section summarizes the organization of declarations and definitions in modules and include files and the use of the **INCLUDE** directive.

Include Files Type declarations that need to be identical in every module should be placed in an include file. Doing so ensures consistency and can save programming time when updating programs. Include files should contain only symbol declarations and any other declarations that are resolved at assembly time. (See Section 1.3.1, "Generating and Running Executable Programs," for a list of assembly-time operations.) If the include file is associated with more than one module, it cannot contain statements that define and allocate memory for symbols unless you include the data conditionally (see Section 1.3.3).

Modules Label definitions that cause the assembler to allocate memory space must be defined in a module, not in an include file. If any of these definitions is located in the include file, it is copied into each file that uses the include file, creating an error.

Include files are inserted at the location of the INCLUDE directive.

Once you have placed public symbols in an include file, you need to associate that file with the main module. The **INCLUDE** statement is usually placed before data and code segments in your modules. When the assembler encounters an **INCLUDE** directive, it opens the specified file and assembles all its statements.

The assembler then returns to the original file and continues the assembly process.

The **INCLUDE** directive takes the form

INCLUDE *filename*

where *filename* is the full name or fully specified path of the include file. For example, the following declaration inserts the contents of the include file SCREEN.INC in your program:

```
INCLUDE SCREEN.INC
```

You must make sure that the assembler can find include files.

The file name in the **INCLUDE** directive must be fully specified; no extensions are assumed. If a full path name is not given, the assembler searches first in the directory of the source file containing the **INCLUDE** directive.

If the include file is not in the source file directory, the assembler searches the paths specified in the assembler's command-line option /I, or in PWB's Include Paths field in the MASM Option dialog box (accessed from the Option menu). The /I option takes this form:

/I *path*

Multiple /I options can be used to specify that multiple directives be searched in the order they appear on the command line. If none of these directories contains the desired include file, the assembler finally searches in the paths specified in the INCLUDE environment variable. If the include file still cannot be found, an assembly error occurs. The related /x option tells the assembler to ignore the INCLUDE environment variable for all subsequent assemblies.

An include file may specify another include file. The assembler processes the second include file before returning to the first. Include files can be nested this way as deeply as desired; the only limit is the amount of free memory.

Put constants used in more than one module into the include file.

Include Files or Modules You can use the **EQU** directive to create named constants that cannot be redefined in your program (see Section 1.2.4, "Integer Constants and Constant Expressions," for information about the **EQU** directive). Placing a constant defined with **EQU** in an include file makes it available to all modules that use that include file.

Placing **TYPEDEF**, **STRUCT**, **UNION**, and **RECORD** definitions in an include file guarantees consistency in type definitions. If required, the variable instances derived from these definitions can be made public among the modules with **EXTERNDEF** declarations (see the next section). Macros (including macros defined with **TEXTEQU**) must be placed in include files to make them visible in other modules.

If you elect to use full segment definitions (along with, or instead of, simplified definitions), you can force a consistent segment order in all files by defining segments in an include file. This technique is explained in Section 2.3.2, "Controlling the Segment Order."

8.2.2 Declaring Symbols Public and External

It is sometimes useful to make procedures and variables (such as large arrays or status flags) global to all program modules. Global variables are freely accessible within all routines; you do not have to explicitly pass them to the routines that need them.

Variables can be made global to multiple modules in several ways. This section describes three ways to make them global by using the **EXTERNDEF**, **PROTO**, or **COMM** declarations within include files. Section 8.3.1 explains how to use the **PUBLIC** and **EXTERN** directives within modules.

External identifiers must be unique.

These methods make symbols global to the modules in which they are used. Therefore, symbols must be unique. The linker enforces this requirement.

8.2.2.1 Using EXTERNDEF

EXTERNDEF can appear in the defining or calling modules.

MASM treats **EXTERNDEF** as a public declaration in the defining module and as an external declaration in accessing module(s). You can use the **EXTERNDEF** statement in your include file to make a variable common among two or more modules. **EXTERNDEF** works with all types of variables, including arrays, structures, unions, and records. It also works with procedures.

As a result, a single include file can contain an **EXTERNDEF** declaration that works in both the defining module and any accessing module. It is ignored in modules that neither define nor access the variable. Therefore, an include file for a library which is used in multiple .EXE files does not force the definition of a symbol as **EXTERN** does.

The **EXTERNDEF** statement takes this form:

EXTERNDEF [[*langtype*]] *name*:*qualifiedtype*

The *name* is the variable's identifier. The *qualifiedtype* is explained in detail in Section 1.2.6, "Data Types."

The optional *langtype* specifier sets the naming conventions for the *name* it precedes. It overrides any language specified in the **.MODEL** directive. The specifier can be **C**, **SYSCALL**, **STDCALL**, **PASCAL**, **FORTRAN**, or **BASIC**. See Section 20.1, "Naming and Calling Conventions," for information on selecting the appropriate *langtype* type.

The diagram below shows the statements that declare an array, make it public, and use it in another module.

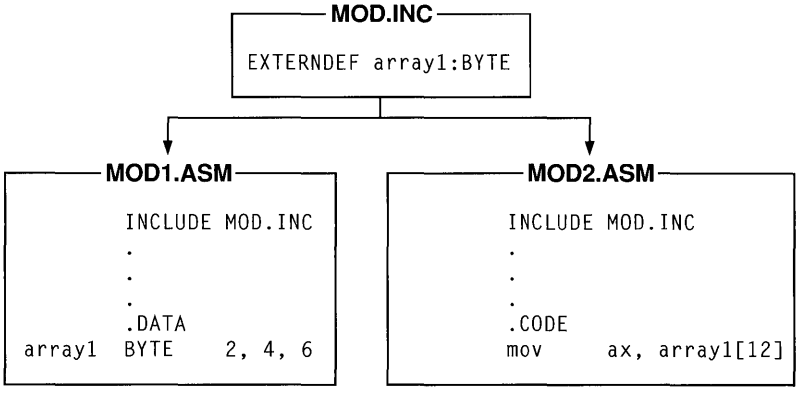


Figure 8.1 Using EXTERNDEF for Variables

The file position of **EXTERNDEF** directives is important. See Section 8.2.3, "Positioning External Declarations," for more information.

The assembler does not check parameters when you call EXTERNDEF procedures.

You can also make procedures visible by using **EXTERNDEF** without **PROTO** inside an include file. This method treats the procedure name as a simple identifier, without the parameter list, so you forgo the assembler's ability to check for the correct parameters during assembly.

The method for using **EXTERNDEF** for procedures is the same as using it with variables. You can also use **EXTERNDEF** to make code labels global.

8.2.2.2 Using PROTO

When a procedure is defined in one module and called from another module, it must be declared public in the defining module and external in the calling modules; otherwise, assembly or linking errors occur.

You have three methods for declaring a procedure public. Using **PUBLIC** and **EXTERN** is the only method prior to MASM 6.0. Section 8.3.1 explains the use of **PUBLIC** and **EXTERN**. The previous section (8.2.2.1) explains the use of **EXTERNDEF**. This section illustrates the use of **PROTO**.

A **PROTO** (prototype) declaration in the include file establishes a procedure's interface in both the defining and calling modules. The **PROTO** directive automatically generates an **EXTERNDEF** for the procedure unless the procedure has been declared **PRIVATE** in the **PROC** statement. Defining a prototype enables type-checking for the procedure arguments.

PROTO and INVOKE simplify procedure calls.

Follow these steps to create an interface for a procedure defined in one module and called from other modules:

1. Place the **PROTO** declaration in the include file.
2. Define the procedure with **PROC**. The **PROC** directive declares the procedure **PUBLIC** by default.
3. Call the procedure with the **INVOKE** statement (or with **CALL**).

The following example is a **PROTO** declaration for the far procedure `CopyFile`, which uses the C parameter-passing and naming conventions, and takes the arguments `filename` and `numberlines`. The diagram following the example shows the file placement for these statements. This definition goes into the include file:

```
CopyFile PROTO FAR C filename:BYTE, numberlines:WORD
```

The procedure definition for `CopyFile` is

```
CopyFile PROC FAR C USES cx, filename:BYTE, numberlines:WORD
```

To call the `CopyFile` procedure, you can use this **INVOKE** statement:

```
        INVOKE   CopyFile, NameVar, 200
```

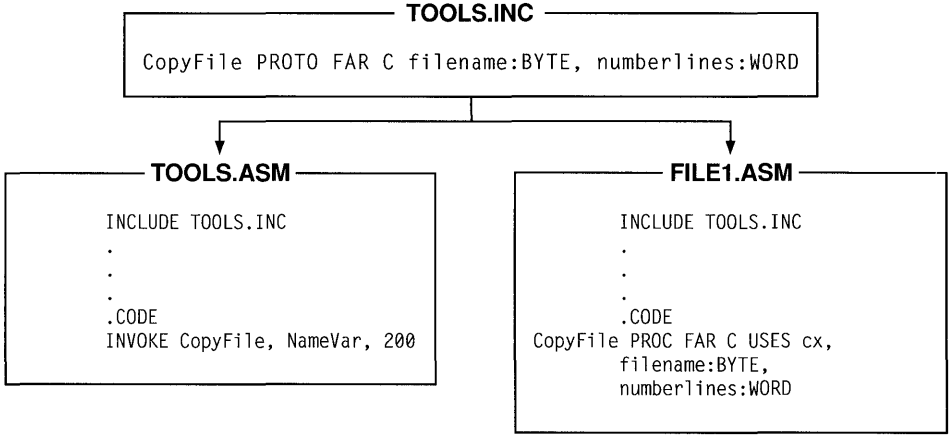


Figure 8.2 Using PROTO and INVOKE

See Chapter 7, "Controlling Program Flow," for descriptions, syntax, and examples of **PROTO**, **PROC**, and **INVOKE**.

8.2.2.3 Using COMM

Another way to share variables among modules is to add the **COMM** (communal) declaration to your include file. Since communal variables are allocated by the linker and cannot be initialized, you cannot depend on their location or sequence.

Communal variables are supported by MASM primarily for compatibility with communal variables in Microsoft C. Communal variables are not used in any other Microsoft language, and they are not compatible with C++ and some other languages.

Communal variables can reduce the size of executable files.

COMM declares a variable external but cannot be used with code. **COMM** also instructs the linker to define the variable if it has not been explicitly defined in a module. The memory space for communal variables may not be assigned until load time, so using communal variables may reduce the size of your executable file.

The **COMM** declaration has the syntax

COMM [[*langtype*]] [[**NEAR** | **FAR**]] *label*:*type*[[:*count*]]

The *label* is the name of the variable. The *langtype* sets the naming conventions for the name it precedes. It overrides any language specified in the **.MODEL** directive.

If **NEAR** or **FAR** is not specified, the variable determines the default from the current memory model (**NEAR** for **TINY**, **SMALL**, **COMPACT**, and **FLAT**; **FAR** for **MEDIUM**, **LARGE**, and **HUGE**).

The *type* can be a constant expression, but it is usually a type such as **BYTE**, **WORD**, or **DWORD**, or a structure, union, or record. If you first declare the *type* with **TYPEDEF**, CodeView can provide type information. The *count* is the number of elements. If no *count* is given, one element is assumed.

The following example creates the common far variable `DataBlock`, which is a 1,024-element array of uninitialized signed doublewords:

```
COMM FAR DataBlock:SDWORD:1024
```

NOTE C variables declared outside functions (except static variables) are communal unless explicitly initialized; they are the same as assembly-language communal variables. If you are writing assembly-language modules for C, you can declare the same communal variables in both C and MASM include files. However, communal variables in C do not have to be declared communal in assembler. The linker will match the **EXTERN**, **PUBLIC**, and **COMM** statements for the variable.

EXTERNDEF is a flexible alternative to using COMM.

EXTERNDEF (explained in the previous section) is more flexible than **COMM** because you can initialize variables defined with it, and you can use those variables in code that depends on the position and sequence of the data.

8.2.3 Positioning External Declarations

Although LINK determines the actual address of an external symbol, the assembler assumes a default segment for the symbol, based on the location of the external directive in the source code. You should therefore position **EXTERN** and **EXTERNDEF** directives according to these rules:

- If you know which segment defines an external symbol, put the **EXTERN** statement in that segment.
- If you know the group but not the segment, position the **EXTERN** statement outside any segment and reference the variable with the group name. For example, if `var1` is in DGROUP, you would reference the variable as

  ```
  mov DGROUP:var1, 10.
  ```

- If you know nothing about the location of an external variable, put the **EXTERN** statement outside any segment. You can use the **SEG** directive to access the external variable like this:

  ```
  mov ax, SEG var1
  mov es, ax
  mov ax, es:var1
  ```

- If the symbol is an absolute symbol or a far code label, you can declare it external anywhere in the source code.

Always close opened segments.

Any segments opened in include files should always be closed so that external declarations following an include statement are not incorrectly placed inside a segment. Any include statements in your program should immediately follow the **.MODEL**, **OPTION**, and processor directives.

For the same reason, if you want to be certain that an external definition is outside a segment, you can use **@CurSeg**. The **@CurSeg** predefined symbol returns a blank if the definition is not in a segment. For example,

```
        .DATA
        .
        .
        .
@CurSeg ENDS                    ; Close segment
        EXTERNDEF var:WORD
```

See Section 1.2.3, "Predefined Symbols," for information about predefined symbols such as **@CurSeg**.

8.3 Using Alternatives to Include Files

If your project uses only two modules (or if it is written with a version of MASM prior to 6.0), you may want to continue using **PUBLIC** in the defining module and **EXTERN** in the accessing module, and not create an include file for the project. The **EXTERN** directive can be used in an include file, but the include file containing **EXTERN** cannot be added to the module that contains the corresponding **PUBLIC** directive for that symbol. This section assumes that you are not using include files.

8.3.1 PUBLIC and EXTERN

The **PUBLIC** and **EXTERN** directives are less flexible than **EXTERNDEF** and **PROTO** because they are module-specific: **PUBLIC** must appear in the defining module and **EXTERN** must appear in the calling modules. This section shows how to use **PUBLIC** and **EXTERN**. Information on where to place the external declarations in your file is in Section 8.2.3, "Positioning External Declarations."

The **PUBLIC** directive makes a name visible outside the module in which it is defined. This gives other program modules access to that identifier.

The **EXTERN** directive performs the complementary function. It tells the assembler that a name referenced within a particular module is actually defined and declared public in another module that will be specified at link time.

A **PUBLIC** directive can appear anywhere in a file. Its syntax is

PUBLIC [[*langtype*]] *name*[[**,** [[*langtype*]] *name*]] ...

The *name* must be the name of an identifier defined within the current source file. Only code labels, data labels, procedures, and numeric equates can be declared public.

If you specify the *langtype* field here, it overrides the language specified by **.MODEL**. The *langtype* field can be **C**, **SYSCALL**, **STDCALL**, **PASCAL**, **FORTRAN**, or **BASIC**. Section 7.3.3, "Declaring Parameters with the PROC Directive," and Section 20.1, "Naming and Calling Conventions," provide more information on specifying *langtype* types.

The **EXTERN** directive tells the assembler that an identifier is external—defined in some other module that will be supplied at link time. Its syntax is

EXTERN [[*langtype*]] *name*:{**ABS** | *qualifiedtype*}

Section 1.2.6, "Data Types," describes *qualifiedtype*. The **ABS** (absolute) keyword can be used only with external numeric constants. **ABS** causes the identifier to be imported as a relocatable unsized constant. This identifier can then be used anywhere a constant can be used. If the identifier is not found in another module at link time, the linker generates an error.

In the following example, the procedure `BuildTable` and the variable `Var` are declared public. The procedure uses the Pascal naming and data-passing conventions:

MOD1.ASM

```
        .MODEL small, Pascal
PUBLIC  BuildTable, Var
        .
        .
        .
        .DATA
Var     BYTE    0
        .
        .
        .
        .CODE
BuildTable PROC USES cx dx,
        sizevar:WORD
        .
        .
        .
        ret
BuildTable ENDP
```

MOD2.ASM

```
EXTERN  Var:BYTE,
        BuildTable:FAR
        .
        .
        .
        mov     al, Var
        call    BuildTable
```

Figure 8.3 Using PUBLIC and EXTERN

8.3.2 Other Alternatives

You can also use the directives discussed earlier (**EXTERNDEF**, **PROTO**, and **COMM**) without the include file. In this case, place the declarations to make a symbol global in the same module where the symbol is defined. You might want to use this technique if you are linking only a few modules that have very little data in common.

8.4 Developing Libraries

As you create reusable procedures, you can place them in a library file for convenient access. Although you can put any routine into a library, each library usually contains related routines. For example, you might place string-manipulation functions in one library, matrix calculations in another, and port communications in another.

A library consists of combined object modules, each created from a single source file. The object module is the smallest independent unit in a library. If you link with one symbol in a module, you get the entire module, but not the entire library.

A library can consist of two files—an include file containing necessary declarations and constants and a .LIB file containing procedures already assembled into object code.

8.4.1 Associating Libraries with Modules

You can choose either of two methods for associating your libraries with the modules that use them: you can use the **INCLUDELIB** directive inside your source files or link the modules from the command line.

Specify library names with INCLUDELIB.

To associate a specified library with your object code, use **INCLUDELIB**. You can add this directive to the source file to specify the libraries you want linked, rather than specifying them in the LINK command line. The **INCLUDELIB** syntax is

INCLUDELIB *libraryname*

The *libraryname* can be a file name or a complete path specification. If you do not specify an extension, .LIB is assumed. The *libraryname* is placed in the comment record of the object file. LINK reads this record and links with the specified library file.

For example, the statement `INCLUDELIB GRAPHICS` passes a message from the assembler to the linker telling LINK to use library routines from the file GRAPHICS.LIB. If this statement is in the source file DRAW.ASM and GRAPHICS.LIB is in the same directory, the program can be assembled and linked with the following command line:

```
ML DRAW.ASM
```

Link libraries with command-line options.

Without the **INCLUDELIB** directive, the program DRAW.ASM has to be linked with either of the following command lines:

```
ML DRAW.ASM GRAPHICS.LIB
ML DRAW /link GRAPHICS
```

If you want to assemble and link separately, you can use

```
ML /c DRAW.ASM
LINK DRAW,,,GRAPHICS
```

LINK searches in a specific order.

If you do not specify a complete path in the **INCLUDELIB** statement or at the command line, LINK searches for the library file in the following order:

1. In the current directory
2. In any directories in the library field of the LINK command line
3. In any directories in the LIB environment variable

The LIB utility provided with MASM 6.0 helps you create, organize, and maintain run-time libraries.

8.4.2 Using EXTERN with Library Routines

In some cases, **EXTERN** helps you limit the size of your executable file by specifying in the syntax an alternative name for a procedure. You would use this form of the **EXTERN** directive when declaring a procedure or symbol that may not need to be used.

The syntax looks like this:

EXTERN [[*langtype*]] *name* [[(*altname*)]] **:***qualifiedtype*

The addition of the *altname* to the syntax provides the name of an alternate procedure that the linker uses to resolve the external reference if the procedure given by *name* is not needed. Both *name* and *altname* must have the same *qualifiedtype*.

When the linker encounters an external definition for a procedure that gives an *altname*, the linker finishes processing that module before it links the object module that contains the procedure given by *name*. If the program does not reference any symbols in the *name* file's object from any of the linked modules, the assembler uses *altname* to satisfy the external reference. This saves space because the library object module is not brought in.

For example, assume that the contents of STARTUP.ASM include these statements:

```
EXTERN  init(dummy)
        .
        .
        .
dummy   PROC
        .
        .
        .                       ; A procedure definition containing no
        ret                     ;  executable code

dummy   ENDP
        .
        .
        .
        call    init    ; Defined in FLOAT.OBJ
```

In this example, the reference to the routine `init` (defined in FLOAT.OBJ) does not force the module FLOAT.OBJ to be linked into the executable file. If another reference causes FLOAT.OBJ to be linked into the executable file, then `init` will refer to the `init` label in FLOAT.OBJ. If there are no references which force FLOAT.OBJ to be loaded, then the alternate name for `init(dummy)` will be used by the linker.

8.5 Related Topics in Online Help

In addition to information covered in this chapter, information on the following topics can be found in online help.

Topic	Access
LIB	From the "Microsoft Advisor Contents" screen, choose "LIB" from the list of Microsoft Utilities
INCLUDE, **INCLUDELIB**, **EXTERNDEF**, **COMM**, and **PUBLIC**	From the "MASM 6.0 Contents" screen, choose "Directives," then "Scope and Visibility"
TYPEDEF	From the "MASM 6.0 Contents" screen, choose "Directives," then "Complex Data Types"
PROTO and **INVOKE**	From the "MASM 6.0 Contents" screen, choose "Directives," then "Procedures and Code Labels"
OPTION directive	From the "MASM 6.0 Contents" screen, choose "Directives," then "Miscellaneous"
@CurSeg	From the "MASM 6.0 Contents" screen, choose "Predefined Symbols"
PWB Options menu	From the "Microsoft Advisor Contents" screen, choose "Programmer's WorkBench"

Chapter 9
Using Macros

A "macro" is a symbolic name you give to a series of characters (a text macro) or to one or more statements (a macro procedure or function). As the assembler evaluates each line of your program, it scans the source code for names of previously defined macros. When it finds one, it substitutes the macro text for the macro name. In this way, you can avoid writing the same code several places in your program.

This chapter describes the following types of macros:

- Text macros, which expand to text within a source statement
- Macro procedures, which expand to one or more complete statements and can optionally take parameters
- Repeat blocks, which generate a group of statements a specified number of times or until a specified condition becomes true
- Macro functions, which look like macro procedures and can be used like text macros but which also return a value
- Predefined macro functions and string directives, which perform string operations

Macro processing is a text-processing mechanism that is done sequentially at assembly time. By the end of assembly, all macros have been expanded and the resulting text assembled into object code.

This chapter shows how to use macros for simple code substitutions as well as how to write sophisticated macros with parameter lists and repeat loops. It also describes how to use these features in conjunction with local symbols, macro operators, and predefined macro functions.

9.1 Text Macros

You can give a sequence of characters a symbolic name and then use the name in place of the text later in the source code. The named text is called a text macro.

The syntax for defining a text macro is

name **TEXTEQU** *<text>*
name **TEXTEQU** *macroId | textmacro*
name **TEXTEQU** *%constExpr*

where *text* is a sequence of characters enclosed in angle brackets, *macroId* is a previously defined macro function (see Section 9.6), *textmacro* is a previously defined text macro, and *%constExpr* is an expression that evaluates to text. The use of angle brackets to delimit text is discussed in more detail in Section 9.3.1, and the % operator is explained in Section 9.3.2.

Here are some examples:

```
msg     TEXTEQU <Some text>          ; Text assigned to symbol
string  TEXTEQU msg                  ; Text macro assigned to symbol
msg     TEXTEQU <Some other text>    ; New text assigned to symbol
value   TEXTEQU %(3 + num)           ; Text representation of
                                     ;  resolved expression assigned
                                     ;  to symbol
```

In the first line, text is assigned to the symbol `msg`. In the second line, the text of the `msg` text macro is assigned to a new text macro called `string`. In the third line, new text is assigned to `msg`. The result is that `msg` has the new text value, while `string` has the original text value. The fourth line assigns `7` to `value` if `num` equals `4`. If a text macro expands to another text macro (or macro function, which is discussed in Section 9.6), the resulting text macro will be recursively expanded.

Text macros are useful for naming strings of text that do not evaluate to integers. For example, you might use a text macro to name a floating-point constant or a bracketed expression. Here are some practical examples:

```
pi      TEXTEQU <3.1416>             ; Floating point constant
WPT     TEXTEQU <WORD PTR>           ; Sequence of key words
arg1    TEXTEQU <[bp+4]>             ; Bracketed expression
```

NOTE Use of the **TEXTEQU** directive to define text macros is new in MASM 6.0. In previous versions, you can use the **EQU** directive for the same purpose. If you have old code that worked under previous versions, it should still work under 6.0. However, the more consistent and flexible **TEXTEQU** is recommended for new code.

9.2 Macro Procedures

If your program needs to perform the same task many times, you can avoid having to type the same statements each time by writing a macro procedure. Macro procedures (commonly called macros) can be seen as text-processing mechanisms that automatically generate repeated text.

The term "macro procedure" rather than macro is used when necessary to distinguish between macro procedures and macro functions (a new feature of MASM 6.0 described in Section 9.6, "Returning Values with Macro Functions").

9.2.1 Creating Macro Procedures

To define a macro procedure without parameters, place the desired statements between the **MACRO** and **ENDM** directives:

name **MACRO**
statements
ENDM

For example, suppose you want a program to beep when it encounters certain errors. A `beep` macro can be defined as follows:

```
beep    MACRO
    mov  ah, 2          ;; Select DOS Print Char function
    mov  dl, 7          ;; Select ASCII 7 (bell)
    int  21h            ;; Call DOS
ENDM
```

Macro comments must start with two semicolons instead of one.

The double semicolons mark the beginning of macro comments. Macro comments appear in a listing file only at the macro's initial definition, not at the point where it is called and expanded. Listings are usually easier to read if the comments aren't always expanded. Regular comments (those with a single semicolon) are listed in macro expansions. Appendix C discusses listing files and shows examples of how macros are expanded in listings.

Once a macro is defined, you can call it anywhere in the program by using the macro's name as a statement. The following example calls the `beep` macro two times if an error flag has been set.

```
.IF     error   ; If error flag is true
beep            ;  execute macro two times
beep
.ENDIF
```

The instructions in the macro take the place of the macro call when the program is assembled. This would be the resulting code (from the listing file):

```
                              .IF     error
0017  80 3E 0000 R 00     *           cmp     error, 000h
001C  74 0C               *           je      @C0001
                            beep
001E  B4 02                 1           mov     ah, 2
0020  B2 07                 1           mov     dl, 7
0022  CD 21                 1           int     21h
                            beep
0024  B4 02                 1           mov     ah, 2
0026  B2 07                 1           mov     dl, 7
0028  CD 21                 1           int     21h
                            .ENDIF
002A                    *@C0001:
```

Contrast this with the results of defining `beep` as a procedure using the **PROC** directive and then calling it using the **CALL** instruction. The instructions of the procedure occur only once in the executable file, but you would also have the additional overhead of the **CALL** and **RET** instructions.

Macros are usually faster than run-time procedures.

In some cases the same task can be done with either a macro or a procedure. Macros are potentially faster because they have less overhead, but they generate the same code multiple times rather than just once.

9.2.2 Passing Arguments to Macros

Parameters allow macros to execute variations of a general task.

By defining parameters for macros, you can define a general task and then execute variations of it by passing different arguments each time you call the macro. The complete syntax for a macro procedure includes a parameter list:

name **MACRO** *parameterlist*
statements
ENDM

The *parameterlist* can contain any number of parameters. Use commas to separate each parameter in the list. Parameter names cannot be reserved words unless the keyword has been disabled with **OPTION NOKEYWORD**, the compatibility modes have been set by specifying **OPTION M510** (see Section 1.3.2), or the /Zm command-line option has been set.

To pass arguments to a macro, place the arguments after the macro name when you call the macro:

macroname arglist

All text between matching quotation marks in an *arglist* is considered one text item.

The `beep` macro introduced in the last section used the DOS interrupt to write the bell character (ASCII 7). It can be rewritten with a parameter to specify any character to write.

```
writechar MACRO char
    mov  ah, 2                  ;; Select DOS Print Char function
    mov  dl, char               ;; Select ASCII char
    int  21h                    ;; Call DOS
ENDM
```

Wherever `char` appears in the macro definition, the assembler replaces it with the argument in the macro call. Each time you call `writechar`, you can print a different value:

```
        writechar 7             ; Causes computer to beep
        writechar 'A'           ; Writes A to screen
```

If you pass more arguments than there are parameters, the additional arguments generate a warning (unless you use the **VARARG** keyword; see Section 9.4.3). If you pass fewer arguments than the macro procedure expects, remaining parameters are assigned empty strings (unless default values have been specified). This may cause errors. For example, if you call the `writechar` macro with no argument, it generates the following:

```
        mov     dl,
```

The assembler generates an error for the expanded statement but not for the macro definition or the macro call.

Macros can be made more flexible by leaving off macro arguments or adding additional ones. The next section tells some of the ways you can handle missing or extra arguments.

9.2.3 Specifying Required and Default Parameters

You can specify required and default parameters for macros.

You can give macro parameters special attributes to make them more flexible and improve error handling; you can make them required, give them default values, or vary their number. Because variable parameters are used almost exclusively with the **FOR** directive, discussion of them is postponed until Section 9.4.3, "FOR Loops and Variable-Length Parameters."

The syntax for a required parameter is

parameter:**REQ**

For example, you can rewrite the `writechar` macro to require the `char` parameter:

```
writechar MACRO char:REQ
    mov  ah, 2                  ;; Select DOS Print Char function
    mov  dl, char               ;; Select ASCII char
    int  21h                    ;; Call DOS
ENDM
```

If the call does not include a matching argument, the assembler reports the error in the line that contains the macro call. The effect of **REQ** is to improve error reporting.

A default value fills in missing parameters.

Another way to handle missing parameters is to specify a default value. The syntax is

parameter:=*textvalue*

Suppose that you often use `writechar` to beep by printing ASCII 7. The following macro definition uses an equal sign to tell the assembler to assume the parameter `char` is `7` unless you specify otherwise:

```
writechar  MACRO char:=<7>
    mov  ah, 2                  ;; Select DOS Print Char function
    mov  dl, char               ;; Select ASCII char
    int  21h                    ;; Call DOS
ENDM
```

In this case, `char` is not required. If you don't supply a value, the assembler fills in the blank with the default value of `7` and the macro beeps when called.

The default parameter value is enclosed in angle brackets so that the supplied value will be recognized as a text value. Section 9.3.1, "Text Delimiters (< >) and the Literal-Character Operator (!)," explains this in more detail.

Missing arguments can also be handled with the **IFB**, **IFNB**, **.ERRB**, and **.ERRNB** directives. They are described briefly in Section 1.3.3, "Conditional Directives," and in online help. Here is a slightly more complex macro that uses some of these techniques.

```
Scroll MACRO distance:REQ, attrib:=<07h>, tcol, trow, bcol, brow
    IFNB <tcol>             ;; Ignore arguments if blank
        mov   cl, tcol
    ENDIF
    IFNB <trow>
        mov   ch, trow
    ENDIF
    IFNB <bcol>
        mov   dl, bcol
    ENDIF
    IFNB <brow>
        mov   dh, brow
    ENDIF
```

```
    IFDIFI <attrib>, <bh>    ;; Don't move BH onto itself
        mov   bh, attrib
    ENDIF
    IF distance LE 0         ;; Negative scrolls up, positive down
        mov   ax, 0600h + (-(distance) AND 0FFh)
    ELSE
        mov   ax, 0700h + (distance AND 0FFh)
    ENDIF
    int   10h
ENDM
```

In this macro, the `distance` parameter is required. The `attrib` parameter has a default value of `07h` (white on black), but the macro also tests to make sure the corresponding argument isn't BH, since it would be inefficient (though legal) to load a register onto itself. The **IFNB** directive is used to test for blank arguments. These are ignored to allow the user to manipulate rows and columns directly in registers CX and DX at run time.

The following are two valid ways to call the macro:

```
; Assume DL and CL already loaded
dec     dh                    ; Decrement top row
inc     ch                    ; Increment bottom row
Scroll -3                     ; Scroll white on black dynamic
                              ;  window up three lines
Scroll 5, 17h, 2, 2, 14, 12   ; Scroll white on blue constant
                              ;  window down five lines
```

This macro can generate completely different code, depending on its arguments. In this sense, it is not comparable to a procedure, which always has the same code regardless of arguments.

9.2.4 Defining Local Symbols in Macros

You can make a symbol local to a macro by declaring it at the start of the macro with the **LOCAL** directive. Any identifier may be declared local.

You can choose whether you want numeric equates and text macros to be local or global. If a symbol will be used only inside a particular macro, you can declare it local so that the name will be available for other declarations inside other macros or at the global level. On the other hand, it is sometimes convenient to define text macros and equates that are not local, so that their values can be shared between macros.

If you need to use a label inside a macro, you must declare it local, since a label can occur only once in the source. The **LOCAL** directive makes a special instance of the label each time the macro is called. This prevents redefinition of the label.

All local symbols must be declared immediately following the **MACRO** statement (although blank lines and comments may precede the local symbol).

Separate each symbol with a comma. Comments are allowed on the **LOCAL** statement. Multiple **LOCAL** statements are also permitted. Here is an example macro that declares local labels:

```
power   MACRO   factor:REQ, exponent:REQ
    LOCAL   again, gotzero      ;; Local symbols
    sub     dx, dx              ;; Clear top
    mov     ax, 1               ;; Multiply by one on first loop
    mov     cx, exponent        ;; Load count
    jcxz    gotzero             ;; Done if zero exponent
    mov     bx, factor          ;; Load factor
again:
    mul     bx                  ;; Multiply factor times exponent
    loop    again               ;; Result in AX
gotzero:
ENDM
```

If the labels `again` and `gotzero` were not declared local, the macro would work the first time it is called, but it would generate redefinition errors on subsequent calls. MASM implements local labels by generating different names for them each time the macro is called. You can see this in listing files. The labels in the `power` macro might be expanded to `??0000` and `??0001` on the first call and to `??0002` and `??0003` on the second.

9.3 Assembly Time Variables and Macro Operators

In writing macros, you will often assign and modify values assigned to symbols. These symbols can be thought of as assembly-time variables. Like memory variables, they are symbols that represent values. But since macros are processed at assembly time, any symbol modified in a macro must be resolved as a constant by the end of assembly.

The three kinds of assembly-time variables are:

- Macro parameters
- Text macros
- Macro functions

When a macro is expanded, the symbols are processed in the order shown above. First macro parameters are replaced with the text of their actual arguments. Then text macros are expanded.

Macro parameters are similar to procedure parameters in some ways, but they also have important differences. In a procedure, a parameter has a type and a memory location. Its value can be modified within the procedure. In a macro, a parameter is a placeholder for the argument text. The value can only be assigned

to another symbol or used directly; it cannot be modified. The macro may interpret the argument text it receives either as a numeric value or as a text value.

It is important to understand the difference between text values and numeric values. Numeric values can be processed with arithmetic operators and assigned to numeric equates. Text values can be processed with macro functions and assigned to text macros.

Macro operators are often helpful when processing assembly-time variables. Table 9.1 shows the macro operators that MASM provides:

Table 9.1 MASM Macro Operators

Symbol	Name	Description
< >	Text Delimiters	Opens and closes a literal string.
!	Literal-Character Operator	Treats the next character as a literal character, even if it would normally have another meaning.
%	Expansion Operator	Causes the assembler to expand a constant expression or text macro.
&	Substitution Operator	Tells the assembler to replace a macro parameter or text macro name with its actual value.

The next sections explain these operators in detail.

9.3.1 Text Delimiters (< >) and the Literal-Character Operator (!)

The angle brackets (< >) are text delimiters. The most common reason to delimit a text value is when assigning a text macro. You can do this with **TEXTEQU**, as previously shown, or with the **SUBSTR** and **CATSTR** directives discussed in Section 9.5, "String Directives and Predefined Functions."

By delimiting the text of macro arguments, you can pass text that includes spaces, commas, semicolons, and other special characters. In the following example, assume you have previously defined a macro called `work`:

```
work    <1, 2, 3, 4, 5> ; Passes one argument
                        ;  with 15 characters
work    1, 2, 3, 4, 5   ; Passes five arguments, each
                        ;  with 1 character
```

Since angle brackets are delimiters, you can't include them as part of a delimited text value. The literal-character operator (!) can be used to override this

limitation. It forces the assembler to treat the character following it literally rather than as a special character.

```
errstr  TEXTEQU <Expression !> 255>  ; errstr = "Expression > 255"
```

Text delimiters also have a special use with the **FOR** directive, as explained in Section 9.4.3.

9.3.2 Expansion Operator (%)

The expansion operator (%) expands text macros or converts constant expressions into their text representations. It performs these tasks differently in different contexts, as discussed below.

9.3.2.1 The Expansion Operator with Constants

The expansion operator can be used in any context where a text value is expected but a numeric value is supplied. In these contexts, it can be thought of as a conversion operator to convert numeric values to text values.

The expansion operator forces immediate evaluation of a constant expression and replaces it with a text value consisting of the digits of the result. The digits are generated in the current radix (default decimal).

This application of the expansion operator is useful when defining a text macro:

```
a       TEXTEQU <3 + 4>          ; a = "3 + 4"
b       TEXTEQU %3 + 4           ; b = "7"
```

When assigning text macros, numeric equates can be used in the constant expressions, but text macros cannot:

```
num     EQU     4                ; num = 4
numstr  TEXTEQU <4>              ; numstr = <4>
a       TEXTEQU %3 + num         ; a = <7>
b       TEXTEQU %3 + numstr      ; b = <7>
```

The expansion operator can be used when passing macro arguments. If you want the value rather than the text of an expression to be passed, use the expansion operator. Use of the expansion operator depends on whether you want the expression to be evaluated inside the macro on each use, or outside the macro once. The following macro

```
work    MACRO   arg
    mov ax, arg * 4
ENDM
```

can be called with these statements:

```
        work    2 + 3           ; Passes "2 + 3"
                                ; Code: mov ax, 2 + 3 * 4 (14)
        work    %2 + 3          ; Passes 5
                                ; Code: mov ax, 5 * 4 (20)
```

Notice that because of operator precedence, results can vary depending on whether the expansion operator is used. Sometimes parentheses can be used inside the macro to force evaluation in a particular order:

```
work    MACRO   arg
    mov ax, (arg) * 4
ENDM

        work    2 + 3           ; Code: mov ax, (2 + 3) * 4 (20)
        work    %2 + 3          ; Code: mov ax, (5) * 4 (20)
```

This example generates the same code regardless of whether you pass the argument as a value or as text, but in some cases you need to specify how the argument is passed.

The value for a default argument must be text, but frequently you need to give a constant value. The expansion operator is one way to force the conversion. The following statements are equivalent:

```
work    MACRO   arg:=<07h>
work    MACRO   arg:=%07h
```

The expansion operator also has several uses with macro functions. See Section 9.6.

9.3.2.2 The Expansion Operator with Symbols

When you use the expansion operator on a macro argument, any text macros or numeric equates in the argument are expanded:

```
num     EQU     4
numstr  TEXTEQU <4>

        work    2 + num         ; Passes "2 + num"
        work    %2 + num        ; Passes "6"
        work    2 + numstr      ; Passes "2 + numstr"
        work    %2 + numstr     ; Passes "6"
```

The arguments can optionally be enclosed in parentheses. For example, these two statements are equivalent:

```
        work    %2 + num
        work    %(2 + num)
```

9.3.2.3 The Expansion Operator as the First Character on a Line

The expansion operator has a different meaning when used as the first character on a line. In this case, it instructs the assembler to expand any text macros and macro functions it finds on the rest of the line.

This feature makes it possible to use text macros with directives such as **ECHO**, **TITLE**, and **SUBTITLE** that take an argument consisting of a single text value. For instance, **ECHO** displays its argument to the standard output device during assembly. Such expansion can be useful for debugging macros and expressions, but the requirement that its argument be a single text value may have unexpected results:

```
        ECHO    Bytes per element: %(SIZEOF array / LENGTHOF array)
```

Instead of evaluating the expression, this line just echoes it:

```
Bytes per element: %(SIZEOF array / LENGTHOF array)
```

However, you can achieve the desired result by assigning the text of the expression to a text macro and then using the expansion operator at the beginning of the line to force expansion of the text macro.

```
temp    TEXTEQU %(SIZEOF array / LENGTHOF array)
%       ECHO    Bytes per element: temp
```

Note that you cannot get the same results by simply putting the % at the beginning of the first echo line, because % expands only text macros, not numeric equates or constant expressions.

Here are more examples of the use of the expansion operator at the start of a line:

```
; Assume memmod, lang, and os are passed in with /D option
%   SUBTITLE  Model: memmod  Language: lang  Operating System: os

; Assume num defined earlier
tnum    TEXTEQU %num
%       .ERRE   num LE 255, <Failed because tnum !> 255>
```

9.3.3 Substitution Operator (&)

In MASM 6.0, the substitution operator (&) enables substitution of macro parameters, even when the parameter occurs within a larger word or within a quoted string. It can also be used to concatenate two macro parameters after they have been expanded.

The syntax for the substitution operator looks like this:

&*parametername*&

The operators delimiting a name always tell the assembler to substitute the actual argument for the name. However, the substitution operator is often optional. The substitution operator is not necessary when there is a space or separation character (comma, tab, or other operator) on that side. In the case of a parameter name inside a string, at least one substitution operator must appear.

The rules for using the substitution operator have changed significantly since MASM 5.1, making macro behavior more consistent and flexible. If you have macros written for a previous version of MASM, you can specify the old behavior by using **OLDMACROS** or **M510** with the **OPTION** directive (see Section 1.3.2).

In the macro

```
work    MACRO   arg
    mov ax, &arg& * 4
ENDM
```

the **&** symbols tell the assembler to replace the value of `arg` with the corresponding argument. However, the characters on both the right and left are spaces. Therefore, the operators are unnecessary. The macro would normally be written like this:

```
work    MACRO   arg
    mov ax, arg * 4
ENDM
```

The substitution operator is used for one of the following reasons:

- To paste together two parameter names or a parameter name and text
- To indicate that a parameter name inside double or single quotation marks should be expanded rather than be treated as part of the quoted string

This macro illustrates both uses:

```
errgen  MACRO   num, msg
    PUBLIC  err&num
    err&num BYTE    "Error &num: &msg"
ENDM
```

When called with the following arguments,

```
        errgen  5, <Unreadable disk>
```

the macro generates this code:

```
        PUBLIC  err5
err5    BYTE    "Error 5: Unreadable disk"
```

In the second line of the macro, the left & symbol must be provided because it is adjacent to the r character, which is a valid identifier symbol. The right & symbol is not needed because there is a space to the right of the m. The statement pastes the text err to the argument value 5 to generate the symbol err5.

The substitution operator is used again inside quotation marks at the start of the parameter names num and msg to indicate that these names should be expanded. In this case, no pasting operation is necessary, so either operator could be omitted, but not both. The macro line could have been written as

```
err&num BYTE    "Error num&: msg&"
```

or

```
err&num BYTE    "Error &num&: &msg&"
```

The assembler processes substitution operators from left to right. This can have unexpected results when you are pasting together two macro parameters. For example, if arg1 has the value var and arg2 has the value 3, you could paste them together with this statement:

```
&arg1&&arg2&    BYTE    "Text"
```

Eliminating extra substitution operators, you might expect the following to be equivalent:

```
&arg1&arg2      BYTE    "Text"
```

However, this actually produces the symbol vararg2 because in processing from left to right the assembler associates both the first and the second & symbols with the first parameter. The assembler replaces &arg1& by var , producing vararg2 . The arg2 is never evaluated. The correct abbreviation is

```
arg1&&arg2      BYTE    "Text"
```

which produces the desired symbol var3. The symbol arg1&&arg2 is replaced by var&arg2, which is replaced by var3.

The substitution operator is also necessary if you want a text macro substituted inside quotes. For example,

```
arg     TEXTEQU <hello>
%echo   This is a string "&arg" ; Produces: This is a string "hello"
%echo   This is a string "arg"  ; Produces: This is a string "arg"
```

The substitution operator can also be used in lines beginning with the expansion operator (%) symbol, even outside macros (see Section 9.3.2.3). Text macros are always expanded in such lines, but it may be necessary to use the substitution operator to paste text macro names to adjacent characters or symbol names, as shown below:

```
text    TEXTEQU <var>
value   TEXTEQU %5
%       ECHO    textvalue is text&&value
```

This echoes the message

```
textvalue is var5
```

Bit-test and macro expansion statements can be confused.

The single ampersand (**&**) is the bit-test operator in MASM, as it is for C. This operator is also used in macro expansion as the substitute operator. Macro substitution always occurs before evaluation of the high-level control structures; therefore, in ambiguous cases, the **&** operator is treated as a macro-expansion character. You can always guarantee the correct use of the bit-test operator by enclosing the bit-test operands in parentheses. The example below illustrates these two uses.

```
test    MACRO   x
    .IF ax==&x      ; &x substituted with parameter value
    mov     ax, 10
    .ELSEIF ax&(x)  ; & is bitwise AND
    mov ax, 20
    .ENDIF
    ENDM
```

9.4 Defining Repeat Blocks with Loop Directives

A "repeat block" is an unnamed macro defined with a loop directive. It generates the statements inside the repeat block a specified number of times or until a given condition becomes true.

Several loop directives are available, providing different ways of specifying the number of iterations. Some loop directives also provide a way to specify arguments for each iteration. Although the number of iterations is usually specified in the directive, you can use the **EXITM** directive to exit from the loop early.

Repeat blocks can be used outside macros, but they frequently appear inside macro definitions to perform some repeated operation in the macro.

This section explains the following four loop directives: **REPEAT**, **WHILE**, **FOR**, and **FORC**. In previous versions of MASM, **REPEAT** was called **REPT**, **FOR** was called **IRP**, and **FORC** was called **IRPC**. MASM 6.0 still recognizes the old names.

NOTE The **REPEAT** and **WHILE** directives should not be confused with the **.REPEAT** and **.WHILE** directives (see Section 7.2.1, "Loop-Generating Directives"), which generate loop and jump instructions for run-time program control.

9.4.1 REPEAT Loops

Repeat loops are expanded at assembly time.

The **REPEAT** directive is the simplest loop directive. It specifies the number of times to generate the statements inside the macro. The syntax is

REPEAT *constexpr*
statements
ENDM

The *constexpr* can be a constant or a constant expression, and must contain no forward references. Since the repeat block will be expanded at assembly time, the number of iterations must be known then.

Here is an example of a repeat block used to generate data. It initializes an array containing sequential ASCII values for all uppercase letters.

```
alpha   LABEL   BYTE            ;  Name the data generated
letter  =       'A'             ;  Initialize counter
REPEAT  26                      ;; Repeat for each letter
    BYTE    letter              ;; Allocate ASCII code for letter
    letter  = letter + 1        ;; Increment counter
ENDM
```

Here is another use of **REPEAT**, this time inside a macro:

```
beep    MACRO   iter:=<3>
    mov ah, 2                   ;; Character output function
    mov dl, 7                   ;; Bell character
    REPEAT iter                 ;; Repeat number specified by macro
        int 21h                 ;; Call DOS
    ENDM
ENDM
```

9.4.2 WHILE Loops

The **WHILE** directive is similar to **REPEAT**, but the loop continues as long as a given condition is true. The syntax is

WHILE *expression*
statements
ENDM

The *expression* must be a value that can be calculated at assembly time. Normally the expression uses relational operators, but it can be any expression that evaluates to zero (false) or nonzero (true). Usually, the condition changes during the evaluation of the macro so that the loop won't attempt to generate an infinite amount of code. However, you can use the **EXITM** directive to break out of the loop.

Loops are especially useful for generating lookup tables.

The following repeat block uses the **WHILE** directive to allocate variables initialized to calculated values. This is a common technique for generating lookup tables. Frequently it is faster to look up a value precalculated by the assembler at assembly time than to have the processor calculate the value at run time.

```
cubes   LABEL   BYTE            ;; Name the data generated
root    =   1                   ;; Initialize root
cube    =   root * root * root  ;; Calculate first cube
WHILE   cube LE 32767           ;; Repeat until result too large
    WORD   cube                 ;; Allocate cube
    root   =    root + 1        ;; Calculate next root and cube
    cube   =    root * root * root
ENDM
```

9.4.3 FOR Loops and Variable-Length Parameters

With the **FOR** directive you can iterate through a list of arguments, doing some operation on each of them in turn. It has the following syntax:

FOR *parameter*, *<argumentlist>*
statements
ENDM

The *parameter* is a placeholder that will be used as the name of each argument inside the **FOR** block. The argument list must be a list of comma-separated arguments and must always be enclosed in angle brackets, as the following example illustrates:

```
series  LABEL   BYTE
FOR     arg, <1,2,3,4,5,6,7,8,9,10>
    BYTE  arg DUP (arg)
ENDM
```

On the first iteration, the `arg` parameter is replaced with the first argument, the value 1. On the second iteration `arg` is replaced with 2. The result is an array with the first byte initialized to 1, the next two bytes initialized to 2, the next three bytes initialized to 3, and so on.

In this example the argument list is given specifically, but in some cases the list must be generated as a text macro. The value of the text macro must include the angle brackets.

```
arglist TEXTEQU <!<3,6,9!>>     ; Generate list as text macro
FOR  arg, arglist
    .                           ; Do something to arg
    .
    .
ENDM
```

Note the use of the literal character operator (**!**) to use angle brackets as characters, not delimiters (see Section 9.3.1).

Variable parameter lists provide flexibility.

The **FOR** directive also provides a convenient way to process macros with a variable number of arguments. To do this, add **VARARG** to the last parameter to indicate that a single named parameter will have the actual value of all additional arguments. For example, the following macro definition includes the three possible parameter attributes—required, default, and variable.

```
work    MACRO   rarg:REQ, darg:=<5>, varg:VARARG
```

The variable argument must always come last. If this macro is called with the statement

```
        work 5, , 6, 7, a, b
```

the first argument is received as passed, the second is replaced by the default value `5`, and the last four are received as the single argument `<6, 7, a, b>`. This is the same format expected by the **FOR** directive. The **FOR** directive discards leading spaces but recognizes trailing spaces.

The following macro illustrates variable arguments:

```
show    MACRO chr:VARARG
    mov     ah, 02h
    FOR arg, <chr>
        mov     dl, arg
        int     21h
    ENDM
ENDM
```

When called with

```
        show  'O', 'K', 13, 10
```

the macro displays each of the specified characters one at a time.

The parameter in a **FOR** loop can have the required or default attribute. The `show` macro can be modified to make blank arguments generate errors:

```
show    MACRO chr:VARARG
    mov     ah, 02h
    FOR arg:REQ, <chr>
        mov     dl, arg
        int     21h
    ENDM
ENDM
```

The macro now generates an error if called with

```
        show  'O',, 'K', 13, 10
```

Another approach would be to use a default argument:

```
show    MACRO chr:VARARG
    mov     ah, 02h
    FOR arg:=<' '>, <chr>
        mov     dl, arg
        int     21h
    ENDM
ENDM
```

Now if the macro is called with

```
        show  'O',, 'K', 13, 10
```

it inserts the default character, a space, for the blank argument.

9.4.4 FORC Loops

The **FORC** directive is similar to **FOR** but takes a string of text rather than a list of arguments. The statements are assembled once for each character (including spaces) in the string, substituting a different character for the parameter each time through.

The syntax looks like this:

FORC *parameter*, < *text*>
statements
ENDM

The *text* must be enclosed in angle brackets. The following example illustrates **FORC**:

```
FORC arg, <ABCDEFGHIJKLMNOPQRSTUVWXYZ>
    BYTE  '&arg'              ;; Allocate uppercase letter
    BYTE  '&arg' + 20h        ;; Allocate lowercase letter
    BYTE  '&arg' - 40h        ;; Allocate ordinal of letter
ENDM
```

Notice that the substitution operator must be used inside the quotation marks to make sure that `arg` is expanded to a character rather than treated as a literal string.

With earlier versions of MASM, **FORC** is often used for complex parsing tasks. A long sentence can be examined character by character. Each character is then either thrown away or pasted onto a token string, depending on whether it is a separator character. In MASM 6.0, the predefined macro functions and string processing directives discussed in Section 9.5 are usually more efficient for these tasks.

9.5 String Directives and Predefined Functions

Predefined macro string functions are new to MASM 6.0.

The assembler provides the following directives for manipulating text: **SUBSTR**, **INSTR**, **SIZESTR**, and **CATSTR**. Each of these has a corresponding predefined macro function version: **@SubStr**, **@InStr**, **@SizeStr**, and **@CatStr**.

You use the directive versions to assign a processed value to a text macro or numeric equate. For example, **CATSTR**, which concatenates a list of text values, can be used like this:

```
num     =       7
newstr  CATSTR  <3 + >, %num, < = > , %3 + num ; "3 + 7 = 10"
```

Assignment with **CATSTR** and **SUBSTR** works like assignment with the **TEXTEQU** directive. Assignment with **SIZESTR** and **INSTR** works like assignment with the = operator.

The arguments to directives must be text values. Use the expansion operator to make sure that constants and numeric equates are expanded to text.

The macro function versions are similar, but their arguments must be enclosed in parentheses. Macro functions return text values and can be used in any context where text is expected. Section 9.6 tells how to write your own macro functions. An equivalent statement to the previous example using **CATSTR** is

```
num     =       7
newstr  TEXTEQU @CatStr( <3 + >, %num, < = > , %3 + num )
```

Although the directive version is simpler in the example above, the function versions are often convenient because they can be used as arguments to string directives or to other macro functions.

Unlike the string directives, predefined macro function names are case sensitive. Since MASM is not case sensitive by default, the case doesn't matter unless you use the /Cp command-line option.

The following sections summarize the syntax for each of the string directives and functions. The explanations focus on the directives, but the functions work the same except where noted.

SUBSTR

name **SUBSTR** *string*, *start*[[, *length*]]
@SubStr(*string*, *start*[[, *length*]] **)**

The **SUBSTR** directive assigns a substring from a given *string* to a new symbol, specified by *name*. *Start* specifies the position (1-based) in *string* to start the substring. *Length* specifies the length of the substring. If *length* is not given, it is assumed to be the remainder of the string including the *start* character. The *string*

in the **SUBSTR** syntax, as well as in the syntax for the other string directives and predefined functions, can be any *textItem* where *textItem* can be text enclosed in angle brackets (< >), the name of a macro, or a constant expression preceded by % (*%constExpr*).

INSTR

name **INSTR** [[*start*,]] *string*, *substring*
@InStr([[*start*]], *string*, *substring*)

The **INSTR** directive searches a specified *string* for an occurrence of a given *substring* and assigns its position (1-based) to *name*. The search is case sensitive. *Start* is the position in *string* to start the search for *substring*. If *start* is not given, it is assumed to be 1 (the start of the string). If *substring* is not found, the position assigned to *name* is 0.

If the **INSTR** directive is used, the position value is assigned to a name as if it were a numeric equate. If the **@InStr** function is used, the value is returned as a string of digits in the current radix.

The **@InStr** function has a slightly different syntax than the **INSTR** directive. You can omit the first argument and its associated comma from the directive. You can leave the first argument blank with the function, but a blank function argument must still have a comma. For example,

```
pos     INSTR   <person>, <son>
```

is the same as

```
pos     = @InStr( , <person>, <son> )
```

The return value could also be assigned to a text macro:

```
strpos  TEXTEQU @InStr( , <person>, <son> )
```

SIZESTR

name **SIZESTR** *string*
@SizeStr(*string*)

The **SIZESTR** directive assigns the number of characters in *string* to *name*. An empty string assigns a length of zero. Although the length is always a positive number, it is assigned as a string of digits in the current radix rather than as a numeric value.

If the **SIZESTR** directive is used, the size value is assigned to a name as if it were a numeric equate. If the **@SizeStr** function is used, the value is returned as a string of digits in the current radix.

CATSTR

name **CATSTR** *string*[[, *string*]]...
@CatStr(*string*[[, *string*]]...)

The **CATSTR** directive concatenates a list of text values specified by *string* into a single text value and assigns it to *name*. **TEXTEQU** is technically a synonym for **CATSTR**. **TEXTEQU** is normally used for single-string assignments, while **CATSTR** is used for multistring concatenations.

The following example that pushes and pops one set of registers illustrates several uses of string directives and functions:

```
; SaveRegs - Macro to generate a push instruction for each
; register in argument list. Saves each register name in the
; regpushed text macro.
regpushed TEXTEQU <>                    ;; Initialize empty string

SaveRegs MACRO regs:VARARG
    FOR reg, <regs>                     ;; Push each register
        push reg                        ;;  and add it to the list
        regpushed CATSTR   <reg>, <,>, regpushed
    ENDM                                ;; Strip off last comma
    regpushed CATSTR <!<>, regpushed    ;; Mark start of list with <
    regpushed SUBSTR regpushed, 1, @SizeStr( regpushed )
    regpushed CATSTR regpushed, <!>>    ;; Mark end with >
ENDM

; RestoreRegs - Macro to generate a pop instruction for registers
; saved by the SaveRegs macro. Restores one group of registers.

RestoreRegs MACRO
    LOCAL regs
    %FOR reg, regpushed                  ;; Pop each register
pop reg
    ENDM
 ENDM
```

Notice how the `SaveRegs` macro saves its result in the `regpushed` text macro for later use by the `RestoreRegs` macro. In this case, a text macro is used as a global variable. By contrast, the `regs` text macro is used only in `RestoreRegs`. It is declared **LOCAL** so that it won't take the name `regs` from the global name space. The MACROS.INC file provided with MASM 6.0 includes expanded versions of these same two macros.

9.6 Returning Values with Macro Functions

A macro function returns a text string.

A macro function is a named group of statements that returns a value. When a macro function is called, its argument list must be enclosed in parentheses, even if the list is empty. The value returned is always text.

Macro functions are new to MASM 6.0, as are several predefined macro functions for common tasks. The predefined macros include **@Environ** (see Section 1.2.3) and the string functions **@SizeStr**, **@CatStr**, **@SubStr**, and **@InStr** (discussed in the preceding section).

Macro functions are defined in exactly the same way as macro procedures, except that a value must always be returned using the **EXITM** directive. Here is an example:

```
DEFINED MACRO   symbol:REQ
    IFDEF symbol
        EXITM <-1>              ;; True
    ELSE
        EXITM <0>               ;; False
    ENDIF
ENDM
```

This macro works like the defined operator in the C language. You can use it to test the defined state of several different symbols with a single statement, as shown below:

```
IF DEFINED( DOS ) AND NOT DEFINED( XENIX )
    ;; Do something
ENDIF
```

Notice that the macro returns integer values as strings of digits, but the **IF** statement evaluates numeric values or expressions. There is no conflict because the value returned by the macro function is seen in the statement exactly as if the user had typed the values directly into the program:

```
IF -1 AND NOT 0
```

Returning Values with EXITM

The return value must be text, a text equate name, or the result of another macro function. If a function must return a numeric value (such as a constant, a numeric equate, or the result of a numeric expression), it must first convert the value to text using angle brackets or the expansion operator (**%**). The defined macro, for example, could have returned its value as

```
        EXITM   %-1
```

Although macro functions can include any legal statement, they seldom need to include instructions. This is because a macro function is expanded and its value returned at assembly time, while instructions are executed at run time.

Here is another example of a macro function. It uses the **WHILE** directive to calculate factorials:

```
factorial   MACRO   num:REQ
    LOCAL   i, factor
    factor  =   num
    i       =   1
    WHILE   factor GT 1
        i       =   i * factor
        factor  =   factor - 1
    ENDM
    EXITM   %i
ENDM
```

The integer result of the calculation is changed to a text string with the expansion operator (%). The `factorial` macro can be used to define data, as shown below:

```
var     WORD    factorial( 4 )
```

The effect of this statement is to initialize `var` with the number 24 (the factorial of 4).

Using Macro Functions with Variable-Length Parameter Lists

Macro functions can enhance FOR loops.

You can use the **FOR** directive to handle macro parameters with the **VARARG** attribute. Section 9.4.3 explains how to do this in simple cases where the variable parameters are handled sequentially, from first to last. However, you may sometimes need to process the parameters in reverse order or nonsequentially. Macro functions make these techniques possible.

You may need to know the number of arguments in a **VARARG** parameter. The following macro functions handle this.

```
@ArgCount MACRO arglist:VARARG
    LOCAL count
    count = 0
    FOR arg, <arglist>
        count = count + 1       ;; Count the arguments
    ENDM
    EXITM %count
ENDM
```

You could use this inside a macro that has a **VARARG** parameter, as shown below:

```
work    MACRO args:VARARG
%   ECHO Number of arguments is: @ArgCount( args )
ENDM
```

Another useful task might be to select an item from an argument list using an index to indicate which item. The following macro simplifies this.

```
@ArgI MACRO index:REQ, arglist:VARARG
    LOCAL count, retstr
    retstr TEXTEQU <>              ;; Initialize count
    count  = 0                     ;; Initialize return string
    FOR arg, <arglist>
        count = count + 1
        IF count EQ index          ;; Item is found
            retstr TEXTEQU <arg>   ;; Set return string
            EXITM                  ;;  and exit IF
        ENDIF
    ENDM
    EXITM retstr                   ;; Exit function
ENDM
```

This function can be used as shown below:

```
work    MACRO args:VARARG
%   ECHO Third argument is: @ArgI( 3, args )
ENDM
```

Finally, you might need to process arguments in reverse order. The following macro returns a new argument list in reverse order.

```
@ArgRev MACRO arglist:REQ
    LOCAL txt, arg
    txt TEXTEQU <>
%   FOR arg, <arglist>
        txt CATSTR <arg>, <,>, txt      ;; Paste each onto list
    ENDM
                                        ;; Remove terminating comma
    txt SUBSTR  txt, 1, @SizeStr( %txt ) - 1
    txt CATSTR  <!<>, txt, <!>>         ;; Add angle brackets
    EXITM txt
ENDM
```

You could call this function as shown below:

```
work    MACRO   args:VARARG
%   FOR  arg, @ArgRev( <args> )   ;; Process in reverse order
        ECHO    arg
    ENDM
ENDM
```

These three macro functions are provided on the MASM distribution disk in the MACROS.INC include file.

Macro Operators and Macro Functions

This list summarizes the behavior of the expansion operator with macro functions.

- If a macro function is not preceded by a %, it will be expanded. However, if it expands to a text macro or a macro function call, the result will not be expanded further.
- If you use a macro function call as an argument for another macro function call, a % is not needed.
- If a macro function expands to a text macro (or another macro function), the macro function will be recursively expanded.
- If a macro function is called inside angle brackets and is preceded by %, it will be expanded.

9.7 Advanced Macro Techniques

The concept of replacing macro names with predefined macro text is simple in theory, but it has many implications and complications. Here is a brief summary of some advanced techniques you can use in macros.

9.7.1 Nesting Macro Definitions

Macros can define other macros or can be redefined. MASM does not process nested definitions until the outer macro has been called. Therefore, the inner macros cannot be called until the outer macro has been called. The nesting of macro definitions is limited only by memory.

```
shifts  MACRO   opname                  ;; Macro generates macros
    opname&s    MACRO operand:REQ, rotates:=<1>
        IF rotates LE 2                 ;; One at a time is faster
            REPEAT rotate               ;;  for 2 or less
                opname  operand, 1
            ENDM
        ELSE                            ;; Using CL is faster for
            mov     cl, rotates         ;;  more than 2
            opname  operand, cl
        ENDIF
    ENDM
ENDM
```

```
; Call macro to make new macros
shifts  ror                     ; Generates rors
shifts  rol                     ; Generates rols
shifts  shr                     ; Generates shrs
shifts  shl                     ; Generates shls
shifts  rcl                     ; Generates rcls
shifts  rcr                     ; Generates rcrs
shifts  sal                     ; Generates sals
shifts  sar                     ; Generates sars
```

This macro generates enhanced versions of the shift and rotate instructions. The macros could be called like this:

```
shrs    ax, 5
rols    bx, 3
```

The macro versions handle multiple shifts by generating different code, depending on how many shifts are specified. The example above is optimized for the 8088 and 8086 processors. If you want to enhance for other processors, you can simply change the outer macro; it automatically changes all the inner macros. Code that uses the inner macros benefits from the enhancements but does not change so long as the macro interface doesn't change.

9.7.2 Testing for Argument Type and Environment

Macros can check the type of arguments and generate different code depending on what they find. For example, you can use the **OPATTR** operator to determine if an argument is a constant, a register, or a memory operand.

If you discover a constant value, you can often optimize the code. In some cases, you can generate better code for 0 or 1 than for other constants. If the argument is a memory operand, you know nothing about the value of the operand, since it may change at run time. However, you may want to generate different code depending on the operand size and on whether it is a pointer. Similarly, if the operand is a register, you know nothing of its contents, but you may be able to optimize if you can identify a particular register with the **IFDIFI** or **IFIDNI** directives.

The following example illustrates some of these techniques. It loads a specified address into a specified offset register. The segment register is assumed to be DS.

```
load    MACRO reg:REQ, adr:REQ
    IF (OPATTR (adr)) AND 00010000y     ;; Register
        IFDIFI reg, adr                 ;; Don't load register
            mov     reg, adr            ;;  onto itself
        ENDIF
    ELSEIF (OPATTR (adr)) AND 00000100y
        mov     reg, adr                ;; Constant
    ELSEIF (TYPE (adr) EQ BYTE) OR (TYPE (adr) EQ SBYTE)
        mov     reg, OFFSET adr         ;; Bytes
    ELSEIF (SIZE (TYPE (adr)) EQ 2
        mov     reg, adr                ;; Near pointer
    ELSEIF (SIZE (TYPE (adr)) EQ 4
        mov     reg, WORD PTR adr[0]    ;; Far pointer
        mov     ds,  WORD PTR adr[2]
    ELSE
        .ERR <Illegal argument>
    ENDIF
ENDM
```

A macro may also generate different code depending on the assembly environment. The predefined text macro **@Cpu** can be used to test for processor type. The following example uses the more efficient constant variation of the **PUSH** instruction if the processor is an 80186 or higher.

```
IF  @Cpu AND 00000010y
    pushc  MACRO op             ;; 80186 or higher
        push op
    ENDM
ELSE
    pushc  MACRO op             ;; 8088/8086
        mov  ax, op
        push ax
    ENDM
ENDIF
```

Note that the example generates a completely different macro for the two cases. This is more efficient than testing the processor inside the macro and conditionally generating different code. With this macro, the environment is checked only once; if the conditional were inside the macro it would be checked every time the macro is called.

You can test the language and operating system using the **@Interface** text macro. The memory model can be tested with the **@Model**, **@DataSize**, or **@CodeSize** text macros.

You can save the contexts inside macros with **PUSHCONTEXT** and **POPCONTEXT**. The options for these keywords are:

Option	Description
RADIX	Saves segment register information
LIST	Saves listing and CREF information
CPU	Saves current CPU and processor
ALL	All of the above

9.7.3 Using Recursive Macros

Macros can call themselves. In previous versions of MASM, recursion is an important technique for handling variable arguments. With MASM 6.0, you can do this much more cleanly using the **FOR** directive and the **VARARG** attribute, as described in Section 9.4.3. However, recursion is still available and may be useful for some macros.

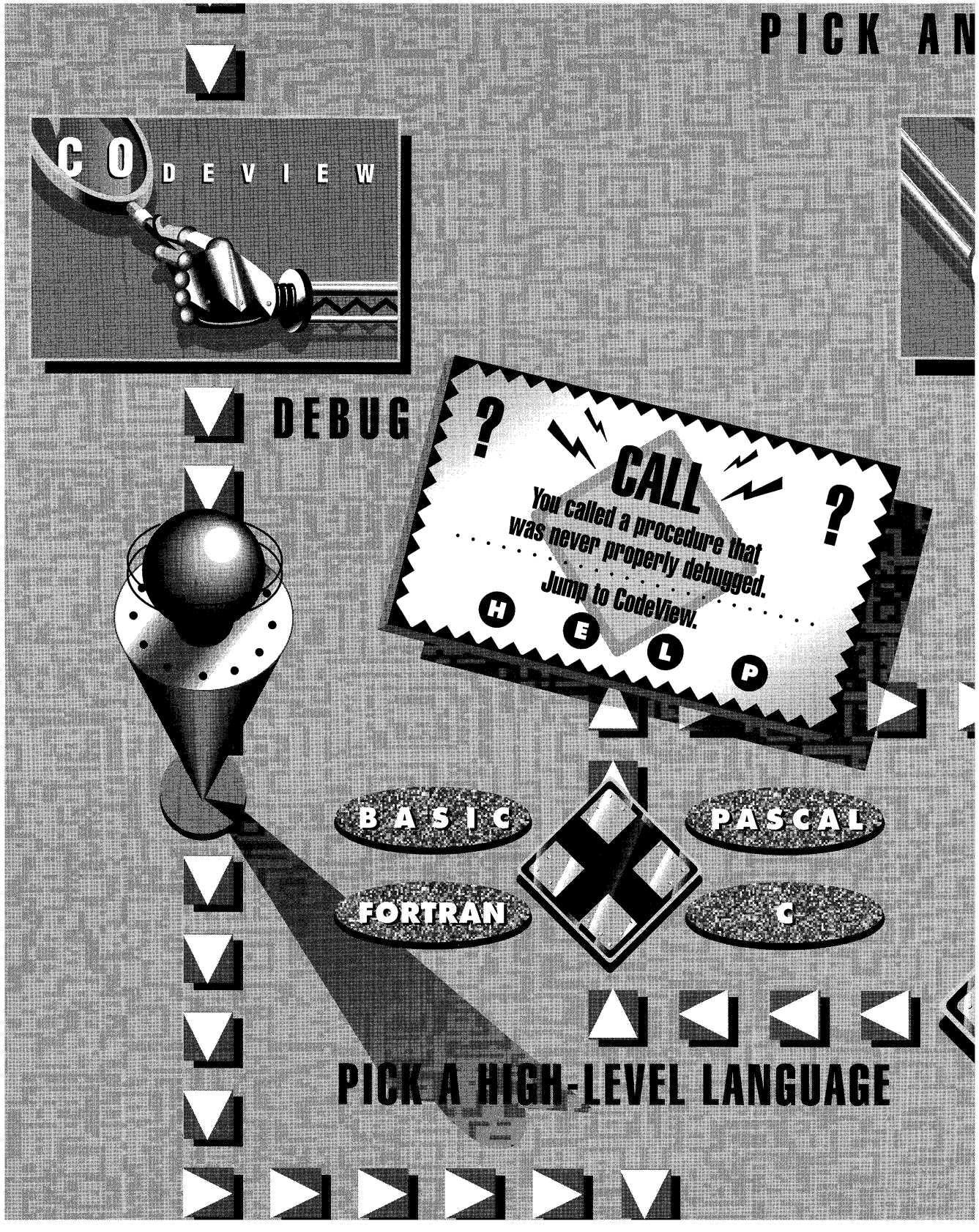
9.8 Related Topics in Online Help

In addition to information covered in this chapter, information on the following topics can be found in online help. From the "MASM 6.0 Contents" screen:

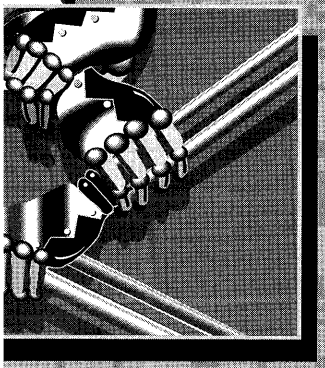
Topics	Access
INCLUDE	Choose "Directives," and then "Scope and Visibility"
GOTO, PURGE	Choose "Directives," and then "Macros and Iterative Blocks"
.LISTMACRO	Choose "Directives," and then "Listing Control"
IFB, IFNB, IFDIFI, and **IFIDNI**	Choose "Directives," and then "Conditional Assembly"
ECHO	Choose "Directives," and then "Miscellaneous"
OPATTR	Choose "Operators," and then "Miscellaneous"
@Cpu, @Interface, @DataSize, @Environ, and **@CodeSize**	Choose "Predefined Symbols"
PUSHCONTEXT, POPCONTEXT	Choose "Directives" and then "Iterative Blocks"

Part 2
Improving Programmer Productivity

Chapters

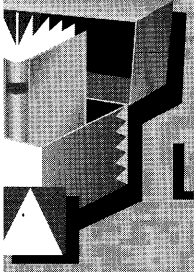

PICK AN
CODEVIEW
DEBUG
?
CALL
You called a procedure that was never properly debugged.
Jump to CodeView.
HELP
?
BASIC
PASCAL
FORTRAN
C
PICK A HIGH-LEVEL LANGUAGE

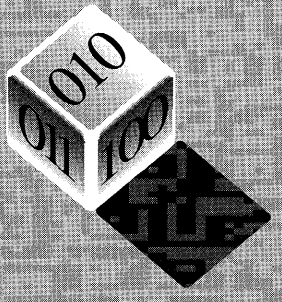
NVIRONMENT

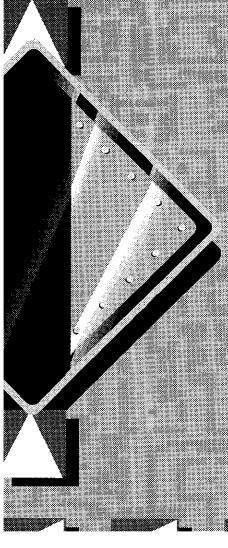


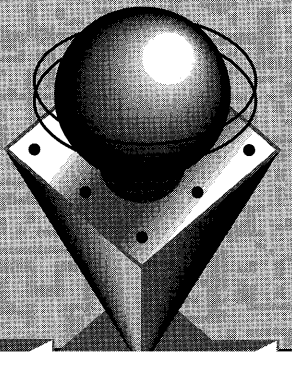
LINK

LIB



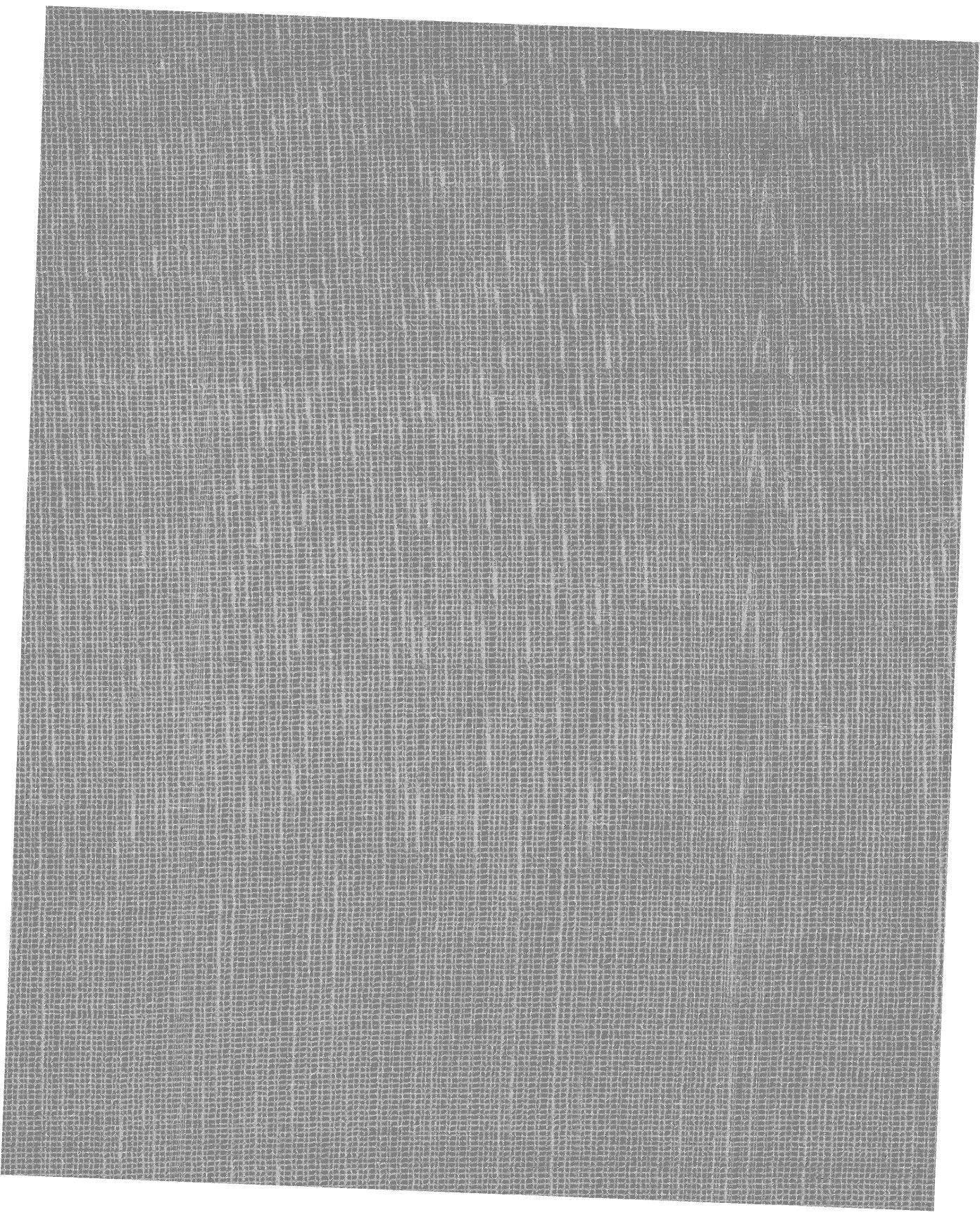






PICK A CARD

PICK A CARD



Chapter 10

Managing Projects with NMAKE

The Microsoft Program Maintenance Utility (NMAKE) is a sophisticated command processor that saves time and simplifies project management. Once you specify which project files depend on others, NMAKE automatically executes the commands needed to update your project when any project file has changed.

The advantage of using NMAKE instead of simple batch files is that NMAKE recompiles only those files that need recompiling. NMAKE doesn't waste time with files that haven't changed since the last build. NMAKE also has advanced features (such as macros) that simplify managing complex projects.

This chapter includes examples that show how each feature of NMAKE works. In addition, Section 10.9, "A Sample NMAKE Description File," shows how many of these features work together.

If you are using the Microsoft Programmer's WorkBench (PWB) to build your project, PWB automatically creates a description file (called a "makefile" in the PWB documentation) and calls NMAKE to run the file. You may want to read this chapter if you intend to build your program outside of PWB or if you want to understand or modify a description file created by PWB.

A utility called NMK allows you to use NMAKE to manage your project under DOS (or in a DOS session under OS/2). Section 10.11, "Using NMK," explains when and how to use NMK.

If you are familiar with MAKE, the predecessor to NMAKE, be sure to read Section 10.10, "Differences between NMAKE and MAKE." These utilities differ in several important respects.

10.1 Overview of NMAKE

NMAKE works by looking at the last times and dates of modification for a "target" file and its "dependents" and then comparing them. A target is usually a file you want to create, such as an executable file. A dependent is usually a file from which a target is created, such as a source file. A target is "out-of-date" if any of its dependents has changed more recently than the target.

WARNING For NMAKE to work properly, the date and time setting on your system must be consistent relative to previous settings. If you set the date and time each time you start the system, be careful to set it accurately. If your system stores a setting, be certain that the battery is working.

When you run NMAKE, it reads a "description file" that you supply. The description file consists of one or more description blocks. Each description block typically lists a target, the target's dependents, and the commands that build the target. NMAKE compares the last time the targets changed to the last time the dependents changed. If the modification time of any dependents is the same or later than the time of the target, NMAKE updates the target by executing the command or commands listed in the description block.

NMAKE's main purpose is to help you update applications quickly and simply. However, it can execute any DOS or OS/2 command, so it is not limited to compiling and linking. NMAKE can also make backups, move files, and perform other project-management tasks that you ordinarily do at the operating-system prompt.

10.2 Running NMAKE

You invoke NMAKE with the following syntax:

NMAKE [[*options*]] [[*macros*]] [[*targets*]]

The *options* field lists NMAKE options, which are described in Section 10.4, "Command-Line Options."

The *macros* field lists macro definitions, which allow you to change text in the description file. The syntax for macros is described in "User-Defined Macros" in Section 10.3.4.1, "Macros."

The *targets* field lists targets to build. NMAKE rebuilds only the targets listed on the command line. If you don't specify any targets, NMAKE builds only the first target in the description file. (This behavior departs significantly from that of MAKE. See Section 10.10, "Differences between NMAKE and MAKE.")

NMAKE follows the instructions you specify in a description file.

NMAKE searches the current directory for the name of a description file you specify with the /F option. It halts and displays an error message if the file does not exist. If you do not use the /F option to specify a description file, NMAKE searches the current directory for a description file named MAKEFILE. If MAKEFILE does not exist, NMAKE checks the command line for target files and tries to build them using predefined inference rules (either default or defined in TOOLS.INI). This feature lets you use NMAKE without a description file (as long as NMAKE has a predefined inference rule for the target). If the command line does not specify any target files, NMAKE halts and displays an error message.

Example

```
NMAKE /S "program=sample" sort.exe search.exe
```

This command supplies four arguments: an option (`/S`), a macro definition (`"program=sample"`), and two target specifications (`sort.exe` and `search.exe`).

The command does not specify a description file, so NMAKE looks for the default description file, MAKEFILE. The /S option tells NMAKE not to display the commands as they are executed. (See Section 10.4, "Command-Line Options.") The macro definition performs a text substitution throughout the description file, replacing every instance of `program` with `sample`. The target specifications tell NMAKE to update the targets SORT.EXE and SEARCH.EXE.

10.3 NMAKE Description Files

The most important parts of a description file are the description blocks, which tell NMAKE how to build your project's target files. A description file can also contain comments, macros, inference rules, and directives. This section describes the elements of description files.

10.3.1 Description Blocks

Description blocks form the heart of the description file. Figure 10.1 illustrates a typical NMAKE description block, including the three sections: targets, dependents, and commands.

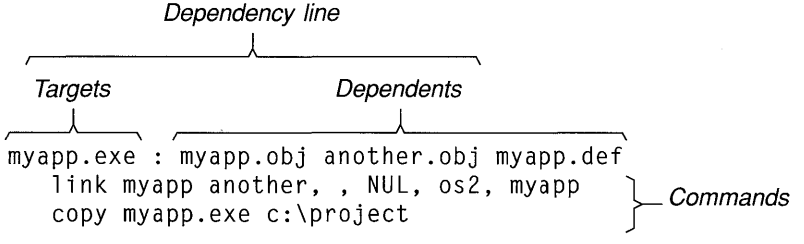


Figure 10.1 Typical Description Block

10.3.1.1 Targets

The target is the file that you want to build.

The targets section of the dependency line lists one or more files to build. The line that lists targets and dependents is called the "dependency line."

The example in Figure 10.1 tells NMAKE how to build a single target, MYAPP.EXE, if it is missing or out-of-date. Although single targets are

common, you can also list multiple targets in a single dependency line; you must separate each target name with a space. If the name of the last target before the colon (:) is one character long, put a space between the name and the colon, so NMAKE won't interpret the character as a drive specification.

A target can appear in only one dependency line when specified as shown above. To update a target using more than one description block, specify two consecutive colons (::) between targets and dependents. For details, see Section 10.3.1.8, "Specifying a Target in Multiple Description Blocks."

The target is usually a file, but it can also be a "pseudotarget," a name that lets you build groups of files or execute a group of commands. For more information, see Section 10.3.2, "Pseudotargets."

10.3.1.2 Dependents

A dependent is a file used to build a target.

The dependents section of the description block lists one or more files from which the target is built. A colon (:) separates it from the targets section. The example in Figure 10.1 lists three dependents after MYAPP.EXE:

```
myapp.exe : myapp.obj another.obj myapp.def
```

You can also specify the directories in which NMAKE should search for a dependent. Enclose one or more directory names in braces ({ }). Separate multiple directories with a semicolon (;). The syntax for a directory specification is

{*directory*[[;*directory...*]]}*dependent*

Example

The following dependency line tells NMAKE to search the current directory first, then the specified directories:

```
forward.exe : {\src\alpha;d:\proj}pass.obj
```

In the line above, the target, FORWARD.EXE, has one dependent, PASS.OBJ. The directory list specifies two directories:

```
{\src\alpha;d:\proj}
```

NMAKE first searches for PASS.OBJ in the current directory. If PASS.OBJ isn't there, NMAKE searches the \SRC\ALPHA directory, then the D:\PROJ directory. If NMAKE cannot find a dependent in the current directory or a listed directory, it looks for a description block with a dependency line containing PASS.OBJ as a target, and uses the commands in that description block to create PASS.OBJ. If NMAKE cannot find such a description block, it looks for an inference rule that describes how to create the dependent. (See Section 10.3.5, "Inference Rules.")

10.3.1.3 Dependency Line

The dependency line in Figure 10.1 tells NMAKE to rebuild the target MYAPP.EXE whenever MYAPP.OBJ, ANOTHER.OBJ, or MYAPP.DEF has changed more recently than MYAPP.EXE.

The object files in the dependency list above would never be newer than the executable file (unless you had recompiled the source code before running NMAKE). So NMAKE checks to see if the object files themselves are targets in other dependency lists, and if any dependents in those lists are targets elsewhere, and so on.

NMAKE continues moving through all dependencies this way to build a "dependency tree" that specifies all the steps required to fully update the target. If NMAKE then finds any dependents in the tree that are newer than the target, NMAKE updates the appropriate files and rebuilds the target.

10.3.1.4 Commands

The commands section can contain one or more commands.

The commands section of the description block lists the commands that NMAKE should use to build the target. You can use any command that can be executed from the command line. The example in Figure 10.1 tells NMAKE to build MYAPP.EXE using the following LINK command:

```
    link myapp another.obj, , NUL, os2, myapp
```

Notice that the line is indented. NMAKE uses indentation to distinguish between a dependency line and a command line. A command line must be indented at least one space or tab. The dependency line must not be indented (it cannot start with a space or tab).

Many targets are built with a single command, but you can place more than one command after the dependency line, each on a separate line, as shown in Figure 10.1.

A long command can span several lines if each line ends with a backslash (\). A backslash at the end of a line is equivalent to a space on the command line. For example, the command

```
    echo abcd\
efgh
```

is equivalent to the command

```
    echo abcd efgh
```

You can also place a command at the end of a dependency line. Use a semicolon (;) to separate the command from the rightmost dependent, as in

```
project.exe : project.obj ; link project;
```

OS/2 allows multiple commands on one command line.

OS/2 allows you to combine two or more commands on a single command line with an ampersand (&). For example, the following command line is legal in an OS/2 description file:

```
DIR & COPY sample.exe backup.exe
```

A slight restriction is imposed on the use of the CD, CHDIR, and SET commands in OS/2 description files. NMAKE executes these commands itself rather than passing them to OS/2. Therefore, if any of these commands is the first command on a line, the remaining commands are not executed because they aren't passed to OS/2.

The following multiple-command line does not display the directory listing because DIR is preceded by a CD command:

```
CD \mydir & DIR
```

To use CD, CHDIR, or SET in a description block, place these commands on separate lines:

```
CD \mydir
DIR
```

NMAKE interprets a percent symbol (%) within a command line as the start of a file specifier. To use a literal percent symbol in a command line, specify it as a double percent symbol (%%). (See Section 10.3.8, "Extracting Filename Components.")

10.3.1.5 Wild Cards

You can use DOS and OS/2 wild-card characters (* and ?) to specify target and dependent filenames. NMAKE expands the wild cards when analyzing dependencies and when building targets. For example, the following description block links all files having the .OBJ extension in the current directory:

```
project.exe : *.obj
    LINK $*.obj;
```

10.3.1.6 Command Modifiers

Command modifiers are special prefixes attached to the command. They provide extra control over the commands in a description block. You can use more than one modifier for a single command. Table 10.1 describes the three NMAKE command modifiers.

Table 10.1 Command Modifiers

Character	Action

@	Prevents NMAKE from displaying the command as it executes. In the example below, the at sign (@) suppresses display of the ECHO command line:
	`sort.exe : sort.obj`
	`  @ECHO Now sorting.`
	The output of the ECHO command is not suppressed.

-[[*number*]]	Turns off error checking for the command. Spaces and tabs can appear before the command. If the dash is followed by a number, NMAKE checks the exit code returned by the command and stops if the code is greater than the number. No space or tab can appear between the dash and number. (See Section 10.12, "Using Exit Codes with NMAKE.")
	In the following example, if the program `sample` returns an exit code, NMAKE does not stop but continues to execute commands; if `sort` returns an exit code greater than 5, NMAKE stops:
	`light.lst : light.txt`
	`  -sample light.txt`
	`  -5 sort light.txt`

!	Executes the command for each dependent file if the command preceded by the exclamation point uses the predefined macros **$**** or **$?**. (See Section 10.3.4, "Macros.") The **$**** macro refers to all dependent files in the description block. The **$?** macro refers to all dependent files in the description block that have a more recent modification time than the target. For example,
	`print : one.txt two.txt three.txt`
	`  !print $** lpt1:`
	generates the following commands:
	`print one.txt lpt1:`
	`print two.txt lpt1:`
	`print three.txt lpt1:`

10.3.1.7 Using Special Characters as Literals

You may need to specify as a literal character one of the characters that NMAKE uses for a special purpose. These characters are

```
: ; # ( ) $ ^ \ { } ! @  -
```

To use one of these characters literally, place a caret (^) in front of it. For example, suppose you define a macro that ends with a backslash:

```
exepath=c:\bin\
```

The line above is intended to define a macro named `exepath` with the value `c:\bin\`. But the second backslash has an unintended side effect. Since the backslash is NMAKE's line-continuation character, the line actually defines `exepath` as `c:\bin`, followed by whatever appears on the next line of the description file. You can avoid this problem by placing a caret in front of the second backslash:

```
exepath=c:\bin^\
```

You can also use a caret to insert a literal newline character in a string or macro:

```
XYZ=abc^
def
```

The caret tells NMAKE to interpret the newline character as part of the macro, not a line break. Note that this effect differs from using a backslash (\) to continue a line. A newline character that follows a backslash is replaced with a space.

NMAKE ignores carets that precede characters other than the special characters listed above. The line

```
ign^ore : these ca^rets
```

is interpreted as

```
ignore : these carets
```

A caret within a quoted string is treated as a literal caret character.

10.3.1.8 Specifying a Target in Multiple Description Blocks

You can specify a target in more than one description block by placing two colons (::) after the target. This feature is useful for building a complex target, such as a library, that contains components created with different commands. For example,

```
target.lib :: a.asm b.asm c.asm
    ML a.asm b.asm c.asm
    LIB target -+a.obj -+b.obj -+c.obj;
target.lib :: d.c e.c
    CL /c d.c e.c
    LIB target -+d.obj -+e.obj;
```

Both description blocks update the library named TARGET.LIB. If any of the assembly-language files have changed more recently than the library, NMAKE executes the commands in the first block to assemble the source files and update the library. Similarly, if any of the C-language files have changed, NMAKE executes the second group of commands to compile the C files and update the library.

If you use a single colon in the example above, NMAKE issues an error message. It is legal, however, to use single colons if the target appears in only one block. In this case, dependency lines are cumulative. For example,

```
target : jump.bas
target : up.c
    echo Building target...
```

is equivalent to

```
target : jump.bas up.c
    echo Building target...
```

No commands can appear between cumulative dependency lines, but blank lines, comment lines, macro definitions, and directives can appear.

10.3.2 Pseudotargets

A "pseudotarget" is similar to a target, but it is not a file. It is a name used as a label for executing a group of commands. In the following example, `UPDATE` is a pseudotarget.

```
UPDATE : *.*
    !COPY $** a:\product
```

NMAKE always considers the pseudotarget to be out-of-date. In the previous example, NMAKE copies all the dependent files to the specified drive and directory.

Like target names, pseudotarget names are not case sensitive.

10.3.3 Comments

You can place comments in a description file by preceding them with a number sign (#):

```
# Comment on line by itself
OPTIONS = /MAP  # Comment on macro's line
all.exe : one.obj two.obj  # Comment on dependency line
    link $(OPTIONS) one.obj two.obj;
```

A comment extends to the end of the line in which it appears. Command lines (and dependency lines containing commands) cannot contain comments.

To specify a literal #, precede it with a caret (^), as in the following:

```
DEF=^#define #Macro representing a C preprocessing directive
```

10.3.4 Macros

Macros offer a convenient way to replace a particular string in the description file with another string. Macros are useful for a variety of tasks, including the following:

- Creating a single description file that works for several projects. You can define a macro that replaces a dummy filename in the description file with the specific filename for a particular project.
- Controlling the options NMAKE passes to the compiler or linker. When you specify options in a macro, you can change options throughout the description file in a single step.

You can define your own macros or use predefined macros. This section describes user-defined macros first.

10.3.4.1 User-Defined Macros

You can define a macro with this syntax:

macroname=string

The *macroname* can be any combination of letters, digits, and the underscore (_) character. Macro names are case sensitive. NMAKE interprets `MyMacro` and `MYMACRO` as different macro names.

The *string* can be any sequence of zero or more characters. (A string of zero characters is called a "null string." A string consisting only of spaces, tabs, or both is also considered a null string.) For example,

```
linkcmd=LINK /map
```

defines a macro named `linkcmd` and assigns it the string `LINK /map`.

You can define macros in the description file, on the command line, in a command file (see Section 10.5, "NMAKE Command File"), or in TOOLS.INI (see Section 10.6, "The TOOLS.INI File"). Each macro defined in the description file must appear on a separate line. The line cannot start with a space or tab.

When you define a macro in the description file, NMAKE ignores spaces on either side of the equal sign. The *string* itself can contain embedded spaces. You do not need to enclose *string* in quotation marks (if you do, they become part of the string).

Slightly different rules apply when you define a macro on the command line or in a command file. The command-line parser treats spaces as argument delimiters. Therefore, the *string* itself, or the entire macro, must be enclosed in double quotation marks if it contains embedded spaces. All three forms of the following command-line macro are legal and equivalent:

```
NMAKE program=sample
NMAKE "program=sample"
NMAKE "program = sample"
```

The macro `program` is passed to NMAKE, with an assigned value of `sample`.

If the string contains spaces, either the string or the entire macro must appear within quotes. Either form of the following command-line macro is allowed:

```
NMAKE linkcmd="LINK /map"
NMAKE "linkcmd=LINK /map"
```

However, the following form of the same macro is not allowed. It contains spaces that are not enclosed by quotation marks:

```
NMAKE linkcmd = "LINK /map"
```

A macro name can be given a null value. Both of the following definitions assign a null value to the macro `linkoptions`:

```
NMAKE linkoptions=
NMAKE linkoptions="    "
```

A macro name can be "undefined" with the **!UNDEF** preprocessing directive (see Section 10.3.7, "Preprocessing Directives"). Assigning a null value to a macro name does not undefine it; the name is still defined, but with a null value.

A macro can be followed by a comment, using the syntax described in the preceding section on comments.

10.3.4.2 Using Macros

Use a macro by enclosing its name in parentheses preceded by a dollar sign ($). For example, you can use the `linkcmd` macro defined above by specifying

```
$(linkcmd)
```

NMAKE replaces every occurrence of `$(linkcmd)` with `LINK /map`.

The following description file defines and uses three macros:

```
program=sample
L=LINK
options=

$(program).exe : $(program).obj
    $(L)  $(options)  $(program).obj;
```

NMAKE interprets the description block as

```
sample.exe : sample.obj
    LINK sample.obj;
```

NMAKE replaces every occurrence of `$(program)` with `sample`, every instance of `$(L)` with `LINK`, and every instance of `$(options)` with a null string.

An undefined macro is replaced by a null string.

If you use as a macro a name that has never been defined, or was undefined, NMAKE treats that name as a null string. No error occurs.

To use the dollar sign ($) as a literal character, specify two dollar signs ($$).

The parentheses are optional if *macroname* is a single character. For example, `$L` is equivalent to `$(L)`. However, parentheses are recommended for consistency.

10.3.4.3 Special Macros

NMAKE provides several special macros to represent various filenames and commands. One use for these macros is in predefined inference rules. (See Section 10.3.5.4.) Like user-defined macro names, special macro names are case sensitive. For example, NMAKE interprets `CC` and `cc` as different macro names.

Tables 10.2 through 10.5 summarize the four categories of special macros. The filename macros offer a convenient representation of filenames from a dependency line; these are listed in Table 10.2. The recursion macros, listed in Table 10.3, allow you to call NMAKE from within your description file. Tables 10.4 and 10.5 describe the command macros and options macros that make it convenient for you to invoke the Microsoft language compilers.

The filename macros conveniently represent filenames from the dependency line.

Table 10.2 lists macros that are predefined to represent file names. As with all one-character macros, these do not need to be enclosed in parentheses. (The $$@ and $** macros are exceptions to the parentheses rule for macros; they do not require parentheses even though they contain two characters.) Note that the macros in Table 10.2 represent filenames as you have specified them in the dependency line, and not the full specification of the filename.

Table 10.2 Filename Macros

Macro Reference	Meaning
$@	The current target's full name, as currently specified. This is not necessarily the full path name.
$*	The current target's full name minus the file extension.
$**	The dependents of the current target.
$?	The dependents that have a more recent modification time than the current target.
$$@	The target that NMAKE is currently evaluating. You can use this macro only to specify a dependent.
$<	The dependent file that has a more recent modification time than the current target (evaluated only for inference rules).

The example below uses the $? macro, which represents all dependents that are more recent than the target. The ! command modifier causes NMAKE to execute a command once for each dependent in the list (see Table 10.1). As a result, the LIB command is executed up to three times, each time replacing a module with a newer version.

```
trig.lib : sin.obj cos.obj arctan.obj
    !LIB trig.lib -+$?;
```

In the next example, NMAKE updates files in another directory by replacing them with files of the same name from the current directory. The $@ macro is used to represent the current target's full name:

```
#Files in objects directory depend on versions in current directory
DIR=c:\objects
$(DIR)\globals.obj : globals.obj
    COPY globals.obj $@
$(DIR)\types.obj : types.obj
    COPY types.obj $@
$(DIR)\macros.obj : macros.obj
    COPY macros.obj $@
```

Macro modifiers specify parts of the predefined filename macros.

You can append one of the modifiers in the following list to any of the filename macros to extract part of a filename. If you add one of these modifiers to the macro, you must enclose the macro name and the modifier in parentheses.

Modifier	Resulting Filename Part
D	Drive plus directory
B	Base name
F	Base name plus extension
R	Drive plus directory plus base name

For example, assume that **$@** has the value C:\SOURCE\PROG\SORT.OBJ. The following list shows the effect of combining each modifier with **$@**:

Macro Reference	Value
$(@D)	C:\SOURCE\PROG
$(@F)	SORT.OBJ
$(@B)	SORT
$(@R)	C:\SOURCE\PROG\SORT

If **$@** has the value SORT.OBJ without a preceding directory, the value of **$(@R)** is just SORT, and the value of **$(@D)** is a dot (.) to represent the current directory.

Recursion macros let you use NMAKE to call NMAKE.

Table 10.3 lists three macros that you can use when you want to call NMAKE recursively from within a description file.

Table 10.3 Recursion Macros

Macro Reference	Meaning
$(MAKE)	The name used to call NMAKE recursively. The line on which it appears is executed even if the /N command-line option is specified.
$(MAKEDIR)	The directory from which NMAKE is called.
$(MAKEFLAGS)	The NMAKE options currently in effect. This macro is passed automatically when you call NMAKE recursively. You cannot redefine this macro. Use the preprocessing directive **!CMDSWITCHES** to update the **MAKEFLAGS** macro. (See Section 10.3.7, "Preprocessing Directives.")

To call NMAKE recursively, use the command

```
$(MAKE) /$(MAKEFLAGS)
```

The **MAKE** macro is useful for building different versions of a program. The following description file calls NMAKE recursively to build targets in the \VERS1 and \VERS2 directories.

```
all : vers1 vers2

vers1 :
    cd \vers1
    $(MAKE)
    cd ..

vers2 :
    cd \vers2
    $(MAKE)
    cd ..
```

The example changes to the \VERS1 directory and then calls NMAKE recursively, causing NMAKE to process the file MAKEFILE in that directory. Then it changes to the \VERS2 directory and calls NMAKE again, processing the file MAKEFILE in that directory.

You can add options to the ones already in effect for NMAKE by following the **MAKE** macro with the options in the same syntax as you would specify them on the command line. You can also pass the name of a description file with the /F option instead of using a file named MAKEFILE.

Deeply recursive build procedures can exhaust NMAKE's run-time stack, causing an error. If this occurs, use the EXEHDR utility to increase NMAKE's run-time stack. The following command, for example, gives NMAKE.EXE a stack size of 16,384 (0x4000) bytes:

```
exehdr /stack:0x4000 nmake.exe
```

Command macros are shortcut calls to Microsoft compilers.

NMAKE defines several macros to represent commands for Microsoft products. (See Table 10.4.) You can use these macros as commands in a description block, or invoke them using a predefined inference rule. (See Section 10.3.5, "Inference Rules.") You can redefine these macros to represent part or all of a command line, including options.

Table 10.4 Command Macros

Macro Reference	Command Action	Predefined Value
$(AS)	Invokes the Microsoft Macro Assembler	`AS=ml`
$(BC)	Invokes the Microsoft Basic Compiler	`BC=bc`
$(CC)	Invokes the Microsoft C Compiler	`CC=cl`
$(COBOL)	Invokes the Microsoft COBOL Compiler	`COBOL=cobol`
$(FOR)	Invokes the Microsoft FORTRAN Compiler	`FOR=fl`
$(PASCAL)	Invokes the Microsoft Pascal Compiler	`PASCAL=pl`
$(RC)	Invokes the Microsoft Resource Compiler	`RC=rc`

Options macros pass preset options to Microsoft compilers.

The macros in Table 10.5 are used by NMAKE to represent options to be passed to the commands for Microsoft languages. By default, these macros are undefined. You can define them to mean the options you want to pass to the commands. Whether or not they are defined, the macros are used automatically in the predefined inference rules. If the macros are undefined, or if they are defined to be null strings, a null string is generated in the command line. (See Section 10.3.5.4, "Predefined Inference Rules.")

Table 10.5 Options Macros

Macro Reference	Passed to
$(AFLAGS)	Microsoft Macro Assembler
$(BFLAGS)	Microsoft Basic Compiler
$(CFLAGS)	Microsoft C Compiler
$(COBFLAGS)	Microsoft COBOL Compiler
$(FFLAGS)	Microsoft FORTRAN Compiler
$(PFLAGS)	Microsoft Pascal Compiler
$(RFLAGS)	Microsoft Resource Compiler

10.3.4.4 Substitution within Macros

You can replace text in a macro as well as in the description file.

Just as macros allow you to substitute text in a description file, you can also substitute text within a macro itself. The substitution is temporary; it applies only to the current use of the macro and does not modify the original macro definition. Use the following form:

$(*macroname*:*string1*=*string2*)

Every occurrence of *string1* is replaced by *string2* in the macro *macroname*. Do not put any spaces or tabs between *macroname* and the colon. Spaces between the colon and *string1* or between *string1* and the equal sign are part of *string1*. Spaces between the equal sign and *string2* or between *string2* and the right parenthesis are part of *string2*. If *string2* is a null string, all occurrences of *string1* are deleted from the *macroname* macro.

Macro substitution is case sensitive. This means that the case as well as the characters in *string1* must exactly match the target string in the macro, or the substitution is not performed. It also means that the *string2* substitution is exactly as specified.

Example 1

The following description file illustrates macro substitution:

```
SOURCES = project.for one.for two.for

project.exe : $(SOURCES:.for=.obj)
    LINK  $**;

COPY : $(SOURCES)
    !COPY $** c:\backup
```

The predefined macro $** stands for the names of all the dependent files (see Table 10.2).

If you invoke the example file with a command line that specifies both targets,

```
NMAKE project.exe copy
```

NMAKE executes the following commands:

```
LINK project.obj one.obj two.obj;
COPY project.for c:\backup
COPY one.for c:\backup
COPY two.for c:\backup
```

The macro substitution does not alter the SOURCES macro definition. Rather, it replaces the listed characters. When NMAKE builds the target PROJECT.EXE, it gets the definition for the predefined macro $** (the dependent list) from the dependency line, which specifies the macro substitution in SOURCES.

The same is true for the second target, COPY. In this case, however, no macro substitution is requested, so SOURCES retains its original value, and $** represents the names of the FORTRAN source files. (In the example above, the target COPY is a pseudotarget; Section 10.3.2 describes pseudotargets.)

Example 2

If the macro OBJS is defined as

```
OBJS=ONE.OBJ TWO.OBJ THREE.OBJ
```

with exactly one space between each object name, you can replace each space in the defined value of OBJS with a space, followed by a plus sign, followed by a newline, by using

```
$(OBJS: = +^
)
```

The caret (^) tells NMAKE to treat the end of the line as a literal newline character. This example is useful for creating response files.

10.3.4.5 Substitution within Predefined Macros

You can also substitute text in any predefined macro except $$@. The principle is the same as for other macros. The command in the following description block substitutes within a predefined macro. Note that even though $@ is a single-character macro, the substitution makes it a multi-character macro invocation, so it must be enclosed in parentheses.

```
target.abc : depend.xyz
    echo $(@:targ=blank)
```

If dependent depend.xyz has a later modification time than target target.abc, then NMAKE executes the command

```
echo blanket.abc
```

The example uses the predefined macro $@, which equals the full name of the current target (target.abc). It substitutes blank for targ in the target, resulting in blanket.abc.

10.3.4.6 Inherited Macros

When NMAKE executes, it inherits macro definitions equivalent to every environment variable. The inherited macro names are converted to uppercase.

Inherited macros can be used like other macros. You can also redefine them. The following example redefines the inherited macro PATH:

```
PATH = c:\tools\bin

sample.exe : sample.obj
    LINK sample;
```

Inherited macros take their definitions from environment variables.

No matter what value the environment variable PATH had before, it has the value `c:\tools\bin` when NMAKE executes the LINK command in this description block. Redefining the inherited macro does not affect the original environment variable; when NMAKE terminates, PATH still has its original value.

Inherited macros have one restriction: in a recursive call to NMAKE, the only macros that are preserved are those defined on the command line or in environment variables. Macros defined in the description file are not inherited when NMAKE is called recursively. To pass a macro to a recursive call:

- Use the SET command before the recursive call to set the variable for the entire NMAKE session.
- Define the macro on the command line for the recursive call.

The /E option causes macros inherited from environment variables to override any macros with the same name in the description file.

10.3.4.7 Precedence among Macro Definitions

If you define the same macro name in more than one place, NMAKE uses the macro with the highest precedence. The precedence from highest to lowest is as follows:

1. A macro defined on the command line
2. A macro defined in a description file or include file
3. An inherited environment-variable macro
4. A macro defined in the TOOLS.INI file
5. A predefined macro such as CC and AS

10.3.5 Inference Rules

Inference rules are templates that define how a file with one extension is created from a file with a different extension. When NMAKE encounters a description block that has no commands, it searches for an inference rule that matches the extensions of the target and dependent files. Similarly, if a dependent file doesn't exist, NMAKE looks for an inference rule that shows how to create the missing dependent from another file with the same base name.

Inference rules tell NMAKE how to create files with a specific extension.

Inference rules provide a convenient shorthand for common operations. For instance, you can use an inference rule to avoid repeating the same command in several description blocks. You can define your own inference rules or use predefined inference rules.

NOTE An inference rule is useful only when a target and dependent have the same base name, and have a one-to-one correspondence. For example, you cannot define an inference rule that replaces several modules in a library, because the modules would have different base names than the target library.

Inference rules can exist only for dependents with extensions that are listed in the **.SUFFIXES** directive. (For information on the **.SUFFIXES** directive, see Section 10.3.6, "Directives.") NMAKE searches in the current or specified directory for a file whose base name matches the target and whose extension is listed in the **.SUFFIXES** list. If it finds such a file, it applies the inference rule that matches the extensions of the target and the located file.

The **.SUFFIXES** list specifies an order of priority for NMAKE to use when searching for files. If more than one file is found, and thus more than one rule matches a dependency line, NMAKE searches the **.SUFFIXES** list and uses the rule whose extension appears earlier in the list. For example, the dependency line

```
project.exe :
```

can be matched to several predefined inference rules and possibly one or more user-defined rules, all of which describe a command for creating an .EXE file. NMAKE uses the inference rule corresponding to the first matching file it finds.

10.3.5.1 Inference Rule Syntax

An inference rule has the following syntax:

.fromext.toext:
  commands

The first line lists two extensions: *fromext* extension represents the filename extension of a dependent file, and *toext* represents the extension of a target file. Extensions are not case sensitive.

The second line of the inference rule gives the command to create a target file of *toext* from a dependent file of *fromext*. Use the same rules for commands in inference rules as in description blocks. (See Section 10.3.1, "Description Blocks.")

10.3.5.2 Inference Rule Search Paths

The inference-rule syntax described above tells NMAKE to look for the specified files in the current directory. You can also specify directories to be searched by NMAKE when it looks for files with the extensions *fromext* and *toext*. An inference rule that specifies paths has the following syntax:

{*frompath*}*.fromext* {*topath*}*.toext*:
  commands

NMAKE searches in the *frompath* directory for files with the *fromext* extension. It uses *commands* to create files with the *toext* extension in the *topath* directory, if the *fromext* file has a later modification time than the *toext* file.

The paths in the inference rule must exactly match the paths explicitly specified in the dependency line of a description block.

If you use a path on one element of the inference rule, you must use paths on both. You can specify the current directory for either element by using the operating system notation for the current directory, which is a dot (.), or by specifying an empty pair of braces.

You can specify only one path for each element in an inference rule. To specify more than one path, repeat the inference rule with the alternate path.

10.3.5.3 User-Defined Inference Rules

You can define inference rules in the description file or in TOOLS.INI (see Section 10.6, "The TOOLS.INI File"). An inference rule lists two file extensions and one or more commands.

Example 1

The following inference rule tells NMAKE how to build a .OBJ file from a .C file:

```
.c.obj:
    CL /c $<
```

In this example, the predefined macro $< represents the name of a dependent that has a more recent modification time than the target.

NMAKE applies this inference rule to the following description block:

```
sample.obj :
```

The description block lists only a target, SAMPLE.OBJ. Both the dependent and the command are missing. However, given the target's base name and extension, plus the inference rule, NMAKE has enough information to build the target.

NMAKE first looks for a file with the same base name as the target and with one of the extensions in the **.SUFFIXES** list. If SAMPLE.C exists (and no files with higher-priority extensions exist), NMAKE compares its time to that of SAMPLE.OBJ. If SAMPLE.C has changed more recently, NMAKE compiles it using the CL command listed in the inference rule:

```
CL /c sample.c
```

Example 2

The following inference rule compares a .C file in the current directory with the corresponding .OBJ file in another directory:

```
{.}.c{c:\objects}.obj:
    cl /c $<;
```

The path for the .C file is represented by a dot. A path for the dependent extension is required because one is specified for the target extension.

This inference rule matches a dependency line containing the same combination of paths, such as:

```
c:\objects\test.obj : test.c
```

This rule does not match a dependency line such as:

```
test.obj : test.c
```

In this case, NMAKE uses the predefined inference rule .c.obj when building the target.

10.3.5.4 Predefined Inference Rules

NMAKE provides predefined inference rules containing commands for creating object, executable, and resource files. Table 10.6 describes the predefined inference rules.

Table 10.6 Predefined Inference Rules

Rule	Command	Default Action
.asm.obj	$(AS) $(AFLAGS) /c $*.asm	ML /c $*.ASM
.asm.exe	$(AS) $(AFLAGS) $*.asm	ML $*.ASM
.bas.obj	$(BC) $(BFLAGS) $*.bas;	BC $*.BAS;
.c.obj	$(CC) $(CFLAGS) /c $*.c	CL /c $*.C
.c.exe	$(CC) $(CFLAGS) $*.c	CL $*.C
.cbl.obj	$(COBOL) $(COBFLAGS) $*.cbl;	COBOL $*.CBL;
.cbl.exe	$(COBOL) $(COBFLAGS) $*.cbl, $*.exe;	COBOL $*.CBL, $*.EXE;
.for.obj	$(FOR) /c $(FFLAGS) $*.for	FL /c $*.FOR
.for.exe	$(FOR) $(FFLAGS) $*.for	FL $*.FOR
.pas.obj	$(PASCAL) /c $(PFLAGS) $*.pas	PL /c $*.PAS
.pas.exe	$(PASCAL) $(PFLAGS) $*.pas	PL $*.PAS
.rc.res	$(RC) $(RFLAGS) /r $*	RC /r $*

For example, assume you have the following description file:

```
sample.exe :
```

This description block lists a target without any dependents or commands. NMAKE looks at the target's extension (.EXE) and searches for an inference rule that describes how to create an .EXE file. Table 10.6 shows that more than one inference rule exists for building an .EXE file. NMAKE looks for a file in the current or specified directory that has the same base name as the target `sample` and one of the extensions in the **.SUFFIXES** list. For example, if a file called SAMPLE.FOR exists, NMAKE applies the `.for.exe` inference rule. If more than one file with the base name SAMPLE is found, NMAKE applies the inference rule for the extension listed earliest in the **.SUFFIXES** list. In this example, if both SAMPLE.C and SAMPLE.FOR exist, NMAKE uses the `.c.exe` inference rule to compile SAMPLE.C and links the resulting file SAMPLE.OBJ to create SAMPLE.EXE.

NOTE By default, the options macros such as **CFLAGS** shown in Table 10.5 are undefined. As explained in Section 10.3.4.2, "Using Macros," this causes no problem; NMAKE replaces an undefined macro with a null string. Because the predefined options macros are included in the inference rules, you can define these macros and have their assigned values passed automatically to the predefined inference rules. The predefined inference rules are listed in Table 10.6.

10.3.5.5 Precedence among Inference Rules

If the same inference rule is defined in more than one place, NMAKE uses the rule with the highest precedence. The precedence from highest to lowest is

1. An inference rule defined in the description file. If more than one, the last one applies.
2. An inference rule defined in the TOOLS.INI file. If more than one, the last one applies.
3. A predefined inference rule.

User-defined inference rules always override predefined inference rules. NMAKE uses a predefined inference rule only if no user-defined inference rule exists for a given target and dependent.

If two inference rules could produce a target with the same extension, NMAKE uses the inference rule whose dependent's extension appears first in the **.SUFFIXES** list. See Table 10.7 in the next section, "Directives."

10.3.6 Directives

The directives in Table 10.7 provide additional control of NMAKE operations. You can use them in a description file outside of a description block or in the TOOLS.INI file. The four directives listed in the table are case sensitive and must appear in all uppercase letters. (Preprocessing directives are not case sensitive; see Section 10.3.7, "Preprocessing Directives.")

Table 10.7 Directives

Directive	Action
.IGNORE :	Ignores exit codes returned by programs called from the description file. This directive has the same effect as invoking NMAKE with the /I option.
.PRECIOUS : *target...*	Tells NMAKE not to delete *target*s if the commands that build them quit or are interrupted. Overrides the NMAKE default, which is to delete the target if building was interrupted by CTRL+C or CTRL+BREAK.
.SILENT :	Does not display lines as they are executed. This directive has the same effect as invoking NMAKE with the /S option.
.SUFFIXES : *list*	Lists file suffixes for NMAKE to try when building a target file for which no dependents are specified. This list is used together with inference rules. See Section 10.3.5, "Inference Rules."

The **.IGNORE** and **.SILENT** directives affect the file from their location onward. Location within the file does not matter for the **.PRECIOUS** and **.SUFFIXES** directives; they affect the entire description file.

NMAKE refers to the value of the **.SUFFIXES** directive when using inference rules. When NMAKE finds a target without dependents, it searches the current directory for a file with the same base name as the target and a suffix from *list*. If NMAKE finds such a file, and if an inference rule applies to the file, then NMAKE treats the file as a dependent of the target. The order of the suffixes in the list defines the order in which NMAKE searches for the file. The list is predefined as follows:

```
.SUFFIXES : .exe .obj .asm .c .bas .cbl .for .pas .res .rc
```

To add additional suffixes to the end of the list, specify `.SUFFIXES :` followed by the additional suffixes. To clear the list, specify `.SUFFIXES :` by itself. To change the list order or to specify an entirely new list, clear the list and specify a new `.SUFFIXES :` setting.

10.3.7 Preprocessing Directives

NMAKE preprocessing directives are similar to compiler preprocessing directives. You can use the **!IF**, **!IFDEF**, **!IFNDEF**, **!ELSE**, and **!ENDIF** directives to conditionally process the description file. With other preprocessing directives you can display error messages, include other files, undefine a macro, and turn certain options on or off. NMAKE reads and executes the preprocessing directives before processing the description file as a whole.

Preprocessing directives (listed in Table 10.8) begin with an exclamation point (!), which must appear at the beginning of the line. You can place spaces between the exclamation point and the directive keyword. These directives are not case sensitive.

Table 10.8 Preprocessing Directives

Directive	Description
!CMDSWITCHES {+|-}*opt...*	Turns on or off NMAKE options /D, /I, /N, and /S. (See Section 10.4, "Command-Line Options.") Do not specify the slash (/). If **!CMDSWITCHES** is specified with no options, all options are reset to the values they had when NMAKE was started. This directive updates the **MAKEFLAGS** macro. Turn an option on by preceding it with a plus sign (+), or turn it off by preceding it with a minus sign (–).
!ERROR *text*	Prints *text*, then stops execution.
!IF *constantexpression*	Reads the statements between the **!IF** keyword and the next **!ELSE** or **!ENDIF** keyword if *constantexpression* evaluates to a nonzero value.
!IFDEF *macroname*	Reads the statements between the **!IFDEF** keyword and the next **!ELSE** or **!ENDIF** keyword if *macroname* is defined. NMAKE considers a macro with a null value to be defined.
!IFNDEF *macroname*	Reads the statements between the **!IFNDEF** keyword and the next **!ELSE** or **!ENDIF** keyword if *macroname* is not defined.
!ELSE	Reads the statements between the **!ELSE** and **!ENDIF** keywords if the preceding **!IF**, **!IFDEF**, or **!IFNDEF** statement evaluated to zero. Anything following **!ELSE** on the same line is ignored.
!ENDIF	Marks the end of an **!IF**, **!IFDEF**, or **!IFNDEF** block. Anything following **!ENDIF** on the same line is ignored.

Table 10.8 *(continued)*

Directive	Description
!INCLUDE *filename*	Reads and evaluates the description file *filename* before continuing with the current description file. If *filename* is enclosed by angle brackets (< >), NMAKE searches for the file first in the current directory and then in the directories specified by the **INCLUDE** macro. Otherwise, it looks only in the current directory. The **INCLUDE** macro is initially set to the value of the INCLUDE environment variable.
!UNDEF *macroname*	Marks *macroname* as undefined in NMAKE's symbol table.

10.3.7.1 Expressions in Preprocessing

The *constantexpression* used with the **!IF** directive can consist of integer constants, string constants, or program invocations. Integer constants can use the unary operators for numerical negation (–), one's complement (~), and logical negation (!). They can also use any binary operator listed in Table 10.9.

Table 10.9 Preprocessing-Directive Binary Operators

Operator	Description
+	Addition
–	Subtraction
*	Multiplication
/	Division
%	Modulus
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
&&	Logical AND
||	Logical OR
<<	Left shift
>>	Right shift
==	Equality
!=	Inequality

Table 10.9 *(continued)*

Operator	Description
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

You can group expressions by enclosing them in parentheses. NMAKE treats numbers as decimal unless they start with 0 (octal) or 0x (hexadecimal). Use the equality (==) operator to compare two strings for equality, or the inequality (!=) operator to compare for inequality. Enclose strings in double quotation marks.

Example

The following example shows how preprocessing directives can be used to control whether the linker inserts CodeView information into the .EXE file:

```
!INCLUDE <infrules.txt>
!CMDSWITCHES +D
winner.exe : winner.obj
!IFDEF debug
!   IF "$(debug)"=="y"
       LINK /CO winner.obj;
!  ELSE
       LINK winner.obj;
!   ENDIF
!ELSE
!   ERROR Macro named debug is not defined.
!ENDIF
```

In this example, the **!INCLUDE** directive inserts the INFRULES.TXT file into the description file. The **!CMDSWITCHES** directive sets the /D option, which displays the times of the files as they are checked. The **!IFDEF** directive checks to see if the macro `debug` is defined. If it is defined, the **!IF** directive checks to see if it is set to `y`. If it is, NMAKE reads the LINK command with the /CO option; otherwise, NMAKE reads the LINK command without /CO. If the `debug` macro is not defined, the **!ERROR** directive prints the specified message and NMAKE stops.

10.3.7.2 Executing a Program in Preprocessing

NMAKE can invoke programs and check their status.

You can invoke any program from within NMAKE by placing the program's name or path name within square brackets ([ ]). The program is executed during preprocessing, and its exit code replaces the program specification in the

description file. A nonzero exit code usually indicates an error. You can use this value to control execution, as in the following example:

```
!IF [c:\util\checkdsk] != 0
!    ERROR Not enough disk space; NMAKE terminating.
!ENDIF
```

10.3.8 Extracting Filename Components

"Special Macros," Section 10.3.4.3, showed how qualifiers could be added to macros that represented filenames in order to select components of the name or path. This feature is especially useful when creating a general-purpose description block that works with the name of any dependent.

Besides these macro modifiers, NMAKE offers another feature that allows you to extract components of the name of the first dependent file as you have specified it in the description file or on the command line (not the full filename specification on disk). The components can then be recombined with specific paths, extensions, or directories to create the particular name or path you need, without having to specify the exact name or path when you write the description block.

The first dependent file is the first file listed to the right of the colon on a dependency line. If a dependent is implied from an inference rule, NMAKE considers it to be the first dependent file. If more than one dependent is implied from inference rules, the **.SUFFIXES** list determines which dependent is first.

You can use either of the following syntaxes:

%s

%| [[*parts*]]**F**

where *parts* can be one or more of the following letters, or can be omitted:

Letter	Description
No letter	Complete name
d	Drive
p	Path
f	File base name
e	File extension

You can specify more than one letter. The order of the letters is not significant; NMAKE constructs the filename that meets (or comes closest to meeting) all the specifications. The letters are case sensitive.

The %s option substitutes the complete name; it is equivalent to both %|F and %|dpfeF.

NMAKE interprets any percent symbol (%) within a command line (either in a description block or an inference rule) as the start of a file specifier using this syntax. Therefore, if you need to use a literal percent symbol within a command line, you must specify it as a double percent symbol (%%).

Example

The following example demonstrates this special syntax:

```
sample.exe : c:\project\sample.obj
    LINK %|dpfF, a:%|pfF.exe;
```

This example represents the following command:

```
LINK c:\project\sample, a:\project\sample.exe;
```

In this example, the sequence %|dpfF represents the same drive, path, and base name as the dependent on the dependency line, while the sequence %|pfF represents only the path and base name of the dependent. The command tells the LINK utility to build the executable file on another drive in a directory of the same name.

10.4 Command-Line Options

NMAKE accepts a number of options, listed in Table 10.10. You can specify options in uppercase or lowercase and use either a slash or dash. For example, –A, /A, –a, and /a all represent the same option. This book uses a slash and uppercase letters.

Table 10.10 NMAKE Options

Option	Action
/A	Forces execution of all commands in description blocks in the description file even if targets are not out-of-date with respect to their dependents. Does not affect the behavior of incremental commands such as ILINK; using /A does not force a full link.
/C	Suppresses nonfatal error or warning messages and the NMAKE copyright message.
/D	Displays the modification time of each file.
/E	Causes environment variables to override macro definitions in description files. See Section 10.3.4, "Macros."

Table 10.10 *(continued)*

Option	Action
/F *filename*	Specifies *filename* as the name of the description file. If you supply a dash (–) instead of a filename, NMAKE gets description-file input from the standard input device. (Terminate keyboard input with either F6 or CTRL+Z.) If you omit /F, NMAKE searches the current directory for a file called MAKEFILE and uses it as the description file. If MAKEFILE doesn't exist, NMAKE uses inference rules for the command-line targets.
/HELP	Calls the QuickHelp utility. If NMAKE cannot locate the help file or QuickHelp, it displays a brief summary of NMAKE command-line syntax and exits to the operating system.
/I	Ignores exit codes from commands listed in the description file. NMAKE processes the whole description file even if errors occur.
/N	Displays but does not execute the description file's commands. This option is useful for debugging description files and checking which targets are out-of-date.
/NOLOGO	Suppresses the NMAKE copyright message.
/P	Displays all macro definitions, inference rules, target descriptions, and the **.SUFFIXES** list on the standard output device.
/Q	Checks modification times for command-line targets (or first target in description file if no command-line targets are specified). NMAKE returns a zero exit code if all such targets are up-to-date and a nonzero exit code if any target is out-of-date. Only preprocessing commands in the description file are executed. This option is useful when running NMAKE from a batch file.
/R	Ignores inference rules and macros that are defined in the TOOLS.INI file or that are predefined.
/S	Suppresses the display of commands listed in the description file.
/T	Changes modification times for command-line targets (or first target in description file if no command-line targets are specified). Only preprocessing commands in the description file are executed. Contents of target files are not modified.
/X *filename*	Sends all error output to *filename*, which can be a file or a device. If you supply a dash (–) instead of a filename, error output is sent to the standard output device.
/Z	Used for internal communication between NMAKE (or NMK) and PWB.
/?	Displays a brief summary of NMAKE command-line syntax and exits to the operating system.

Example

The following command line specifies two NMAKE options:

```
NMAKE /F sample.mak /C targ1 targ2
```

The /F option tells NMAKE to read the description file SAMPLE.MAK. The /C option tells NMAKE not to display nonfatal error messages and warnings. The command specifies two targets (`targ1` and `targ2`) to update.

In the following example, NMAKE updates the target `targ1`:

```
NMAKE /D /N targ1
```

Since no description file is specified, NMAKE searches the current directory for a description file named MAKEFILE. The /D option displays the modification time of each file; the /N option displays the commands in MAKEFILE without executing them.

10.5 NMAKE Command File

If you find yourself repeatedly using the same sequence of command-line arguments, you can place them in a text file and pass the file's name as a command-line argument to NMAKE. NMAKE opens the command file and reads the arguments. This feature is especially useful if the argument list exceeds the maximum length of a command line (128 characters in DOS, 256 in OS/2).

To provide input to NMAKE with a command file, type

NMAKE **@***commandfile*

In the *commandfile* field, enter the name of a file containing the information NMAKE expects on the command line. You can split input between the command line and a command file. Use the name of the command file (preceded by **@**) in place of the input information on the command line.

Example 1

Assume you have created a filenamed UPDATE containing this line:

```
/S "program = sample" sort.exe search.exe
```

If you start NMAKE with the command

```
NMAKE @update
```

then NMAKE reads its command-line arguments from UPDATE. The at sign (**@**) tells NMAKE to read arguments from the file. The effect is the same as if you typed the arguments directly on the command line:

```
NMAKE /S "program = sample" sort.exe search.exe
```

NMAKE treats the file as if it were a single set of arguments and replaces each line break with a space. Macro definitions that contain spaces must be enclosed in quotation marks, just as if you had typed them on the command line.

The quotation marks that delimit a macro force all characters between them to be interpreted literally. Therefore, if you split a macro between lines, an unwanted line break is inserted into the macro. Macros that span multiple lines must be continued by ending each line except the last with a backslash (\ ):

```
/S "program \
= sample" sort.exe search.exe
```

This file is equivalent to the first example. The backslash allows the macro definition ("`program = sample`") to span two lines.

Example 2

If the command-file UPDATE contains this line:

```
/S "program = sample" sort.exe
```

you can give NMAKE the same command-line input as in the example above by specifying the command

```
NMAKE @update search.exe
```

10.6 The TOOLS.INI File

You can customize NMAKE by placing commonly used macros, inference rules, and description blocks in the TOOLS.INI initialization file. Settings for NMAKE must follow a line that begins with [NMAKE]. This section of the initialization file can contain macro definitions, **.SUFFIXES** lists, and inference rules. For example, if TOOLS.INI contains the following section:

```
[NMAKE]
CC=qcl
CFLAGS=/Gc /Gs /W3 /Oat
.c.obj:
    $(CC) /c $(CFLAGS) $*.c
```

NMAKE reads and applies the lines following `[NMAKE]`. The example redefines the macro **CC** to invoke the Microsoft QuickC® Compiler, defines the macro **CFLAGS**, and redefines the inference rule for making .OBJ files from .C sources. (Note that macros are case sensitive; a macro called **cc** is not substituted in a rule that uses **$(CC)**.)

NMAKE looks for TOOLS.INI in the current directory. If it isn't there, NMAKE searches the directory specified by the INIT environment variable.

Macros and inference rules appearing in TOOLS.INI can be overridden. See Section 10.3.4.7, "Precedence among Macro Definitions," and Section 10.3.5.5, "Precedence among Inference Rules."

10.7 Inline Files

NMAKE can create "inline files" which contain any text you specify. One use of inline files is to write a response file for another utility such as LINK or LIB. This eliminates the need to maintain a separate response file and removes the restraint on the maximum length of a command line.

Use this syntax to create an inline file called *filename*:

target : *dependents*
  command << [[*filename*]]
inlinetext
.
.
.
<<[[**KEEP** | **NOKEEP**]]

All *inlinetext* between the two sets of double angle brackets (<<) is placed in the inline file. The *filename* is optional. If you don't supply *filename*, NMAKE gives the inline file a unique name. NMAKE places the inline file in the directory specified by the TMP environment variable. If TMP is not defined, the inline file is placed in the current directory.

Directives are not allowed in an inline file. NMAKE treats a directive in an inline file as literal text.

The inline file can be temporary or permanent. If you don't specify the option, or if you specify **NOKEEP**, the file is temporary. Specify **KEEP** to retain the file after the build ends.

Example

The following description block creates a LIB response file named LIB.LRF:

```
OBJECTS=add.obj sub.obj mul.obj div.obj
math.lib : $(OBJECTS)
    LIB @<<lib.lrf
$*.lib
-+$(OBJECTS: = &^
-+)
listing;
<<KEEP
```

The resulting response file tells LIB which library to use, the commands to execute, and the name of the listing file to produce:

```
math.lib
-+add.obj &
-+sub.obj &
-+mul.obj &
-+div.obj
listing;
```

The file MATH.LIB must exist beforehand for this example to work.

Multiple Inline Files

The inline file specification can create more than one inline file. For instance,

```
target.abc : depend.xyz
    cat <<file1 <<file2
I am the contents of file1.
<<KEEP
I am the contents of file2.
<<KEEP
```

The example creates the two inline files, FILE1 and FILE2. All inline text is written to the files sequentially. Therefore, the text

```
I am the contents of file1.
```

goes into FILE1, not FILE2, even though the text is nested between the angle brackets for FILE2 and the `<<KEEP` statement which follows. NMAKE then executes the command

```
cat file1 file2
```

The **KEEP** keywords tell NMAKE not to delete FILE1 and FILE2 when done.

10.8 Sequence of NMAKE Operations

When you are writing a complex description file, it can be helpful to know the sequence in which NMAKE performs operations. This section describes those operations and their order.

NMAKE first looks for a description file.

When you run NMAKE from the command line, NMAKE's first task is to find the description file:

1. If the /F option is used, NMAKE searches for the filename specified in the option. If NMAKE cannot find that file, it returns an error.
2. If the /F option is not used, NMAKE looks for a file named MAKEFILE in the current directory. If there are targets on the command line, NMAKE

builds them according to the instructions in MAKEFILE. If there are no targets on the command line, NMAKE builds only the first target it finds in MAKEFILE.

3. If NMAKE cannot find MAKEFILE, NMAKE looks for target files on the command line and attempts to build them using inference rules (either defined by the user in TOOLS.INI or predefined by NMAKE). If no target is specified, NMAKE returns an error.

Macro definitions follow a priority.

NMAKE then assigns macro definitions with the following precedence (highest first):

1. Macros defined on the command line
2. Macros defined in a description file or include file
3. Inherited macros
4. Macros defined in the TOOLS.INI file
5. Predefined macros (such as **CC** and **RFLAGS**)

Macro definitions are assigned in order of priority, not in the order in which NMAKE encounters them. For example, a macro defined in an include file overrides a macro with the same name from the TOOLS.INI file. Note that a macro within a description file can be redefined; the most recent definition in the description file is used.

Inference rules also follow a priority.

NMAKE also assigns inference rules, using the following precedence (highest first):

1. Inference rules defined in a description file or include file
2. Inference rules defined in the TOOLS.INI file
3. Predefined inference rules (such as .c.obj)

You can use command-line options to change some of these precedences.

- The /E option allows macros inherited from the environment to override macros defined in the description file.
- The /R option tells NMAKE to ignore macros and inference rules that are defined in TOOLS.INI or are predefined.

NMAKE preprocesses directives before running the description-file commands.

Next, NMAKE evaluates any preprocessing directives. If an expression for conditional preprocessing contains a program in square brackets ([ ]), the program is invoked during preprocessing, and the program's exit code is used in the expression. If an **!INCLUDE** directive is specified for a file, NMAKE preprocesses the

included file before continuing to preprocess the rest of the description file. Preprocessing determines the final description file that NMAKE reads.

NMAKE updates targets in the description file.

NMAKE is now ready to update the targets. If you specified targets on the command line, NMAKE updates only those targets. If you did not specify targets on the command line, NMAKE updates just the first target it finds in the description file. (This behavior differs from the MAKE utility's default; see Section 10.10, "Differences between NMAKE and MAKE.") If you specify a pseudotarget, NMAKE always updates the target. If you use the /A option, NMAKE always updates the target, even if the file is not out-of-date.

If the dependents of the targets are themselves out-of-date or do not exist yet, NMAKE updates them first. If the target has no explicit dependent, NMAKE looks in the current directory for one or more files with the same base name as the target and whose extensions are in the **.SUFFIXES** list. (See Section 10.3.6, "Directives," for a description of the **.SUFFIXES** list.) If it finds such files, NMAKE treats them as dependents and updates the target according to the commands.

Errors usually stop the build.

NMAKE normally stops processing the description file when a command returns a nonzero exit code. In addition, if NMAKE cannot tell whether the target was built successfully, it deletes the target. If you use the /I command-line option, NMAKE ignores error codes and attempts to continue processing. The **.IGNORE** directive has the same effect as the /I option. To prevent NMAKE from deleting the partially created target if you interrupt the build with CTRL+C or CTRL+BREAK, specify the target name in the **.PRECIOUS** directive.

Alternatively, you can use the dash (–) command modifier to ignore the error code for an individual command. An optional number after the dash tells NMAKE to continue if the command returns an exit code that is less than or equal to the number, and to stop if the exit code is greater than the number.

You can document errors by using the **!ERROR** directive to print descriptive text. The directive causes NMAKE to print some text, then stop, even if you use /I, **.IGNORE**, or the dash (–) modifier.

10.9 A Sample NMAKE Description File

The following example illustrates many of NMAKE's features. The description file creates an executable file from C-language source files:

```
#  This description file builds SAMPLE.EXE from SAMPLE.C,
#  ONE.C, and TWO.C, then deletes intermediate files.

CFLAGS   = /c /AL /Od $(CODEVIEW)  # controls compiler options
LFLAGS   = /CO                     # controls linker options
CODEVIEW = /Zi                     # controls CodeView data

OBJS = sample.obj one.obj two.obj
```

```
all : sample.exe

sample.exe : $(OBJS)
    link $(LFLAGS) @<<sample.lrf
$(OBJS: =+^
)
sample.exe
sample.map;
<<KEEP

sample.obj : sample.c sample.h common.h
     CL $(CFLAGS) sample.c

one.obj : one.c one.h common.h
     CL $(CFLAGS) one.c

two.obj : two.c two.h common.h
     CL $(CFLAGS) two.c

clean :
    -del *.obj
    -del *.map
    -del *.lrf
```

Assume that this description file is named SAMPLE.MAK. To invoke it, enter

```
NMAKE /F SAMPLE.MAK all clean
```

NMAKE then builds SAMPLE.EXE and deletes intermediate files.

Here is how the description file works. The **CFLAGS**, **CODEVIEW**, and **LFLAGS** macros define the default options for the compiler, linker, and inclusion of CodeView information. You can redefine these options from the command line to alter or delete them. For example,

```
NMAKE /F SAMPLE.MAK CODEVIEW= CFLAGS= all clean
```

creates an .EXE file that does not contain CodeView information.

The **OBJS** macro specifies the object files that make up SAMPLE.EXE, so they can be reused without having to type them again. Their names are separated by exactly one space so that the space can be replaced with a plus sign (+) and a carriage return in the link response file. (This is illustrated in the second example in Section 10.3.4.4, "Substitution within Macros.")

The `all` pseudotarget points to the real target, `SAMPLE.EXE`. If you do not specify any target on the command line, NMAKE ignores the `clean` pseudotarget but still builds `all`, since `all` is the first target in the description file.

The dependency line containing the target `sample.exe` makes the object files specified in **OBJS** the dependents of SAMPLE.EXE. The command section of the block contains only link instructions. No compilation instructions are given,

since they are given explicitly later in the file. (You could also define an inference rule to specify how an object file is to be created from a C source file.)

The link command is unusual in that the link parameters and options are not passed directly to LINK. Rather, an inline response file is created containing these elements. This eliminates the need to maintain a separate link response file. It also allows the LINK command line to exceed the normal limit on the length of a command line (128 characters in DOS, 256 characters in OS/2).

The next three dependencies define the relationship of the source code to the object files. The .H (header or include) files are also dependents, since any changes to them would require recompilation.

The `clean` pseudotarget deletes unneeded files after a build. The dash modifier (–) tells NMAKE to ignore errors returned by the deletion commands. If you want to save any of these files, don't specify `clean` on the command line; NMAKE then ignores the `clean` pseudotarget.

10.10 Differences between NMAKE and MAKE

NMAKE replaces the Microsoft MAKE program. NMAKE differs from MAKE in the following ways:

- NMAKE does not evaluate targets sequentially. Instead, NMAKE updates the targets you specify when you invoke it, regardless of their positions in the description file. If no targets are specified, NMAKE updates only the first target in the file.
- NMAKE requires a special syntax when specifying a target in more than one dependency line. (See Section 10.3.1.8, "Specifying a Target in Multiple Description Blocks.")
- NMAKE accepts command-line arguments from a file.
- NMAKE provides more command-line options.
- NMAKE provides more predefined macros.
- NMAKE permits substitutions within macros.
- NMAKE supports directives placed in the description file.
- NMAKE allows you to specify include files in the description file.

The first item in the list deserves special emphasis. While MAKE updates every target, working from beginning to end of the description file, NMAKE expects you to specify targets on the command line. If you do not, NMAKE builds only the first target in the description file.

This difference is clear if you run NMAKE using a typical MAKE description file, which lists a series of subordinate targets followed by a higher-level target that depends on the following subordinates:

```
pmapp.obj : pmapp.c
    CL /c /G2sw /W3 pmapp.c

pmapp.exe : pmapp.obj pmapp.def
    LINK pmapp, /align:16, NUL, os2, pmapp
```

MAKE builds both targets (PMAPP.OBJ and PMAPP.EXE), but NMAKE builds only the first target (PMAPP.OBJ).

Because of these performance differences, you may want to convert MAKE files to NMAKE files. MAKE description files are easy to convert. One way is to create a new description block at the beginning of the file. Give this block a pseudo-target named `all` and list the top-level target as a dependent of `all`. To build `all`, NMAKE must update every file upon which the target `all` depends:

```
all : pmapp.exe

pmapp.obj : pmapp.c
    CL /c /G2sw /W3 pmapp.c

pmapp.exe : pmapp.obj pmapp.def
    LINK pmapp, /align:16, NUL, os2, pmapp
```

If the above file is named MAKEFILE, you can update the target PMAPP.EXE with the command

```
NMAKE
```

or the command

```
NMAKE all
```

It is not necessary to list PMAPP.OBJ as a dependent of `all`. NMAKE builds a dependency tree for the entire description file and builds whatever files are needed to update PMAPP.EXE. If PMAPP.C has a later modification time than PMAPP.OBJ, NMAKE compiles PMAPP.C to create PMAPP.OBJ, then links PMAPP.OBJ to create PMAPP.EXE.

The same technique is suitable for description files with more than one top-level target. List all the top-level targets as dependents of `all`:

```
all : pmapp.exe second.exe another.exe
```

The example updates the targets PMAPP.EXE, SECOND.EXE, and ANOTHER.EXE.

If the description file lists a single, top-level target, you can use an even simpler technique. Move the top-level block to the beginning of the file:

```
pmapp.exe : pmapp.obj pmapp.def
    LINK pmapp, /align:16, NUL, os2, pmapp
pmapp.obj : pmapp.c
    CL /c /G2sw /W3 pmapp.c
```

NMAKE updates the second target (PMAPP.OBJ) whenever needed to keep the first target (PMAPP.EXE) current.

10.11 Using NMK

When you maintain a project under DOS or in a DOS session under OS/2, you will probably need to use the NMK utility. NMK uses only 5K of memory, leaving room for the programs called during the build. You run NMK the same way you run NMAKE, using the same command-line syntax and the same description-file syntax. NMK calls NMAKE to read the description file and perform the build.

The behavior of NMK is slightly different from that of NMAKE. The fundamental difference is that NMAKE rechecks the update status of all files after each build step, whereas NMK checks file status only once, at the start of the build process. If your description file simply compiles a series of files and then links them, this difference never causes a problem. But consider the following example, which uses a pseudotarget to clean up old files during the build:

```
all : clean example.exe

example.exe : example.asm
    ML example

clean :
    del example.obj
    del example.exe
```

This description file erases EXAMPLE.OBJ and EXAMPLE.EXE, then recompiles. Under NMAKE, it works as intended; that is, it

1. Erases files
2. Checks the status of EXAMPLE.EXE
3. Rebuilds EXAMPLE.EXE because EXAMPLE.EXE is no longer present

However, NMK checks the status of the environment only at the beginning of the build. Since EXAMPLE.EXE exists when the build starts, the preceding description file

1. Erases files
2. Stops execution, because EXAMPLE.EXE was present and up-to-date at the beginning of the process

PWB never generates a description file that requires dynamic status checking to run correctly, so you can use PWB-created description files with either NMAKE or NMK.

10.12 Using Exit Codes with NMAKE

NMAKE stops execution if a program executed by one of the commands in the NMAKE description file encounters an error. The exit code returned by the program is displayed as part of the error message.

Assume the NMAKE description file `TEST` contains the following lines:

```
TEST.OBJ : TEST.FOR
    FL /c TEST.FOR
```

If the source code in `TEST.FOR` causes an error (but not a warning), you would see the following message the first time you use NMAKE with the NMAKE description file `TEST`:

```
NMAKE : fatal error U1077: 'FL /c TEST.FOR' - return code '2'
```

This error message indicates that the command `FL /c TEST.FOR` in the NMAKE description file returned exit code 2.

You can cause NMAKE to ignore an exit code for a command by preceding the command with a dash modifier (–). If you specify a number after the dash modifier (*–n*), NMAKE stops only if the exit code is greater than the specified number. (See Table 10.1.) You disable this behavior for the entire description file by invoking NMAKE with the /I option.

You can also test exit codes in NMAKE description files with the **!IF** preprocessing directive. See Section 10.3.7.2, "Executing a Program in Preprocessing."

If you prefer to use DOS batch files instead of NMAKE description files, you can test the code returned with the IF command. See a DOS manual for more information.

NMAKE returns an exit code to the operating system or the calling program. A value of 0 indicates execution of NMAKE with no errors. Warnings return exit code 0.

Code	Meaning
0	No error
2	Program error
4	System error—out of memory

10.13 Related Topics in Online Help

In addition to information covered in this chapter, information on the following topics can be found in online help.

Topics	Access
Syntax and procedural information on NMAKE	From the list of Utilities on the "Microsoft Advisor Contents" screen, choose "NMAKE"
Using TOOLS.INI	From the "Microsoft Advisor Contents" screen, choose "Programmer's WorkBench"; then choose "Using TOOLS.INI" from the list of topics relating to customizing PWB

Chapter 11

Creating Help Files with HELPMAKE

If you've used the Programmer's WorkBench (PWB) or one of the Microsoft Quick languages, you already know the advantages of online help, or the Microsoft Advisor. The Microsoft Help File Maintenance utility (HELPMAKE) lets you extend these advantages by customizing the help files supplied with Microsoft language products, or by creating your own help files for them.

HELPMAKE translates help text files into a help database accessible within these environments:

- Microsoft Programmer's WorkBench (PWB)
- Microsoft QuickHelp utility
- Microsoft CodeView debugger
- Microsoft Editor version 1.02
- Microsoft QuickC compiler versions 2.0 and later
- Microsoft QuickBasic™ versions 4.5 and later
- Microsoft QuickPascal™ version 1.0
- Microsoft Word version 5.5

This chapter describes how to create and modify help files using the HELPMAKE utility.

11.1 Structure and Contents of a Help Database

HELPMAKE creates a help database from one or more input files that contain information formatted for the help system. This section defines some of the terms involved in formatting and outlines the formats that HELPMAKE can process.

11.1.1 Contents of a Help File

Each help input file consists of one or more help "topics." A topic is the fundamental unit of help information. It is usually a screenful of information about a particular subject. You identify the subject by one or more "context strings," which are the words and phrases for which you want to be able to request help. When help is requested on a context string, the topic is displayed.

The **.context** command defines a context string for the topic that follows it. In the source file for C help, for example, this line introduces help for the **#include** directive:

```
.context #include
```

The **.context** command and other formatting elements are described in Section 11.5, "Help Text Conventions."

Whether a context string contains one word or several words depends on the application. For example, because Microsoft QuickBasic considers spaces to be delimiters, a context string in QuickBasic help files is limited to a single word. Other applications, such as PWB, can handle context strings that span several words. In either case, the application hands the context string to an internal "help engine" that searches the database for information.

Often, especially with library routines, the same information applies to more than one subject. For example, the C-language string-to-number functions **strtod**, **strtol**, and **strtoul** share the same help text. The help file lists all three function names as contexts for one block of topic text. The converse, however, is not true. You cannot associate a single context string with several blocks of topic text located at different places in the help file.

Cross-references help you navigate a help database.

Cross-references make it possible to view information about related topics, including header files and code examples. The help for the C-language **open** function, for example, references the **access** function. Cross-references can point to other contexts in the same help database, to contexts in other help databases, or even to ASCII files outside the database.

Help files can have two kinds of cross-references:

- Implicit
- Explicit, or hyperlinks

Implicit cross-references are coded with an ordinary .context command.

The word "open" is an implicit cross-reference throughout Microsoft C help, and introduces help for the **open** function. If you select the word "open" anywhere in C help, the help system displays information on the **open** function. The context for **open** begins with an ordinary **.context** command. As a result, anywhere that you select "open," the help system references this context.

Hyperlinks are explicit cross-references marked by invisible text.

A "hyperlink" is an explicit cross-reference tied to a word or phrase at a specific location in the help file. You create hyperlinks when you write the help text. The hyperlink consists of a word or phrase followed by invisible text that gives the context to which the hyperlink refers.

For example, to cause an instance of the word "formatting" to display help on the **printf** function, you would create an explicit cross-reference from the word "formatting" to the context "printf." Elsewhere in the file, "formatting" has no special significance, but at that one position, it references the help for **printf**. For details on how to create hyperlinks, see Section 11.5.4.

Formatting flags let you change the appearance of text.

Help text can also include formatting attributes to control the appearance of the text on the screen. Using these attributes, you can make certain words appear in various colors, inverse video, and so forth, depending on the application displaying help and the graphics capabilities of your computer.

11.1.2 Help File Formats

You can create sources for help text files in any of three formats:

- QuickHelp format
- Rich Text Format (RTF)
- Minimally formatted ASCII

In addition, you can reference unformatted ASCII files, such as include files, from within a help database.

An entire help system (such as the ones supplied with Microsoft C, FORTRAN, MASM, or QuickBasic) can use any combination of files formatted with different format types. With C, for example, the README.DOC information file is encoded as minimally formatted ASCII; the help files for the PWB, C language, and run-time library are written in QuickHelp format before being compressed by HELPMAKE. The database also cross-references the header (include) files, which are unformatted ASCII files stored outside the database.

QuickHelp

QuickHelp format is the default format into which HELPMAKE decodes help databases. Any text editor can create a QuickHelp-format help text file. QuickHelp format also lends itself to a relatively easy automated translation from other document formats.

QuickHelp files can contain any kind of cross-reference or formatting attribute. Typically, you use QuickHelp format when modifying a Microsoft-supplied database.

QuickHelp format makes use of dot commands (such as **.context**—see the description of QuickHelp dot commands in Section 11.6.1). To use dot commands other than **.context** and **.comment**, the /T option is required for encoding and decoding. For details, see Section 11.3, "Helpmake Options."

Rich Text Format

Rich Text Format (RTF) is a Microsoft word-processing format that several word processors support, including Microsoft Word version 5.0 and later, and Microsoft Word for Windows. You can use RTF as an intermediate format to simplify transferring help files from one format to another. Like QuickHelp files, RTF files can contain formatting attributes and cross-references.

An RTF word processor provides the easiest way to create an RTF file, but you can manually insert RTF codes with an ordinary text editor. There are also utility programs that convert text files in other formats to RTF format.

See Section 11.6.2, "Rich Text Format," for more information.

Minimally Formatted ASCII

Minimally formatted ASCII files define contexts and their topic text; they cannot contain screen-formatting commands or explicit cross-references. (Implicit cross-references work the same way they do in the other formats.) Minimally formatted ASCII files are often used to display text in a README.DOC or small help files that do not require compression. See Section 11.6.3, "Minimally Formatted ASCII Format," for more information.

Unformatted ASCII

Unformatted ASCII files are exactly what their name implies: regular ASCII files with no formatting commands, context definitions, or special information. HELPMAKE does not process unformatted ASCII files in any special way. An unformatted ASCII file does not become part of the help database; only its name is used as the object of a cross-reference. Unformatted ASCII files are useful for storing program examples. Any word that is an implicit cross-reference in other help files is also an implicit cross-reference in unformatted ASCII files.

11.2 Invoking HELPMAKE

The HELPMAKE program can encode to create new help files or decode to modify existing ones. Encoding converts a text file to a compressed help database. HELPMAKE can encode text files written in QuickHelp, RTF, and minimally formatted ASCII format. Decoding converts a help database to a text file for editing. Regardless of the source format, HELPMAKE always decodes a help database into a QuickHelp-format text file.

You invoke HELPMAKE with the following syntax:

HELPMAKE {/E[[*n*]] | /D[[*c*]] | / H| /?} [[*options*]] *sourcefiles*

The *options* modify the action of HELPMAKE; they are described in Section 11.3, "HELPMAKE Options."

You must supply either the /E (encode) or the /D (decode) option. When encoding, you must also use the /O option to specify the file name of the database.

The *sourcefiles* field is required. It specifies the input file(s) for HELPMAKE. If you use the /D (decode) option, *sourcefiles* can be one or more help database files (such as PWB.HLP). HELPMAKE decodes the database files to the standard output device. If you use the /E (encode) option, *sourcefiles* can be one or more help text files (such as PWB.SRC). File names are separated with a space. You can use standard wild-card characters to specify a group of related files.

The example below invokes HELPMAKE with the /V, /E, and /O options (see Section 11.3.1, "Options for Encoding"). HELPMAKE reads input from the text file `my.txt` and writes the compressed help database in the file `my.hlp`. The /E option, without a compression specification, maximizes compression. Note that the DOS or OS/2 redirection symbol (>) sends a log of HELPMAKE activity to the file `my.log`. You may want to redirect the log file because, in its verbose mode (given by /V), HELPMAKE can generate a lengthy log.

```
HELPMAKE /V /E /Omy.hlp my.txt > my.log
```

The example below invokes HELPMAKE to decode the help database `my.hlp` into the text file `my.src`, given with the /O option. Once again, the /V option results in verbose output, and the output is directed to the log file `my.log`. Section 11.3.2 describes additional options for decoding.

```
HELPMAKE /V /D /Omy.src my.hlp > my.log
```

11.3 HELPMAKE Options

HELPMAKE accepts the command-line options described below. You can specify options in uppercase or lowercase letters and precede them with either a forward slash (/) or a dash (–). Most options apply only to encoding, others apply only to decoding, and a few apply to both. The /T option is required if you want to use dot commands with the QuickHelp format (which is the default format).

11.3.1 Options for Encoding

When you encode a file—that is, when you build a help database—you must specify the /E option. HELPMAKE also accepts other options to control encoding. The encoding options are listed below:

Option	Action
/A*c*	Specifies *c* as an application-specific control character for the help database file. The character marks a line that contains special information for internal use by the application. For example, the Microsoft Advisor uses the colon (:).
/C	Makes context strings for this help file case sensitive.
/E[[*n*]]	Creates (encodes) a help database from a specified text file. The *n* specifies the type(s) of compression. If *n* is omitted, HELPMAKE compresses the file as much as possible (about 50%). The value of *n* is in the range 0–15. It is the sum of successive integral powers of 2 representing various compression techniques:

Value	Technique
0	No compression
1	Run-length compression
2	Keyword compression
4	Extended keyword compression
8	Huffman compression

Add values to combine compression techniques. For example, use /E3 to get run-length and keyword compression. Use /E0 in the testing stages of help database creation where you need to create the database quickly and are not yet concerned with size.

/K*filename*	Optimizes keyword compression by supplying a list of characters that act as word separators. The *filename* is a file containing your list of separator characters.

Option	Action
	The /E2 and /E3 options tell HELPMAKE to identify "keywords"—words occurring often enough to justify replacing them with shorter character sequences. A word is any series of characters that do not appear in the separator list. The default separator list includes all ASCII characters from 0 to 32, ASCII character 127, and the following characters: ! " # & ‘ ’ ( ) * + – , / : ; < = > ? @ [\ ] ^ _ { | } ~ You can improve keyword compression by designing a separator list tailored to a specific help file. If your help file contains **#include** directives, **#include** is encoded (by default) as **include**. To encode **#include** as a keyword, create a separator list that omits the #: ! " & ‘ ’ ( ) * + – , / : ; < = > ? @ [\ ] ^ _ { | } ~ Characters in the range 0–31 are always separators, so you need not include them. A customized list must include all other separators, however, including the space (which follows **!** in the list above). If you omit the space, HELPMAKE encodes sequences of words as keywords.
/L	Locks the generated file so that it cannot later be decoded.
/NOLOGO	Suppresses the HELPMAKE copyright message.
/O*outfile*	Specifies *outfile* as the name of the help database.
/S*n*	Specifies the type of input file, according to the following *n* values: **Option** — **File Type** /S1 — Rich Text Format (RTF) /S2 — QuickHelp (default) /S3 — Minimally formatted ASCII
/T	Translates dot commands into internal format. If your help file contains dot commands other than **.context** and **.comment**, you must supply this option when encoding it. Dot commands are described in Section 11.6.1,"QuickHelp Format," and in later sections. The /T option causes the option /A: to be assumed.

Option	Action
/V[[*n*]]	Controls verbosity of diagnostic and informational output. Larger values of *n* add more information. Omitting *n* produces a full listing. The values of *n* are listed below:

Option	Output
/V	Maximum diagnostic output
/V0	No diagnostic output and no banner
/V1	HELPMAKE banner only
/V2	Pass names
/V3	Contexts on first pass
/V4	Contexts on each pass
/V5	Any intermediate steps within each pass
/V6	Statistics on help file and compression

/W*width*	Indicates the fixed width of the resulting help text in number of characters. The value of *width* can range from 11 to 255. If the /W option is omitted, the default is 76. When encoding an RTF source (/S1), HELPMAKE automatically formats the text to *width*. When encoding QuickHelp (/S2) or minimally formatted ASCII (/S3) files, HELPMAKE truncates lines to this width.

11.3.2 Options for Decoding

The /D option decodes a help database into QuickHelp files. HELPMAKE also accepts other options to control decoding. The decoding options are listed below:

Option	Action
/D[[*c*]]	Decodes the input file into its original text or component parts. If a destination file is not specified with the /O option, the help file is decoded to the standard output device. The form of decoding is controlled by the form of /D[[*c*]] specified:

Form	Effect
/D	Fully decodes the help database, leaving all cross-references and formatting information intact.
/DS	Splits a concatenated help database into its components using their original names. If the database was not created by concatenation, HELPMAKE copies it to a file with its original name. The database is not decompressed.
/DU	Decompresses the database and removes all screen formatting and cross-references. The output can be used later for input and recompression, but all screen formatting and cross-references are lost.
/NOLOGO	Suppresses the HELPMAKE copyright message.

/O[[*outfile*]]	Specifies *outfile* for the decoded output from HELPMAKE. If *outfile* is omitted, the help database is decoded to the standard output device. HELPMAKE always decodes help database files into QuickHelp format.
/T	Translates dot commands from internal format into dot-command format. You must always supply this option when decoding a help database that contains dot commands other than **.context** and **.comment**.

Option	Action
/V[[*n*]]	Controls verbosity of diagnostic and informational output. Larger values of *n* add more information. Omitting *n* produces a full listing. The values of *n* are listed below:

Option	Output
/V	Maximum diagnostic output
/V0	No diagnostic output and no banner
/V1	HELPMAKE banner only
/V2	Pass names
/V3	Contexts on first pass

11.3.3 Options for Help

The following are the options for help.

Option	Action
/?	Displays a brief summary of HELPMAKE command-line syntax and exits without encoding or decoding any files. All other information on the command line is ignored.
/[[HELP]]	Calls the QuickHelp utility and displays help about HELPMAKE. If HELPMAKE cannot find QuickHelp or the help file, it displays the same information as with the /? option. No files are encoded or decoded. All other information on the command line is ignored.

11.4 Creating a Help Database

There are two ways to create a Microsoft-compatible help database.

The first method is to decompress an existing help database, modify the resulting help text file, and recompress the help text file to form a new database.

The second method is to append a new help database to an existing help database. This method involves the following steps:

1. Create a help text file in QuickHelp format, RTF, or minimally formatted ASCII.
2. Use HELPMAKE to create a help database file. The example below invokes HELPMAKE, using `yourhelp.txt` as the input file and producing a help database file named `yourhelp.hlp`:

   ```
   HELPMAKE /V /E /Oyourhelp.hlp yourhelp.txt > yourhelp.log
   ```

3. Back up the existing database.
4. Append the new help database file to the existing database. The example below appends the new database `yourhelp.hlp` to the `alang.hlp` database. (In the example, the /b modifier for the DOS COPY command combines the files as binary files.)

   ```
   COPY alang.hlp /b + yourhelp.hlp /b
   ```

5. Test the database. Assume `yourhelp.hlp` contains the context `sample`. If you type `sample` in PWB and request help on it, the help window should display the text associated with the context `sample`.

WARNING The PWB editor truncates lines longer than about 250 characters. Some databases contain lines longer than this. To edit or create database files with extremely long lines, you must either use an editor (such as Microsoft Word) that does not restrict line length, or extend long lines using the backslash (\) line-continuation character.

11.5 Help Text Conventions

The source text that HELPMAKE uses to create Microsoft help databases must follow specific organizational conventions. The following sections explain these conventions.

11.5.1 Structure of the Help Text File

The Microsoft help system is simply a data-retrieval tool. It imposes no restrictions on the content or organization of help data. However, the HELPMAKE utility and the data-display routines in the help system expect a help file to follow a standard format. This section explains how to create correctly formatted help text files.

In all three help text formats, the help text source file is a sequence of topics, each preceded by one or more context definitions. The following table lists the various formats and the corresponding context definition statements:

Format	Context Definition
QuickHelp	**.context** *context*
RTF	\par >>*context* \par
Minimally formatted ASCII	>>*context*
Unformatted ASCII	None

In QuickHelp format, each topic begins with one or more **.context** statements. These statements link the *context* string to its topic text. The topic text consists of all subsequent lines up to the next **.context** statement.

In RTF format, each context definition must be in a paragraph of its own (denoted by \par), beginning with the help delimiter (>>). As in QuickHelp, the topic text consists of all subsequent paragraphs up to the next context definition.

In minimally formatted ASCII, each context definition must be on a separate line, and each must begin with the help delimiter (>>). As in RTF and QuickHelp files, all subsequent lines up to the next context definition constitute the topic text.

See Section 11.6, "Using Help Database Formats," for detailed information about these three formats.

WARNING HELPMAKE warns you if it encounters a duplicate context string definition within a given help source file. Each context string must be unique.

11.5.2 Local Contexts

Context strings beginning with the "at" sign (@) are "local." Making a context local saves file space and speeds access. However, local contexts cannot be cross-referenced with an implicit link, and they have no meaning outside the local file.

When you use a local context, HELPMAKE does not generate a global context string (a context string that is known throughout the help system). Instead, it embeds an encoded cross-reference that has meaning only within the current context. For example,

```
.context normal
This is a normal topic, accessible by the context string "normal".
[button\v@local\v] is a cross-reference to the following topic.

.context @local

This topic can be reached only by the explicit cross-reference
in the previous topic (or by browsing the file sequentially).
```

In the example above, the text `button\v@local\v` references `local` as a local context. If the user selects the text `button` or scrolls through the file, the help system displays the topic text that follows the context definition for `local`. Because `local` is defined with the "at" sign @, it can be accessed only by a hyperlink within the same help file or by sequentially browsing the file.

If you want a topic to be accessible in both local and global contexts, you simply mark the topic text with both global and local **.context** statements. For example, to make `topic` both global and local, add the following statements:

```
.context topic
.context @topic
```

Naturally, both **.context** statements must appear immediately before the topic text to which they point.

To create a context that begins with a literal @, precede it with a backslash (\ ).

11.5.3 Context Prefixes

Microsoft help databases use several "context prefixes." A context prefix is a single letter followed by a period. It appears before a context string with a predefined meaning. These contexts may appear in the resulting text file when you decode a Microsoft help database.

Context prefixes are used internally by Microsoft.

Except for the h. prefix described below, the context prefixes are used by Microsoft to mark environment- or product-specific features. You would not normally add them to the help files you write.

You can use the h. prefix to identify standard help-file contexts. For instance, h.default identifies the default help screen (the screen that normally appears when you select top-level help). Table 11.1 lists the standard h. contexts.

Table 11.1 Standard h. Contexts

Context	Description
h.contents	The table of contents for the help file. You should also define the string "contents" for direct reference to this context.
h.default	The default help screen, typically displayed when the user presses SHIFT+F1 at the "top level" in some applications.
h.index	The index for the help file. You can also define the string "index" for direct reference to this context.
h.notfound	The help text displayed by some applications when the help system cannot find information about the requested context. The text could be an index of contexts, a topical list, or general information about using help.
h.pg#	A specific page within the help file. This is used in response to a "go to page #" request.
h.pg$	The help text that is logically last in the file. This is used by some applications in response to a "go to the end" request made within the help window.
h.pg1	The help text that is logically first in the file. This is used by some applications in response to a "go to the beginning" request made within the help window.
h.title	The title of the help database.

The context prefixes in Table 11.2 are internal to Microsoft products. They appear in decompressed databases, but you do not need to use them.

Table 11.2 Microsoft Product Context Prefixes

Prefix	Purpose
d.	Dialog box. Each dialog box is assigned a number. Its help context string is d. followed by the number (for example, `d.12`).
e.	Error number. If a product supports the error-numbering scheme used by Microsoft languages, it displays help for each error using this prefix. For example, the context `e.P0105` refers to the Microsoft QuickPascal Compiler error message number P0105.
h.	Help item. Prefixes miscellaneous help context strings that may be constructed or otherwise hidden from the user. For example, most applications look for the context string h.contents when Contents is chosen from the Help menu.
m.	Menu item. Contexts that relate to product menu items are defined by their shortcut keys. For example, the Exit selection on the File menu item is accessed by ALT+F, X and is referenced in help by `m.f.x`.
n.	Message number. Each message box is assigned a number. Its help context string is n. plus the number (for example, `n.5`).

11.5.4 Hyperlinks

Explicit cross-references, or hyperlinks, are marked with invisible text in the help text file. A hyperlink is a word or phrase followed by invisible text that names the context to which the hyperlink refers.

The keystroke that activates the hyperlink depends on the application. Consult the documentation for each product for the specific keystroke.

When the user activates the hyperlink, the help system displays the topic referenced by the invisible text. The invisible cross-reference text is formatted as one of the following:

Hidden Text	Action
contextstring	Displays the topic associated with *contextstring*. For example, `exeformat` displays the topic text for the context `exeformat`.
filename!	Treats *filename* as a single topic to be displayed. For example, `$INCLUDE:stdio.h!` searches the directories in the INCLUDE environment variable for file `stdio.h` and displays it as a single help topic.
filename!*contextstring*	Works the same as *contextstring*, except only the help file *filename* is searched for the context. If the file is not already open, the help system finds it (by searching either the current path or an explicit environment variable) and opens it. For example, `$BIN:readme.doc!patches` searches for `readme.doc` in the BIN environment variable and displays the topic associated with `patches`.
!*command*	Executes the command specified after the exclamation point (!).

In the following example, the word `Example` is a hyperlink. The `\b` , `\p`, and `\v` formatting flags mark hyperlinks in the help text. (The formatting flags are listed later in this chapter, in Table 11.4.)

```
\bSee also:\p    Example\vopen.ex\v
```

The hyperlink refers to `open.ex`. If you select any of the letters of `Example`, the help system displays the topic whose context is `open.ex`. On the screen, this line appears as follows:

```
See also: Example
```

An application might display `See also:` and `Example` in different colors or character types, depending on factors such as your default color selection and type of monitor.

When a hyperlink needs to cross-reference more than one word, you must use an anchor, as in the following example:

```
\bSee also:\p   \uExample\p\vprintf.ex\v, fprintf, scanf, sprintf,
vfprintf, vprintf, vsprintf
            \aformatting table\vprintf.table\v
```

This part of the example is an anchored hyperlink:

```
\aformatting table\vprintf.table\v
```

The anchor must fit on one line.

The \a flag creates an anchor for the cross-reference. In the example, the phrase following the \a flag (`formatting table`) is the hyperlink. It refers to the context `printf.table`. The first \v flag marks both the end of the hyperlink and the beginning of the invisible text. The name `printf.table` is invisible; it does not appear on the screen when the help is displayed. The second \v flag ends the invisible text.

11.6 Using Help Database Formats

A database can be written in any of three text formats. The list below briefly describes these types. Sections 11.6.1–11.6.3 describe the formatting types in detail.

An entire help system (such as the one supplied with PWB or QuickC) can handle any combination of formats. For example, the help files for Microsoft C are written in QuickHelp format, and the README.DOC file is unformatted ASCII.

Type	Characteristics
QuickHelp	Uses dot commands and embedded formatting characters (the default formatting type expected by HELPMAKE); supports highlighting, color, and cross-references. Files in this format must be compressed before use.
RTF	Uses a subset of standard RTF; supports highlighting, color, and cross-references; supports some dot commands. Files in this format must be compressed before use.
Minimally formatted ASCII	Uses a help delimiter (>>) to define help contexts; does not support highlighting, color, or cross-references. Files in this format can be compressed, but compression is not required.

11.6.1 QuickHelp Format

The QuickHelp format uses a dot command and embedded formatting flags to convey information to HELPMAKE.

11.6.1.1 QuickHelp Dot Commands

QuickHelp provides a number of dot commands that identify topics and convey other topic-related information to the help system. If your help file contains dot commands other than **.context** or **.comment**, you must supply the /T option when encoding and decoding with HELPMAKE.

You can define more than one context for a single topic.

The most important dot command is the **.context** command. Every topic in a QuickHelp file begins with one or more **.context** commands. Each **.context** command defines a context string for the topic text. You can define more than one context for a single topic, as long as you do not place any topic text between them.

Typical **.context** commands are shown below. The first defines a context for the **#include** C preprocessor directive. The second set illustrates multiple contexts for one block of topic text. In this case, the same topic text explains all of the string-to-number conversion routines in C.

```
.context #include
       .
       . description of #include goes here
       .
.context strtod
.context strtol
.context strtoul
       .
       . description of string-to-number functions goes here
       .
```

The QuickHelp format includes several other dot commands. Table 11.3 lists the dot commands available in QuickHelp format.

Table 11.3 QuickHelp Dot Commands

Command	Action
.category *string*	Lists the category in which the current topic appears and its position in the list of topics. The category name is used by the QuickHelp Categories command, which displays the topics list. Supported only by QuickHelp.
.command	Indicates that the topic text is not a displayable help topic. Use this command to hide hyperlink topics and other internal information.
.comment *string* **..** *string*	The *string* is a comment that appears only in the help source file. Comments are not inserted in the help database, so they cannot be restored when you decompress a help file.
.context *string*	The *string* introduces a topic.
.end	Ends a paste section. See the **.paste** command below. Supported only by QuickHelp.
.freeze *numlines*	Locks the first *numlines* lines at the top of the screen. This can be used to preserve a bar of cross-reference buttons for a help topic and prevent it from being scrolled.
.length *topiclength*	Indicates the default window size, in *topiclength* lines, of the topic about to be displayed.
.line *number*	Tells HELPMAKE to reset the line number to begin at *number* for subsequent lines of the input file. Line numbers appear in HELPMAKE error messages. HELPMAKE does not put the **.line** command into the help database, so it is not restored during decompression. See **.source**.
.list	Indicates that the current topic contains a list of topics. QuickHelp displays a highlighted line; you can choose a topic by moving the highlighted line over the desired topic and pressing ENTER. Help searches for the first word of the line. Supported only by QuickHelp.
.mark *name* [[*column*]]	Defines a mark immediately preceding the following line of text. The marked line shows a script command where the display of a topic begins. The *name* identifies the mark. The *column* is an integer value specifying a column location within the marked line. Supported only by QuickHelp.

Table 11.3 (*continued*)

Command	Action
.next *context*	Tells the help system to look up the next topic using *context* instead of the topic that physically follows it in the file. You can use this command to skip large blocks of **.command** or **.popup** topics.
.paste *pastename*	Begins a paste section. The *pastename* appears in the QuickHelp Paste menu. Supported only by QuickHelp.
.popup	Tells the help system to display the current topic as a popup instead of a normal, scrollable topic. Supported only by QuickHelp.
.previous *context*	Tells the help system to look up the previous topic using *context* instead of the topic that physically precedes it in the file. You can use this command to skip large blocks of **.command** or **.popup** topics.
.raw	Turns off special processing of certain characters by the application.
.ref *topic* [[, *topic*]] ...	Tells the help system to display the *topic* in the Reference menu. You can list as many *topic*s as needed; separate each additional *topic* with a comma. A **.ref** command is formatted without regard to the /W option. Supported only by QuickHelp. If no *topic* is specified, QuickHelp searches the line immediately following for a See: or See Also: reference; if present, the reference must be the first non-white-space characters on the line.
.source *filename*	Tells HELPMAKE that subsequent topics come from *filename*. By default, when an error occurs, the error message contains the name and line number of the input file. The **.source** command tells HELPMAKE to use *filename* in the error message instead of the name of the input file and to reset the line number to 1. This is useful when you concatenate several sources to form the input file. HELPMAKE does not put the **.source** command into the help database, so it is not restored during decompression. See **.line**.
.topic *text*	Defines *text* as the name or title to be displayed in place of the context string if the application help displays a title. This command is always the first line in the context unless you also use the **.length** or **.freeze** commands.

11.6.1.2 QuickHelp Formatting Flags

The QuickHelp format provides a number of formatting flags that are used to highlight parts of the help database and to mark hyperlinks in the help text.

Each formatting flag consists of a backslash (\ ) followed by a character. Table 11.4 lists the formatting flags.

Table 11.4 QuickHelp Formatting Flags

Formatting Flag	Action
\a	Anchors text for cross-references
\b, \B	Turns boldface on or off
\i, \I	Turns italics on or off
\p, \P	Turns off all attributes
\u, \U	Turns underlining on or off
\v, \V	Turns invisibility on or off (hides cross-references in text)
\\	Inserts a single backslash in text

On monochrome monitors, text labeled with the bold, italic, and underline attributes appears in various ways, depending on the application (for example, high intensity and reverse video are commonly displayed). On color monitors, these attributes are translated by the application into suitable colors, depending on the user's default color selections.

The \b, \i, \u, and \v options are toggles, turning on and off their respective attributes. You can use several of these on the same text. Use the \p attribute to turn off all attributes. Use the \v attribute to hide cross-references and hyperlinks in the text.

HELPMAKE truncates the lines in QuickHelp files to the width specified with the /W option. Only visible characters count toward the character-width limit. Lines that begin with an application-specific control character are truncated to 255 characters regardless of the width specification. See Section 11.3.1, "Options for Encoding," for more information on truncation and application-specific control characters.

In the example below, the \b flag initiates boldface text for `Returns:`, and the \p flag changes the remaining text to plain text.

```
\bReturns:\p    a handle if successful, or -1 if not.
           errno:  EACCES, EEXIST, EMFILE, ENOENT
```

In the example below, the \a flag anchors text for the hyperlink `Example`. The \v flags define the cross-reference `sample_prog` and make the text between the \v flags invisible. Cross-references are described in the following section.

```
\aExample \vsample_prog\v
```

11.6.1.3 QuickHelp Cross-References

Help databases contain two types of cross-references, implicit and explicit. They are described in Section 11.1.1, "Contents of a Help File."

Any word that appears as a global context is implicitly cross-referenced. For example, any time you request help in PWB on **close**, the help window displays information about that function. You do not code implicit cross-references into your help text files.

Insert formatting flags to mark explicit cross-references.

Explicit cross-references (hyperlinks) are words or phrases on the screen that point to a context. For example, almost every "See:" and "See also:" reference in online help has a hyperlink pointing to the appropriate context. You can view the cross-referenced material immediately by activating the hyperlink, without having to search the help system's menus for the topic. You must insert formatting flags in your help text files to mark explicit cross-references.

If the hyperlink consists of a single word, you can use invisible text to flag it in the source file. The \v formatting flag creates invisible text, as follows:

hyperlink **\v***context* **\v**

Put the first \v flag immediately following the word you want to be the hyperlink. Following the flag, insert the context that the hyperlink points to. The second \v flag marks the end of the context; that is, the end of the invisible text. HELPMAKE generates a cross-reference whose context is the invisible text and whose hyperlink is the word.

If the hyperlink consists of a phrase, rather than a single word, you must use anchored text to create explicit cross-references. Use the \a and \v flags to create anchored text as follows:

\ a*hyperlink-words* **\v***context***\v**

The \a flag marks an anchor for the cross-reference. The text that follows the \a flag is the hyperlink. The hyperlink must fit entirely on one line. The first \v flag marks both the end of the hyperlink and the beginning of the invisible text that contains the cross-reference context. The second \v flag marks the end of the invisible text.

The C functions **abs**, **cabs**, and **fabs** in the following examples are implicit cross-references because they have a global context in the help system.

```
See also: abs, cabs, fabs
```

The next example shows the encoding for an explicit cross-reference to an example program and a function template from the help database for the Microsoft C run-time library:

```
See also: Example\vopen.ex\v, Template\vopen.tm\v, close
```

Here, the hyperlinks are `Example` and `Template`, which reference the contexts `open.ex` and `open.tm`. The example also contains an implicit cross-reference to the close function.

The final example shows the encoding for an explicit cross-reference to an entire family of functions:

```
See also: \ais... functions\vis_functions\v, atoi
```

The cross-reference uses anchored text to associate a phrase, rather than just a word, with a context. In this example, the hyperlink is the anchored phrase `is... functions`, and it cross-references the context `is_functions`. In addition, the example contains an implicit cross-reference to the C-language **atoi** routine.

11.6.1.4 QuickHelp Example

The code below is an example in QuickHelp format that contains a single entry:

```
.context open
.length 13
\bInclude:\p   <fcntl.h>, <io.h>, <sys\\types.h>, <sys\\stat.h>

\bPrototype:\p  int open(char *path, int flag[, int mode]);
         oflag:  O_APPEND O_BINARY O_CREAT O_EXCL O_RDONLY
                 O_RDWR    O_TEXT    O_TRUNC  O_WRONLY
                 (can be joined by |)
         pmode:  S_IWRITE  S_IREAD   S_IREAD | S_IWRITE

\bReturns:\p    a handle if successful, or -1 if not.
            errno:  EACCES, EEXIST, EMFILE, ENOENT

\bSee also:\p  \uExample\p\vopen.ex\v, \uTemplate\p\vopen.tp\v,
             access, chmod, close, creat, dup, dup2, fopen, sopen, umask
```

The **.length** command near the beginning of the example specifies the size of the initial window for the help text. Here, the initial window displays 13 lines.

The manifest constants (such as **O_WRONLY** and **EEXIST**), the C keywords (such as **int** and **char**), and the other functions (such as **access** and **sopen**) are implicit cross-references. The words `Example` and `Template` are explicit cross-references to the example `open.ex` and to the **open** template `open.tp`, respectively. Note the use of double backslashes in the include file names.

11.6.2 Rich Text Format

Rich Text Format (RTF) is a Microsoft word-processing format supported by several word processors, including Microsoft Word 5.0 and Microsoft Word for Windows. RTF allows documents to be transferred between applications without loss of formatting. The HELPMAKE utility recognizes a subset of the full RTF syntax. If your file contains RTF codes that are not part of the subset, HELPMAKE discards them.

To create an RTF-formatted file, enter the text and format it as you want it to appear: bold, underlined, hidden, italic, and so forth. (You can combine attributes.) You can also format paragraphs, selecting body and first-line indenting. The only items you need to insert into an RTF file manually are the help delimiter (>>) and the context string that start each entry.

When you have entered and formatted the text, save it in RTF format. In Microsoft Word 5.0, for example, this means choosing Transfer Save, then highlighting RTF in the format: field.

You do not see the RTF formatting codes when you load an RTF file into a compatible word processor; the word processor removes them and displays the text with the specified attribute(s). However, you can view these codes by loading an RTF file into a plain-text word processor.

HELPMAKE recognizes the subset of RTF codes listed in Table 11.5.

Table 11.5 RTF Formatting Codes

RTF Code	Action
\b	Boldface. The application decides how to display this; often it is intensified text.
\fi*n*	Paragraph first-line indent, *n* columns.
\i	Italic. The application decides how to display this; often it is reverse video.
\ li*n*	Paragraph indent from left margin, *n* columns.
\ line	New line (not new paragraph).
\ par	End of paragraph.
\ pard	Default paragraph formatting.
\ plain	Default attributes. On most screens, this is nonblinking normal intensity.
\tab	Tab character.
\ul	Underline. The application decides how to display this; some adapters that do not support underlining display it as blue text.
\v	Hidden text. Hidden text is used for cross-reference information and for some application-specific communications; it is not displayed.

When HELPMAKE compresses the file, it formats the text to the width given with the /W option, ignoring the paragraph formats.

As with the other text formats, each entry in the database source consists of one or more context strings, followed by topic text. An RTF file can contain Quick-Help dot commands.

The help delimiter (>>) at the beginning of any paragraph marks the beginning of a new help entry. The text that follows on the same line is defined as a context for the topic. If the next paragraph also begins with the help delimiter, it also defines a context string for the same topic text. You can define any number of contexts for a block of topic text. The topic text comprises all subsequent paragraphs up to the next paragraph that begins with the help delimiter.

The example below is a help database containing a single entry using subset RTF text. Note that RTF uses curly braces ({ }) for nesting. Thus, the entire file is enclosed in curly braces, as is each specially formatted text item.

```
{\rtf1
\pard >>open\par
  {\b Include:}    <fcntl.h>, <io.h>, <sys\\types.h>, <sys\\stat.h>\par
\par
  {\b Syntax:}     int open( char * filename, int oflag[, int pmode ] );\par
             oflag:  O_APPEND  O_BINARY  O_CREAT  O_EXCL  O_RDONLY\par
                     O_RDWR    O_TEXT    O_TRUNC  O_WRONLY\par
                     (may be joined by |)\par
             pmode:  S_IWRITE  S_IREAD   S_IREAD | S_IWRITE\par
\par
  {\b Returns:}    a handle if successful, or -1 if not.\par
             errno:  EACCES, EEXIST, EMFILE, ENOENT\par
\par
  {\b See also:}  Examples{\v open.ex}, access, chmod, close, creat, dup,\par
             dup2, fopen, sopen, umask\par
>>open.ex\par
To build this help file, use the following command:\par
\par
HELPMAKE /S1 /E15 /OOPEN.HLP OPEN.RTF\par
\par

        < Back >{\v !B}
}
```

RTF files normally contain additional information that is not visible to the user; HELPMAKE ignores this extra information.

11.6.3 Minimally Formatted ASCII Format

A minimally formatted ASCII text file comprises a sequence of topics, each preceded by one or more unique context definitions. Each context definition must be on a separate line beginning with a help delimiter (>>). Subsequent lines up to the next context definition constitute the topic text.

Minimally formatted ASCII files cannot contain highlighting.

There are two ways to use a minimally formatted ASCII file. You can compress it with HELPMAKE, creating a help database, or an application can access the uncompressed file directly. Compressing minimally formatted ASCII files increases search speed. Uncompressed files are somewhat larger and slower to search. Minimally formatted ASCII files have a fixed width, and they cannot contain highlighting (or other nondefault attributes) or explicit cross-references.

The following example, coded in minimally formatted ASCII, shows the same text as the QuickHelp example presented earlier in this section. The first line of the example defines `open` as a context string. The minimally formatted ASCII help file must begin with the help delimiter (>>), so that HELPMAKE or the application can verify that the file is indeed an ASCII help file.

```
>>open

Include:    <fcntl.h>, <io.h>, <sys\types.h>, <sys\stat.h>

Prototype:  int open(char *path, int flag[, int mode]);
       oflag:  O_APPEND  O_BINARY  O_CREAT  O_EXCL  O_RDONLY
               O_RDWR    O_TEXT    O_TRUNC  O_WRONLY
               (can be joined by |)
       pmode:  S_IWRITE  S_IREAD   S_IREAD | S_IWRITE

Returns:    a handle if successful, or -1 if not.
            errno:  EACCES, EEXIST, EMFILE, ENOENT

See also:  access, chmod, close, creat, dup, dup2, fopen, sopen, umask
```

When displayed, the help information appears exactly as it is typed into the file. Any formatting codes are treated as ASCII text.

11.7 Related Topics in Online Help

Information on the following related topics can be found in online help.

Topic	Access
HELPMAKE	Choose "HELPMAKE" from the "Microsoft Advisor Contents" screen
QuickHelp	Choose "QH" from the "Microsoft Advisor Contents" screen

Chapter 12
Linking Object Files with LINK

This chapter describes the Microsoft Segmented-Executable Linker (LINK), which combines compiled or assembled object files into an executable file. It explains LINK's input syntax and fields and tells how to use options to control LINK. It discusses overlays in DOS programs and concludes with background information about LINK.

12.1 Overview

LINK combines 80*x*86 object files into either an executable file or a dynamic-link library (DLL). The object-file format is the Microsoft Relocatable Object-Module Format (OMF), based on the Intel 8086 OMF. LINK uses library files in Microsoft library format.

LINK creates "relocatable" executable files and DLLs—that is, the operating system can load and execute these files in any unused section of memory. LINK can create DOS executable files with up to 1 megabyte of code and data (or up to 16 megabytes when using overlays), or OS/2 and Microsoft Windows programs with up to 16 megabytes.

For more information on OMF, executable-file format, and the linking process, see the *MS-DOS Encyclopedia*.

Use BIND to create an OS/2 program that also runs under DOS.

The linker produces programs that run under DOS only or under OS/2 only, but not both. However, if an OS/2 program limits its OS/2 function calls to the Family API subset, you can use the Microsoft Bind Utility (BIND) to modify the OS/2 executable file so that it runs under both OS/2 and DOS. For more information, see online help.

Use EXEHDR to examine the finished file.

When the file (either executable or DLL) is created, you can examine the information that LINK puts in the file's header by using the Microsoft EXE File Header Utility (EXEHDR). For more information, see online help.

Other programs can call LINK automatically.

The Programmer's WorkBench (PWB) invokes LINK to create the final executable file or DLL. Therefore, if you develop your software with PWB, you might not need to read this chapter. However, the detailed explanations of LINK options might be helpful when you use the LINK Options dialog box in PWB. This information is also available in online help.

The compiler or assembler supplied with your language (CL with C, FL with FORTRAN, ML with MASM) also invokes LINK. You can use most of the LINK options described in this chapter with this utility. Online help has more

information about the compilers and assembler: select help for the appropriate language from the Compiler box of the help Contents screen.

NOTE Unless otherwise noted, all references to "library" in this chapter refer to a static library, either a standard library created by the Microsoft Library Manager (LIB) or an import library created by the Microsoft Import Library Manager (IMPLIB), and not a DLL.

12.2 LINK Output Files

LINK is a bound application that runs under both DOS and OS/2 and can create executable files for DOS, OS/2, or Windows. You do not have to run LINK under OS/2 to create OS/2 applications, or under DOS to create DOS programs. The kind of file produced is determined by the way the source code is compiled and the information supplied to LINK, not the operating system LINK runs under.

A program that runs under DOS is called an executable file or application. A program or DLL that runs under Windows or OS/2 is called a segmented executable file. LINK creates the appropriate file according to the following rules:

- If a module-definition file or import library is not specified and the object files and libraries do not contain export definitions, LINK creates an application that runs under DOS.
- If a module-definition file containing a **LIBRARY** statement is specified, LINK creates a DLL for Windows or OS/2.
- If any other form of module-definition file is specified, or if any of the object files contains an exported definition, LINK creates an application to run under Windows or OS/2.

LINK looks for the default run-time libraries named in the object files. Default libraries can be real or protected mode. (The mode is usually set when the language product is installed.) Protected-mode libraries contain export definitions. If LINK finds protected-mode default libraries, the output file will be a segmented executable file rather than a DOS file.

The file OS2.LIB is an import library. Linking with OS2.LIB produces an OS/2 application or DLL. When you use a Microsoft high-level language to compile for protected mode, the compiler automatically specifies OS2.LIB as a default library.

LINK's output is either an executable file or a DLL. For simplicity, this chapter sometimes refers to this output as the "main file" or "main output."

Map files list the segments and symbols in a program.

LINK also creates a "rnap" file, which lists the segments in the executable file. The /MAP option adds public symbols to the map file, and the /LINE option adds line numbers.

LINK produces other files when certain options are used.

Other options tell LINK to create other kinds of output files. The /INCR option creates .ILK and .SYM files for incremental linking with ILINK. LINK produces a .COM file instead of an .EXE file when the /TINY option is specified. The combination of /CO and /TINY puts debugging information into a .DBG file. A Quick library results when the /Q option is specified. For more information on these and other options, see Section 12.5, "LINK Options."

12.3 LINK Syntax and Input

The LINK command has the following syntax:

LINK *objfiles*[[, [[*exefile*]] [[, [[*mapfile*]][[, [[*libraries*]][[, *deffile*]]]]]]]][[;]]

The LINK fields perform the following functions:

- The *objfiles* field is a list of the object files that are to be linked into an executable file or DLL. It is the only required field.
- The *exefile* field lets you change the name of the output file from its default.
- The *mapfile* field gives the map file a name other than its default name.
- The *libraries* field specifies additional (or replacement) libraries to search for unresolved references.
- The *deffile* field gives the name of a description file needed to create Windows and OS/2 applications and DLLs.

Fields are separated by commas. You can specify all the fields or leave one or more fields (including *objfiles*) blank; LINK will then prompt you for the missing input. (For an explanation of how to use LINK prompts, see Section 12.4, "Running LINK.") To leave a field blank, enter only the field's trailing comma.

Options can be specified in any field. For descriptions of each of LINK's options, see Section 12.5, "LINK Options."

The fields must be entered in the order shown, whether they contain input or are left blank. A semicolon (;) at the end of the LINK command line terminates the command and suppresses prompting for any missing fields. LINK then assumes the default values for the missing fields.

If your file appears in or is to be created in another directory or device, you must supply the full pathname. Filenames are not case sensitive.

The next five sections explain how to use each of the LINK fields.

12.3.1 The *objfiles* Field

The *objfiles* field specifies one or more object files to be linked. At least one filename must be entered. If you do not supply an extension, LINK assumes a default .OBJ extension. If the filename has no extension, add a period (.) at the end of its name.

If you name more than one object file, separate the names with a plus sign (+) or a space. To extend *objfiles* to the following line, type a plus sign (+) as the last character on the current line, press ENTER, and continue. Do not split a name across lines.

12.3.1.1 Load Libraries

The *objfiles* field can also specify library files. A library specified this way becomes a "load library." You must specify the library's filename extension; otherwise, LINK assumes an .OBJ extension.

LINK treats load libraries as any other object file: it puts every object module from a load library in the executable file, regardless of whether a module satisfies an unresolved external reference. The effect is the same as if you had specified all the library's object-module names in the *objfiles* field.

Specifying a load library can therefore create an executable file or DLL that is larger than it needs to be. (A library named in the *libraries* field adds only those modules required to resolve external references.) However, loading an entire library can be useful when

- Repeatedly specifying the same group of object files
- Placing a library in an overlay
- Debugging, so you can call library routines that would not be included in the release version of the program

12.3.1.2 How LINK Searches for Object Files

When searching for object (and load-library) files, LINK looks in the following locations in the order specified:

1. The directory specified for the file (if a path is included). If the file is not in that directory, the search terminates.
2. The current directory.
3. Any directories specified in the LIB environment variable.

If LINK cannot find an object file, and a floppy drive is associated with that object file, LINK pauses and prompts you to insert a disk containing the object file.

If you specify a library in the *objfiles* field, LINK treats it like any other object file. LINK therefore does not search for load libraries in directories named in the *libraries* field.

12.3.1.3 Overlays

A special syntax for the *objfiles* field lets you create DOS programs that use overlay modules. For more information about overlays, see Section 12.7, "Using Overlays under DOS."

12.3.2 The *exefile* Field

The *exefile* field is used to specify a name for the main output file. If you do not supply an extension, LINK assumes a default extension, either .EXE, .COM (when using the /TINY option), .DLL (when using a module-definition file containing a **LIBRARY** statement), or .QLB (when using the /Q option).

If you do not specify an *exefile*, LINK gives the main output a default name. This name is the base name of the first file listed in the *objfiles* field, plus the extension appropriate for the type of executable file being created.

LINK creates the main file in the current directory unless you specify an explicit path with the filename.

12.3.3 The *mapfile* Field

The *mapfile* field is used to specify a filename for the map file or to suppress creation of a map file. A map file lists the segments in the executable file or DLL.

You can specify a path with the filename. The default extension is .MAP. Specify `NUL` to suppress the creation of a map file. The default for the *mapfile* field is one of the following:

- If this field is left blank on the command line or in a response file, LINK creates a map file with the base name of the *exefile* (or the first object file if no *exefile* is specified) and the extension .MAP.
- When using LINK prompts, LINK assumes either the default described above (if an empty *mapfile* field is specified) or `NUL.MAP`, which suppresses creation of a map file.

To add line numbers to the map file, use the /LINE option. To add public symbols, use the /MAP option. Both /LINE and /MAP force a map file to be created unless NULL is explicitly specified.

12.3.4 The *libraries* Field

You can specify one or more standard or import libraries (not DLLs) in the *libraries* field. If you name more than one library, separate the names with a plus sign (+) or a space. To extend *libraries* to the following line, type a plus sign (+) as the last character on the current line, press ENTER, and continue. Do not split a name across lines. If you specify the base name of a library without an extension, LINK assumes a default .LIB extension.

If no library is specified, LINK searches only the default libraries named in the object files to resolve unresolved references. If one or more libraries are specified, LINK searches them in the order named before searching the default libraries.

You can tell LINK to search additional directories for specified or default libraries by giving a drive name or path specification in the *libraries* field; end the specification with a backslash (\ ). (If you don't include the backslash, LINK assumes the last element of the path is a library file.) LINK looks for files ending in .LIB in these directories.

You can specify a total of 32 paths or libraries in the field. If you give more than 32 paths or libraries, LINK ignores the additional specifications without warning you.

You might need to specify library names when you want to

- Use a default library that has been renamed.
- Specify a library other than the default named in the object file (for example, a library that handles floating-point arithmetic differently from the default library).
- Search additional libraries.
- Find a library not in the current directory and not in a directory specified by the LIB environment variable.

12.3.4.1 Overriding Default-Library Searches

Most compilers insert the names of the required language libraries in the object files. LINK searches for these default libraries automatically; you do not need to specify them in the *libraries* field. The libraries must already exist with the name expected by LINK. Default-library names usually refer to combined libraries built and named during setup; consult your compiler documentation for more information about default libraries.

To make LINK ignore the default libraries, use the /NOD option. This leaves unresolved references in the object files, so you must use the *libraries* field to specify the alternative libraries that LINK is to search.

12.3.4.2 Import Libraries

You can specify import libraries created by the IMPLIB utility anywhere you can specify standard libraries. You can also use the LIB utility to combine import libraries and standard libraries. These combined libraries can then be specified in the *libraries* field.

12.3.4.3 How LINK Resolves References

LINK searches static libraries to resolve external references. A static library is either a standard library created by the LIB utility or an import library created by the IMPLIB utility. The linker searches first in the libraries and library directories you specify (in the order you specify them), then in the default libraries. If a default library is explicitly specified, it is searched in the order it is given.

LINK uses only those library modules needed to resolve external references, not the entire library. However, if you enter a library as a load library in the *objfiles* field, all the modules of a load library are added to the main output.

12.3.4.4 How LINK Searches for Library Files

When searching for libraries, LINK looks in the following locations in this order:

1. The directory specified for the file (if a path is included). If the file is not in that directory, the search terminates. (The default libraries named in object files by Microsoft compilers do not include path specifications.)
2. The current directory.
3. Any directories in the *libraries* field.
4. Any directories specified in the LIB environment variable.

If LINK cannot locate a library file, it prompts you to enter the location. The /BATCH option disables this prompting.

Example

The following is a specification in the *libraries* field:

```
C:\TESTLIB\ NEWLIBV3 C:\MYLIBS\SPECIAL
```

LINK searches NEWLIBV3.LIB first for unresolved references. Since no directory is specified for NEWLIBV3.LIB, LINK searches the following locations in this order:

1. The current directory
2. The C:\TESTLIB\ directory
3. The directories in the LIB environment variable

If LINK still cannot find NEWLIBV3.LIB, it prompts you with the message

```
Enter new file spec
```

You can then enter either a path to the library or a full pathname for another library.

If unresolved references remain after searching NEWLIBV3.LIB, LINK then searches the library C:\MYLIBS\SPECIAL.LIB. If LINK cannot find this library, it prompts you as described above for NEWLIBV3.LIB. If there are still unresolved references, LINK searches the default libraries.

12.3.5 The *deffile* Field

Use the *deffile* field to specify a module-definition file when you are linking a segmented executable file, which is an application or DLL for OS/2 or Windows. A module-definition file is optional for an application but required for a DLL. If you specify a base name with no extension, LINK assumes a .DEF extension. If the filename has no extension, put a period (.) at the end of the name.

By default, LINK assumes that no *deffile* needs to be specified. If you are linking for DOS, use a semicolon to terminate the command line before the *deffile* field (or accept the default NUL.DEF at the `Definitions File` prompt).

12.3.5.1 How LINK Searches for Module-Definition Files

LINK searches for the module-definition file in the following order:

1. The directory specified for the file (if a path is included). If the file is not in that directory, the search terminates.
2. The current directory.

For information on module-definition files, see Chapter 13.

12.3.6 Examples

The following examples illustrate various uses of the LINK command line.

Example 1

```
LINK FUN+TEXT+TABLE+CARE, , FUNLIST, XLIB.LIB;
```

This command line links the object files FUN.OBJ, TEXT.OBJ, TABLE.OBJ, and CARE.OBJ. By default, the executable file is named FUN.EXE, because the base name of the first object file is `FUN`, and no name is specified for the executable file. The map file is named FUNLIST.MAP. LINK searches for unresolved external references in the library XLIB.LIB before searching in the default

libraries. LINK does not prompt for a .DEF file because a semicolon appears before the *deffile* field.

Example 2

```
LINK FUN, , ;
```

This command produces a map file named FUN.MAP because a comma appears as a placeholder for the *mapfile* field on the command line.

Example 3

```
LINK FUN, ;
LINK FUN;
```

Neither of these commands produces a map file, because commas do not appear as placeholders for the *mapfile* field. The semicolon (;) terminates the command line and accepts all remaining defaults without prompting; the prompting default for the map file is *not* to create one.

Example 4

```
LINK MAIN+GETDATA+PRINTIT, , MAIN ;
```

This command links the files MAIN.OBJ, GETDATA.OBJ, and PRINTIT.OBJ into a DOS executable file because no module-definition file is specified. The map file MAIN.MAP is created.

Example 5

```
LINK GETDATA+PRINTIT, , , , MODDEF
```

This command links GETDATA.OBJ and PRINTIT.OBJ into a DLL if MODDEF.DEF contains a **LIBRARY** statement. Otherwise, it links them into a segmented executable file for OS/2 or Windows. LINK creates a map file named GETDATA.MAP.

12.4 Running LINK

The simplest use of LINK is to combine one or more object files with a run-time library to create an executable file. You type `LINK` at the command-line prompt, followed by the names of the object files and a semicolon (;). LINK combines the object files with language libraries specified in the object files to create an executable file. By default, the executable file takes the name of the first object file in the list.

To interrupt LINK and return to the operating-system prompt, press CTRL+C at any time.

LINK expects you to supply at least one input field (the *objfiles* field), and as many as five. There are several ways to supply the input fields LINK expects:

- Enter all the required input directly on the command line.
- Omit one or more of the input fields and respond when LINK prompts for the missing fields.
- Put the input in a response file and enter the response-file name in place of the expected input.

These methods can be used in combination. The LINK command line was discussed in Section 12.3. The following sections explain the other two methods.

12.4.1 Specifying Input with LINK Prompts

If any field is missing from the LINK command line and the line does not end with a semicolon, or if any of the supplied fields are invalid, LINK prompts you for the missing or incorrect information. LINK displays one prompt at a time and waits until you respond:

```
Object Modules [.OBJ]:
Run File [basename.EXE]:
List File [NUL.MAP]:
Libraries [.LIB]:
Definitions File [NUL.DEF]:
```

The LINK prompts correspond to the command-line fields described earlier in this chapter. If you want LINK to prompt you for every input field, including *objfiles*, type the command `LINK` by itself.

Options can be entered anywhere in any field, before the semicolon if specified.

12.4.1.1 Defaults

The default values for each field are shown in brackets. Press ENTER to accept the default, or type in the filename(s) you want. The *basename* is the base name of the first object file you specified. To select the default responses for all the remaining prompts and terminate prompting, type a semicolon (;) and press ENTER.

If you specify a filename without giving an extension, LINK adds the appropriate default extension. To specify a filename that does not have an extension, type a period (.) after the name.

Use a space or plus sign (+) to separate multiple filenames in the *objfiles* and *libraries* fields. To extend a long *objfiles* or *libraries* response to a new line, type a plus sign (+) as the last character on the current line and press ENTER. You can continue entering your response when the same prompt appears on a new line. Do not split a filename or a pathname across lines.

12.4.2 Specifying Input in a Response File

You can supply input to LINK in a response file. A response file is a text file containing the input LINK expects on the command line or in response to prompts. Response files can be used to hold frequently used options or responses, or to overcome the 128-character limit on the length of a DOS command line.

12.4.2.1 Usage

Specify the name of the response file in place of the expected command-line input or in response to a prompt. Precede the name with an at sign (@), as in *@responsefile*. You must specify an extension if the response file has one; there is no default extension. You can specify a path with the filename.

You can specify a response file in any field (either on the command line or when responding to prompts) to supply input for one or more consecutive fields or all remaining fields. Note that LINK assumes nothing about the contents of the response file; LINK simply reads the fields from the file and applies them, in order, to the fields for which it has no input. LINK ignores any fields in the response file or on the command line after the five expected fields are satisfied or a semicolon (;) appears.

Example

The following command invokes LINK and supplies all input in a response file, except the last input field:

```
LINK @input.txt, mydefs
```

12.4.2.2 Contents of the Response File

Each input field must appear on a separate line or be separated from other fields on the same line by a comma. You can extend a field to the following line by adding a plus sign (+) at the end of the current line. A blank field can be represented by either a blank line or a comma.

Options can be entered anywhere in any field, before the semicolon if specified.

If a response file does not specify all the fields, LINK prompts you for the rest. Use a semicolon (;) to suppress prompting and accept the default responses for all remaining fields.

Example

```
FUN TEXT TABLE+
CARE
/MAP
FUNLIST
GRAF.LIB ;
```

If the response file above is named `FUN.LNK`, the command

```
LINK @FUN.LNK
```

causes LINK to

- Link the four object files FUN.OBJ, TEXT.OBJ, TABLE.OBJ, and CARE.OBJ into an executable file named FUN.EXE.
- Include public symbols and addresses in the map file.
- Make the name of the map file FUNLIST.MAP.
- Link any needed routines from the library file GRAF.LIB.
- Assume no module-definition file.

12.5 LINK Options

This section explains how to use options to control LINK's behavior and modify LINK's output. It contains a description of each option following a brief introduction on how to specify options.

12.5.1 Specifying Options

The following paragraphs discuss rules for using options.

12.5.1.1 Syntax

All options begin with a slash (/). You can specify an option by using the shortest sequence of characters that uniquely identifies the option. The description for each option shows the minimum legal abbreviation with the optional part enclosed in double brackets. No gaps or transpositions of letters are allowed. For example,

/B[[ATCH]]

indicates that either /B or /BATCH can be used, as can /BA, /BAT, or /BATC. Option names are not case sensitive, so you can also specify /batch or /Batch. This chapter uses meaningful yet legal forms of the option names.

12.5.1.2 Usage

LINK options can appear on the command line, in response to a prompt, or as part of a field in a response file. They can also be specified in the LINK environment variable. (For more information, see Section 12.6, "Setting Options with the LINK Environment Variable.") Options can appear in any field before the last input, except as noted in the descriptions.

If an option appears more than once (for example, on the command line and in the LINK variable), the effect is the same as if the option was given only once. If two options conflict, the most recently specified option takes effect. This means that a command-line option or one given in response to a prompt overrides one specified in the LINK environment variable. For example, the command-line option /SEG:512 cancels the effect of the environment-variable option /SEG:256.

12.5.1.3 Numeric Arguments

Some LINK options take numeric arguments. You can enter numbers either in decimal format or in standard C-language notation.

12.5.2 The /ALIGN Option

Option

/A[[LIGNMENT]]:*size*

The /ALIGN option aligns segments in a segmented executable file at the boundaries specified by *size*. The *size* argument must be an integer power of two. For example,

```
/ALIGN:16
```

indicates an alignment boundary of 16 bytes. The default alignment is 512 bytes.

This option reduces the size of the disk file by reducing the size of gaps between segments. It has no effect on the size of the file when loaded in memory.

12.5.3 The /BATCH Option

Option

/B[[ATCH]]

The /BATCH option suppresses prompting for libraries or object files that LINK cannot find. By default, the linker prompts for a new pathname whenever it cannot find a library that it has been directed to use. It also prompts you if it cannot find an object file that it expects to find on a floppy disk. When /BATCH is used, the linker generates an error or warning message (if appropriate). The /BATCH option also suppresses the LINK copyright message and echoed input from response files.

Using this option can cause unresolved external references. It is intended primarily for users who use batch files or makefiles for linking many executable files with a single command and who wish to prevent linker operation from halting.

NOTE This option does not suppress prompts for input fields. Use a semicolon (;) at the end of the LINK input to suppress input prompting.

12.5.4 The /CO Option

Option

/CO[[DEVIEW]]

The /CO option adds line numbers and symbolic data to the executable file for use with the Microsoft CodeView debugger. The /CO option has no effect if the object files do not contain CodeView debugging information.

You can run the resulting executable file outside CodeView; the debugging data in the file is ignored. However, it increases file size and slows execution slightly. You should link a separate release version without the /CO option after the program has been debugged.

When /CO is used with the /TINY option, debug information is put in a separate file with the same base name as the .COM file and with the .DBG extension.

The /CO option is not compatible with the /EXEPACK option for DOS executable files.

12.5.5 The /CPARM Option

Option

/CP[[ARMAXALLOC]]:*number*

The /CPARM option sets the maximum number of 16-byte paragraphs needed by the program when it is loaded into memory. The operating system uses this value to allocate space for the program before loading it. This option is useful when you want to execute another program from within your program and you need to reserve memory for the program. The /CPARM option is valid only when linking DOS programs.

LINK normally requests the operating system to set the maximum number of paragraphs to 65,535. Since this is more memory than DOS can supply, the operating system always denies the request and allocates the largest contiguous block of memory it can find. If the /CPARM option is used, the operating system allocates no more space than the option specified. Any memory in excess of that required for the program loaded is free for other programs.

The *number* can be any integer value in the range 1 to 65,535. If *number* is less than the minimum number of paragraphs needed by the program, LINK ignores your request and sets the maximum value equal to whatever the minimum value happens to be. The minimum number of paragraphs needed by a program is never less than the number of paragraphs of code and data in the program. To

free more memory for programs compiled in the medium and large models, link with /CPARM:1. This leaves no space for the near heap.

NOTE You can change the maximum allocation after linking by using the EXEHDR utility, which modifies the executable-file header.

12.5.6 The /DOSSEG Option

Option

/DO[[SSEG]]

The /DOSSEG option forces segments to be ordered as follows:

1. All segments with a class name ending in CODE
2. All other segments outside DGROUP
3. DGROUP segments, in the following order:
   a. Any segments of class BEGDATA. (This class name is reserved for Microsoft use.)
   b. Any segments not of class BEGDATA, BSS, or STACK.
   c. Segments of class BSS.
   d. Segments of class STACK.

In addition, /DOSSEG option defines the following two labels:

```
_edata = DGROUP : BSS
_end   = DGROUP : STACK
```

The variables `_edata` and `_end` have special meanings for Microsoft compilers, so you should not define program variables with these names. Assembly-language programs can reference these variables but should not change them.

The /DOSSEG option also inserts 16 null bytes at the beginning of the _TEXT segment (if this segment is defined). This behavior of the option is overridden by the /NONULLS option when both are used; use /NONULLS to override the DOSSEG comment record commonly found in standard Microsoft libraries.

This option is principally for use with assembly-language programs. When you link high-level-language programs, a special object-module record in the Microsoft language libraries automatically enables the /DOSSEG option. This option is also enabled by assembly modules that use MASM directive **.DOSSEG**.

12.5.7 The /DSALLOC Option

Option

/DS[[ALLOCATE]]

The /DSALLOC option tells LINK to load all data starting at the high end of the data segment. At run time, the data segment (DS) register is set to the lowest data-segment address that contains program data.

By default, LINK loads all data starting at the low end of the data segment. At run time, the DS register is set to the lowest possible address to allow the entire data segment to be used.

The /DSALLOC option is most often used with the /HIGH option to take advantage of unused memory within the data segment. These options are valid only for assembly-language programs that create DOS .EXE files.

12.5.8 The /EXEPACK Option

Option

/E[[XEPACK]]

The /EXEPACK option directs LINK to remove sequences of repeated bytes (usually null characters) and to optimize the load-time relocation table before creating the executable file. (The load-time relocation table is a table of references relative to the start of the program, each of which changes when the executable image is loaded into memory and an actual address for the entry point is assigned.)

The /EXEPACK option does not always produce a significant saving in disk space and may sometimes actually increase file size. Programs that have a large number of load-time relocations (about 500 or more) and long streams of repeated characters are usually shorter if packed. LINK notifies you if the packed file is larger than the unpacked file. The time required to expand a packed file may cause it to load more slowly than a file linked without this option.

You cannot debug packed files with CodeView, because the /EXEPACK option removes symbolic information. A LINK warning message notifies you of this.

The /EXEPACK option is not compatible with the /INCR option or with Windows programs.

12.5.9 The /FARCALL Option

Option

/F[[ARCALLTRANSLATION]]

The /FARCALL option directs the linker to optimize far calls to procedures that lie in the same segment as the caller. This can result in slightly faster code; the gain in speed is most apparent on 80286-based machines and later. The /PACKC option can be used with /FARCALL when linking for OS/2. /PACKC is not recommended when linking Windows applications with /FARCALL.

The /FARCALL option is off by default. If an environment variable (such as LINK or FL) includes /FARCALL, you can use the /NOFARCALL option to override it.

FARCALL optimizes by creating more efficient code.

A program that has multiple code segments may make a far call to a procedure in the same segment. Since the segment address is the same (for both the code and the procedure it calls), only a near call is necessary. Far calls appear in the relocation table; a near call does not require a table entry. By converting far calls to near calls in the same segment, the /FARCALL option both reduces the size of the relocation table and increases execution speed, since only the offset needs to be loaded, not a new segment. The /FARCALL option has no effect on programs that make only near calls, since there are no far calls to convert.

When /FARCALL is specified, the linker optimizes code by removing the instruction `call FAR label` and substituting the following sequence:

```
nop
push    cs
call    NEAR label
```

During execution, the called procedure still returns with a far-return instruction. However, because both the code segment and the near address are on the stack, the far return is executed correctly. The `nop` (no-op) instruction is added so that exactly five bytes replace the five-byte far-call instruction.

In rare cases, /FARCALL should be used with caution.

There is a small risk with the /FARCALL option. If LINK sees the far-call opcode (9A hexadecimal) followed by a far pointer to the current statement, and that segment has a class name ending in `CODE`, it interprets that as a far call. This problem can occur when using `_based (`*segname*` ("CODE"))` in a C program. If a program linked with /FARCALL fails for no apparent reason, try using /NOFARCALL.

Object modules produced by Microsoft high-level languages are safe from this problem because little immediate data is stored in code segments. Assembly-language programs are generally safe for use with the /FARCALL option if they do not involve advanced system-level code, such as might be found in operating systems or interrupt handlers.

12.5.10 The /HELP Option

Option

/HE[LP]

The /HELP option calls the QuickHelp utility. If LINK cannot find the help file or QuickHelp, it displays a brief summary of LINK command-line syntax and options. Do not give a filename when using the /HELP option.

12.5.11 The /HIGH Option

Option

/HI[GH]

At load time, the executable file can be placed either as low or as high in memory as possible. The /HIGH option causes DOS to place the executable file as high as possible in memory. Without the /HIGH option, DOS places the executable file as low as possible. This option is usually used with the /DSALLOC option. These options are valid only for assembly-language programs that create DOS .EXE files.

12.5.12 The /INCR Option

Option

/INC[REMENTAL]

The /INCR option must be used to prepare for subsequent linking with ILINK. This option produces a .SYM file and an .ILK file, each containing additional information needed by ILINK.

When /INCR is specified, LINK creates the main output file as a segmented executable file. If the main output is a DOS application, LINK adds a stub loader so that the program can run under DOS. The file is slightly larger than it would be without /INCR.

The /PADC and /PADD options are often used with the /INCR option to increase buffer size and thereby increase the likelihood that incremental linking will be successful. The /TINY and /EXEPACK options are not compatible with /INCR.

You should not use /INCR or ILINK for the release version of a product. ILINK is intended to speed linking during development and debugging. In rare cases, linking with /INCR causes warning `L4001` to be generated. If this occurs, do not use this option or ILINK.

12.5.13 The /INFO Option

Option

/INF[[ORMATION]]

The /INFO option displays to the standard output information about the linking process, including the phase of linking and the names of the object files being linked. This option is a useful way to determine the locations of the object files being linked, the number of segments, and the order in which they are linked.

12.5.14 The /LINE Option

Option

/LI[[NENUMBERS]]

The /LINE option adds the line numbers and associated addresses from source files to the map file. The object file must contain line-number information for it to appear in the map file. If the object file has no line-number information, the /LINE option has no effect. (Use the /Zd or /Zi option with Microsoft compilers such as CL, FL, and ML to add line numbers to the object file.) If you also want to add public symbols to the map file, use the /MAP option.

The /LINE option causes a map file to be created even if you did not explicitly tell the linker to do so. By default, the map file is given the same base name as the executable file with the extension .MAP. You can override the default name by specifying a new map filename in the *mapfile* field or in response to the `List File` prompt.

12.5.15 The /MAP Option

Option

/M[[AP]]

The /MAP option adds to the map file all public (global) symbols defined in object files. When /MAP is specified, the map file contains a list of all the symbols sorted by name and a list of all the symbols sorted by address. If you do not use this option, the map file contains only a list of segments. If you also want to add line numbers to the map file, use the /LINE option.

The /MAP option causes a map file to be created even if you did not explicitly tell the linker to do so. By default, the map file is given the same base name as the executable file with the extension .MAP. You can override the default name by specifying a new map filename in the *mapfile* field or in response to the `List File` prompt.

Under some circumstances, adding symbols slows the linking process. If this is a problem, do not use /MAP.

12.5.16 The /NOD Option

Option

/NOD[[EFAULTLIBRARYSEARCH]][[:*libraryname*]]

The /NOD option tells LINK not to search default libraries named in object files. Specifying *libraryname* tells LINK to search all libraries named in the object files except *libraryname*. If you want LINK to ignore more than one library, specify /NOD once for each library. To tell LINK to ignore all default libraries, specify /NOD without a *libraryname*.

High-level-language object files usually must be linked with a run-time library to produce an executable file. Therefore, if you use the /NOD option, you must also use the *libraries* field to specify an alternate library that resolves the external references in the object files.

12.5.17 The /NOE Option

Option

/NOE[[XTDICTIONARY]]

The /NOE option prevents the linker from searching extended dictionaries, which are lists of symbol locations in libraries created with LIB. The linker consults extended dictionaries to speed up library searches.

Using /NOE slows the linker. Use this option when you are redefining a symbol or function defined in a library and you get the error

```
L2044 symbol multiply defined, use /NOE
```

12.5.18 The /NOFARCALL Option

Option

/NOF[[ARCALLTRANSLATION]]

The /NOFARCALL option turns off far-call optimization (translation). Far-call optimization is off by default. However, if an environment variable (such as LINK or FL) includes the /FARCALL option, you can use /NOFARCALL to override /FARCALL.

12.5.19 The /NOGROUP Option

Option

/NOG[ROUPASSOCIATION]

The /NOGROUP option ignores group associations when assigning addresses to data and code items. It is provided primarily for compatibility with previous versions of the linker (2.02 and earlier) and early versions of Microsoft compilers. This option is valid only for assembly-language programs that create DOS .EXE files.

12.5.20 The /NOI Option

Option

/NOI[GNORECASE]

This option preserves case in identifiers. By default, LINK treats uppercase and lowercase letters as equivalent. Thus `ABC`, `Abc`, and `abc` are considered the same name. When you use the /NOI option, the linker distinguishes between uppercase and lowercase, and considers these identifiers to be three different names.

In most high-level languages, identifiers are not case sensitive, so this option has no effect. However, case is significant in C. It's a good idea to use this option with C programs to catch misnamed identifiers.

12.5.21 The /NOLOGO Option

Option

/NOL[OGO]

The /NOLOGO option suppresses the copyright message displayed when LINK starts. This option has no effect if not specified first on the command line or in the LINK environment variable.

12.5.22 The /NONULLS Option

Option

/NON[ULLSDOSSEG]

The /NONULLS option arranges segments in the same order they are arranged by the /DOSSEG option. The only difference is that the /DOSSEG option inserts 16 null bytes at the beginning of the _TEXT segment (if it is defined), but /NONULLS does not insert the extra bytes.

If both the /DOSSEG and /NONULLS options are given, the /NONULLS option takes precedence. You can therefore use /NONULLS to override the DOSSEG comment record found in run-time libraries. This option is for segmented executable files.

12.5.23 The /NOPACKC Option

Option

/NOP[[ACKCODE]]

This option turns off code-segment packing. Code-segment packing is normally off by default. However, if an environment variable (such as LINK or FL) includes the /PACKC option to turn on code-segment packing, you can use /NOPACKC to override /PACKC.

12.5.24 The /OV Option

Option

/O[[VERLAYINTERRUPT]]:*number*

This option sets an interrupt number for passing control to overlays. By default, the interrupt number used for passing control to overlays is 63 (3F hexadecimal). The /OV option allows you to select a different interrupt number. This option is valid only when linking DOS programs.

The *number* can be any number from 0 to 255, specified in decimal format or in C-language notation. Numbers that conflict with DOS interrupts can be used; however, their use is not advised. You should use this option only when you want to use overlays with a program that already reserves interrupt 63 for some other purpose.

12.5.25 The /PACKC Option

Option

/PACKC[[ODE]][[:*number*]]

The /PACKC option turns on code-segment packing. The linker packs code segments by grouping neighboring code segments that have the same attributes. Segments in the same group are assigned the same segment address; offset addresses are adjusted accordingly. All items have the same physical address whether or not the /PACKC option is used. However, /PACKC changes the segment and offset addresses so that all items in a group share the same segment.

The *number* specifies the maximum size of groups formed by /PACKC. The linker stops adding segments to a group when it cannot add another segment without exceeding *number*; then it starts a new group. The default segment size

without /PACKC (or when /PACKC is specified without *number*) is 65,500 bytes (64K – 36 bytes).

The /PACKC option produces slightly faster and more compact code. It affects only programs with multiple code segments. This option is off by default and, if specified in an environment variable, can be overridden with the /NOPACKC option.

Code-segment packing provides more opportunities for far-call optimization (which is enabled with the /FARCALL option). The /FARCALL and /PACKC options together produce faster and more compact code. However, this combination is not recommended for Windows applications.

Use caution when packing assembly-language programs.

Object code created by Microsoft compilers can safely be linked with the /PACKC option. This option is unsafe only when used with assembly-language programs that make assumptions about the relative order of code segments. For example, the following assembly code attempts to calculate the distance between `CSEG1` and `CSEG2`. This code produces incorrect results when used with /PACKC, because /PACKC causes the two segments to share the same segment address. Therefore, the procedure would always return zero.

```
CSEG1       SEGMENT PUBLIC 'CODE'
.
.
.
CSEG1       ENDS

CSEG2       SEGMENT PARA PUBLIC 'CODE'
            ASSUME  cs:CSEG2

; Return the length of CSEG1 in AX

codesize    PROC  NEAR
            mov   ax, CSEG2  ; Load para address of CSEG1
            sub   ax, CSEG1  ; Load para address of CSEG2
            mov   cx, 4      ; Load count
            shl   ax, cl     ; Convert distance from paragraphs
                             ;  to bytes
codesize    ENDP

CSEG2       ENDS
```

12.5.26 The /PACKD Option

Option

/PACKD[[ATA]][[:*number*]]

The /PACKD option turns on data-segment packing. The linker considers any segment definition with a class name that does not end in `CODE` as a data segment. Adjacent data-segment definitions are combined into the same physical

segment. The linker stops adding segments to a group when it cannot add another segment without exceeding *number* bytes; then it starts a new group. The default segment size without /PACKD (or when /PACKD is specified without *number*) is 65,536 bytes (64K).

The /PACKD option produces slightly faster and more compact code. It affects only programs with multiple data segments and is valid for OS/2 and Windows programs only. It might be necessary to use the /PACKD option to get around the limit of 255 physical data segments per executable file imposed by OS/2 and Windows. Try using /PACKD if you get the following LINK error:

```
L1073 file-segment limit exceeded
```

This option may not be safe with other compilers that do not generate fixup records for all far data references.

12.5.27 The /PADC Option

Option

/PADC[[ODE]][[:*padsize*]]

The /PADC option adds filler bytes to the end of each code segment for use when later linking with ILINK. If you use /PADC, you must also specify the /INCR option.

The *padsize* is optional; the default is 0 bytes. If incremental linking fails, you can specify a *padsize* in decimal format or C-language notation. For example, `/PADC:256` adds an additional 256 bytes to each code segment. (You can also use `0400` or `0x100` to specify 256 bytes.)

The linker recognizes code segments as segment definitions with class names that end in `CODE`. Microsoft high-level languages automatically use this declaration for code segments. Code padding is not usually necessary for programs with multiple code segments but is recommended for mixed-model programs, programs with one code segment, and assembly-language programs in which code segments are grouped.

12.5.28 The /PADD Option

Option

/PADD[[ATA]][[:*padsize*]]

The /PADD option adds filler bytes to the end of each data segment to permit subsequent linking with ILINK. If you use /PADD, you must also specify the /INCR option.

The *padsize* is optional; the default is 16 bytes. The /INCR option itself adds 16 bytes. This default padding is usually sufficient for successful incremental

linking. If incremental linking fails, you can specify a *padsize* in decimal format or C-language notation. (If you specify too large a padsize, you might exceed the 64K limitation on the size of the default data segment.) For example, `/PADD:32` adds an additional 32 bytes to each data segment. (You can also use `040` or `0x20` to specify 32 bytes.)

12.5.29 The /PAUSE Option

Option

/PAU[[SE]]

The /PAUSE option pauses the session before LINK writes the executable file or DLL to disk. This option is supplied for compatibility with machines that have two floppy drives but no hard disk. It allows you to swap floppy disks before LINK writes the executable file.

If you specify the /PAUSE option, LINK displays the following message before it creates the main output:

```
About to generate .EXE file
Change diskette in drive letter and press <ENTER>
```

The *letter* is the current drive. LINK resumes processing when you press ENTER.

Do not remove a disk that contains either the map file or the temporary file. If LINK creates a temporary file on the disk you plan to remove, terminate the LINK session and rearrange your files so that the temporary file is on a disk that does not need to be removed. For more information on how LINK determines where to put the temporary file, see Section 12.9, "LINK Temporary Files."

12.5.30 The /PM Option

Option

/PM[[TYPE]]:*type*

This option specifies the type of Windows or OS/2 application being generated. The /PM option is equivalent to including a type specification in the **NAME** statement in a module-definition file.

The *type* field can take one of the following values:

Value	Description
PM	Presentation Manager (PM) or Windows application. The application uses the API provided by PM or Windows and must be executed in the PM or Windows environment. This is equivalent to **NAME WINDOWAPI**.
VIO	Character-mode application to run in a text window in the PM or Windows session. This is equivalent to **NAME WINDOWCOMPAT**.
NOVIO	The default. Character-mode application that must run full screen and cannot run in a text window in PM or in Windows. This is equivalent to **NAME NOTWINDOWCOMPAT**.

12.5.31 The /Q Option

Option

/Q[[UICKLIBRARY]]

The /Q option directs the linker to produce a "Quick library" instead of an executable file. A Quick library is similar to a standard library in that both contain routines that can be called by a program. However, a standard library is linked with a program at link time; in contrast, a Quick library is linked with a program at run time.

When /Q is specified, the *exefile* field refers to a Quick library instead of an application. The default extension for this field is then .QLB instead of .EXE.

Quick libraries can be used only with programs created with Microsoft QuickBasic or early versions of Microsoft QuickC. These programs have the special code that loads a Quick library at run time.

12.5.32 The /SEG Option

Option

/SE[[GMENTS]][[:*number*]]

The /SEG option sets the maximum number of program segments. The default without /SEG or *number* is 128. You can specify *number* as any value from 1 to 16,384 in individual format or C-language notation. However, the number of segment definitions is constrained by available memory.

LINK must allocate some memory to keep track of information for each segment; the larger the *number* you specify, the less free memory LINK has to run in. A relatively low segment limit (such as the 128 default) reduces the chance

LINK will run out of memory. For programs with fewer than 128 segments, you can minimize LINK's memory requirements by setting *number* to reflect the actual number of segments in the program. If a program has more than 128 segments, however, you must set a higher value.

If the number of segments allocated is too high for the amount of memory available while linking, LINK displays the error message

```
L1054 requested segment limit too high
```

When this happens, try linking again after setting /SEG to a smaller *number*.

12.5.33 The /STACK Option

Option

/ST[[ACK]]:*number*

The /STACK option lets you change the stack size from its default value of 2,048 bytes. The *number* is any positive value in decimal or C-language notation, up to 64K.

Programs that pass large arrays or structures by value or with deeply nested subroutines may need additional stack space. In contrast, if your program uses the stack very little, you might be able to save space by decreasing the stack size. If a program fails with a stack-overflow message, try increasing the size of the stack.

NOTE You can also use the EXEHDR utility to change the default stack size by modifying the executable-file header.

12.5.34 The /TINY Option

Option

/T[[INY]]

The /TINY option produces a .COM file instead of an .EXE file. The default extension of the output file is .COM. When the /CO option is used with /TINY, debug information is put in a separate file with the same base name as the .COM file and with the .DBG extension.

Not every program can be linked in the .COM format. The following restrictions apply:

- The program must consist of only one physical segment. You can declare more than one segment in assembly-language programs; however, the segments must be in the same group.
- The code must not use far references.

- Segment addresses cannot be used as immediate data for instructions. For example, you cannot use the following instruction:

```
mov     ax, CODESEG
```

- Windows and OS/2 programs cannot be converted to a .COM format.

12.5.35 The /W Option

Option

/W[[ARNFIXUP]]

The /W option issues the `L4000` warning when LINK uses a displacement from the beginning of a group in determining a fixup value. This option is provided because early versions of the Windows linker (LINK4) performed fixups without this displacement. This option is for linking segmented executable files.

12.5.36 The /? Option

Option

/?

The /? option displays a brief summary of LINK command-line syntax and options.

12.6 Setting Options with the LINK Environment Variable

You can use the LINK environment variable to set options that will be in effect each time you link. (Microsoft compilers such as CL, FL, and ML also use the options in the LINK environment variable.)

12.6.1 Setting the LINK Environment Variable

You set the LINK environment variable with the following operating-system command:

`SET LINK=`*options*

LINK expects to find *options* listed in the variable exactly as you would type them in fields on the command line, in response to a prompt, or in a response file. It does not accept input for other fields; filenames in the LINK variable cause an error.

Example

```
SET LINK=/NOI /SEG:256 /CO
LINK TEST;
LINK /NOD PROG;
```

In the example above, the commands are specified at the system prompt. The file TEST.OBJ is linked using the options `/NOI`, `/SEG:256`, and `/CO`. The file PROG.OBJ is then linked with the option `/NOD`, in addition to `/NOI`, `/SEG:256`, and `/CO`.

12.6.2 Behavior of the LINK Environment Variable

You can specify options on the LINK command line or in a response file in addition to those in the LINK environment variable. If an option appears both in an input field and in the LINK variable, the input-field option overrides any environment-variable option it conflicts with. For example, the command-line option /SEG:512 overrides the environment-variable option /SEG:256.

12.6.3 Clearing the LINK Environment Variable

You must reset the LINK environment variable to prevent LINK from using its options. To clear the LINK variable, use the operating-system command

```
SET LINK=
```

To see the current setting of the LINK variable, type `SET` at the operating-system prompt.

12.7 Using Overlays under DOS

LINK can create DOS programs with "overlays." Overlays allow sections of a program to be loaded into memory only as needed. This permits running a program that would otherwise be too large to fit in available memory. Overlay programs execute more slowly, however, since the various program modules must be swapped into and out of memory.

The CodeView debugger is compatible with overlaid modules. If you use CodeView to debug a program that has an overlay containing more than one code segment, you will see only the identifiers contained in the first segment of the overlay.

12.7.1 Restrictions on Overlays

Not all programs can use overlays. You will probably need to reorganize the code to accommodate the limitations explained in this section. Even after reorganization, some programs might not be convertible to overlay form or might not show a significant reduction in the amount of memory needed to execute them.

Consider the following restrictions before trying to overlay a program:

- You can use overlays only in programs with multiple code segments, because separate segment names are needed for overlays. Only code is overlaid, not data. The data becomes part of the "root" section of the program that is always in memory.
- Only 255 overlays can be specified. The program can define only 255 logical segments (segments with different names). This limits the total size of an overlaid program to 16 megabytes.
- Only one overlay (in addition to the root) can be in memory at any one time. You must structure your program accordingly.
- Duplicate names for different overlays are not supported; each module can appear only once in a program.
- You must use far call/return instructions to transfer control between overlaid files. You cannot overlay files containing near routines if other overlays call those routines.
- You cannot jump out of or into overlaid files using the **longjmp** C-library function. You can, however, use long jumps within an overlaid file.
- You cannot use a function pointer to call a routine out of or into overlaid files. You can, however, use a function pointer to call a routine within an overlaid file.
- You cannot use the same public name in different overlays.
- The code required to manage overlays adds about 2K to 3K to the size of the root module.

WARNING Never rename an executable program file containing overlays if it is to run under DOS 2.*x* and earlier. LINK records the .EXE filename in the program file. If you rename the file, the overlay manager may not be able to locate the proper file. You can rename an .EXE file that will run under DOS 3.*x* and later.

12.7.2 Specifying Overlays

Specify overlays by enclosing object-file (and possibly load-library) names in parentheses in the *objfiles* field. Each group of object files bracketed by parentheses represents one overlay. Overlays cannot be nested.

The remaining modules (those not in parentheses), and any drawn from the run-time libraries, constitute the resident (or root) part of your program. The entry point to the program (for example, `main()` in a C program, or `PROGRAM` in a FORTRAN program) must be in the root.

Example

The following list of files contains three overlays:

```
a + (b+c) + (d+e) + f + (g)
```

In this example, the groups `(b+c)`, `(d+e)`, and `(g)` are overlays. The remaining files `a` and `f` and any modules from libraries in the *libraries* field remain memory-resident throughout the execution of the program.

It is important to remember that whichever object file first defines a segment gets all contributions to that segment. In the example above, if D.OBJ and F.OBJ both define the same segment, the contribution from F.OBJ to that segment goes into the `(d+e)` overlay rather than into the root.

12.7.3 How Overlays Work

Programs that use overlays require the overlay-manager code to handle module swapping. This code is included as part of the standard libraries for Microsoft high-level languages. If you specify overlays during linking, the code for the overlay manager is automatically linked with the rest of your program.

LINK produces only one .EXE file. The overlay manager searches for this file whenever another overlay needs to be loaded. It first searches in the current directory. If the file is not there, the manager then searches the directories in the PATH environment variable. If the overlay manager still cannot find the file, it prompts for the pathname.

Example

Assume that an executable program called PAYROLL.EXE uses overlays and does not exist in either the current directory or the directories specified by PATH. If you run PAYROLL.EXE by entering a complete path specification, the overlay manager displays the following message when it attempts to load an overlay file:

```
Cannot find PAYROLL.EXE
Please enter new program spec:
```

You can then enter the drive or directory, or both, where PAYROLL.EXE is located. For example, if the file is located in directory \EMPLOYEE\DATA\ on drive B, enter `B:\EMPLOYEE\DATA\`; if the current drive is B, you can enter just `\EMPLOYEE\DATA\`.

If you later remove the disk in drive B and the overlay manager needs the overlay again, it does not find PAYROLL.EXE and displays the following message:

```
Please insert diskette containing B:\EMPLOYEE\DATA\PAYROLL.EXE
in drive B: and strike any key when ready.
```

After the overlay file has been read from the disk, the overlay manager displays the following message:

```
Please restore the original diskette.
Strike any key when ready.
```

12.7.4 Overlay Interrupts

LINK replaces far calls to routines in overlays with interrupts (followed by the module identifier and offset). By default, the interrupt number is 63 (3F hexadecimal). You can use the /OV option to change the interrupt number.

12.8 Linker Operation under DOS

LINK performs the following steps to produce a DOS executable file:

1. Reads the object modules submitted
2. Searches the given libraries, if necessary, to resolve external references
3. Assigns addresses to segments
4. Assigns addresses to public symbols
5. Reads code and data in the segments
6. Reads all relocation references in object modules
7. Performs fixups
8. Outputs an executable file (executable image and relocation information)

Steps 5, 6, and 7 are performed iteratively—that is, LINK repeats these steps as many times as required before it progresses to step 8.

The "executable image" contains the code and data that constitute the executable file. The "relocation information" is a list of references relative to the start of the

program, each of which changes when the executable image is loaded into memory and an actual address for the entry point is assigned.

The following sections explain the process LINK uses to concatenate segments and resolve references to items in memory.

12.8.1 Segment Alignment

LINK uses each segment's alignment type to set the starting address for the segment. The alignment types are **BYTE**, **WORD**, **DWORD**, **PARA**, and **PAGE**. These correspond to starting addresses at byte, word, doubleword, paragraph, and page boundaries, representing addresses that are multiples of 1, 2, 4, 16, and 256, respectively. The default alignment is **PARA**.

When LINK encounters a segment, it checks the alignment type before copying the segment to the executable file. If the alignment is **WORD**, **DWORD**, **PARA**, or **PAGE**, LINK checks the executable image to see if the last byte copied ends at an appropriate boundary. If not, LINK pads the image with extra null bytes.

12.8.2 Frame Number

LINK computes a starting address for each segment in a program. The starting address is based on a segment's alignment and the sizes of the segments already copied to the executable file. The address consists of an offset and a "canonical frame number." The canonical frame number specifies the address of the first paragraph in memory containing one or more bytes of the segment. (A paragraph is 16 bytes of memory; therefore, to compute a physical location in memory, multiply the frame number by 16 and add the offset.) The offset is the number of bytes from the start of the paragraph to the first byte in the segment. For **BYTE**, **WORD**, and **DWORD** alignments, the offset may be nonzero. The offset is always zero for **PARA** and **PAGE** alignments. (An offset of zero means that the physical location is an exact multiple of 16.)

The frame number of a segment can be obtained from the map file created by LINK. The first four digits of the start address give the frame number in hexadecimal. For example, a start address of `0C0A6` gives a frame number of `0C0A`.

12.8.3 Segment Order

LINK copies segments to the executable file in the same order that it encounters them in the object files. This order is maintained throughout the program unless LINK encounters two or more segments having the same class name. Segments having identical class names belong to the same class type and are copied as a contiguous block to the executable file.

The /DOSSEG option might change the way in which segments are ordered.

12.8.4 Combined Segments

LINK uses combine types to determine whether two or more segments sharing the same segment name should be combined into one large segment. The valid combine types are **PUBLIC**, **STACK**, **COMMON**, and **PRIVATE**.

If a segment has combine type **PUBLIC**, LINK automatically combines it with any other segments having the same name and belonging to the same class. When LINK combines segments, it ensures that the segments are contiguous and that all addresses in the segments can be accessed using an offset from the same frame address. The result is the same as if the segment were defined as a whole in one source file.

LINK preserves each individual segment's alignment type. This means that even though the segments belong to a single large segment, the code and data in the segments do not lose their original alignment. If the combined segments exceed 64K, LINK displays an error message.

If a segment has combine type **STACK**, LINK carries out the same combine operation as for **PUBLIC** segments. The only exception is that **STACK** segments cause LINK to copy an initial stack-pointer value to the executable file. This stack-pointer value is the offset to the end of the first stack segment (or combined stack segment) encountered.

If a segment has combine type **COMMON**, LINK automatically combines it with any other segments having the same name and belonging to the same class. When LINK combines **COMMON** segments, however, it places the start of each segment at the same address, creating a series of overlapping segments. The result is a single segment no larger than the largest segment combined.

A segment has combine type **PRIVATE** only if no explicit combine type is defined for it in the source file. LINK does not combine private segments.

12.8.5 Groups

Groups allow segments to be addressed relative to the same frame address. When LINK encounters a group, it adjusts all memory references to items in the group so that they are relative to the same frame address.

Segments in a group do not have to be contiguous, belong to the same class, or have the same combine type. The only requirement is that all segments in the group fit within 64K.

Groups do not affect the order in which the segments are loaded. Unless you use class names and enter object files in the right order, there is no guarantee the segments will be contiguous. In fact, LINK may place segments that do not belong to the group in the same 64K of memory. LINK does not explicitly check that all segments in a group fit within 64K of memory; however, LINK is likely to encounter a fixup-overflow error if this requirement is not met.

12.8.6 Fixups

Once the starting address of each segment in a program is known and all segment combinations and groups have been established, LINK can "fix up" any unresolved references to labels and variables. To fix up unresolved references, LINK computes an appropriate offset and segment address and replaces the temporary values generated by the assembler with the new values.

LINK carries out fixups for the types of references shown in Table 12.1.

The size of the value to be computed depends on the type of reference. If LINK discovers an error in the anticipated size of a reference, it displays a fixup-overflow message. This can happen, for example, if a program attempts to use a 16-bit offset to reach an instruction which is more than 64K away. It can also occur if all segments in a group do not fit within a single 64K block of memory.

Table 12.1 LINK Fixups

Type	Location of Reference	LINK Action
Short	In **JMP** instructions that attempt to pass control to labeled instructions in the same segment or group. The target instruction must be no more than 128 bytes from the point of reference.	Computes a signed, eight-bit number for the reference and displays an error message if the target instruction belongs to a different segment or group (has a different frame address), or if the target is more than 128 bytes away in either direction.
Near self-relative	In instructions that access data relative to the same segment or group.	Computes a 16-bit offset for the reference and displays an error if the data are not in the same segment or group.
Near segment-relative	In instructions that attempt to access data in a specified segment or group, or relative to a specified segment register.	Computes a 16-bit offset for the reference and displays an error message if the offset of the target within the specified frame is greater than 64K or less than 0, or if the beginning of the canonical frame of the target is not addressable.
Long	In **CALL** instructions that attempt to access an instruction in another segment or group.	Computes a 16-bit frame address and 16-bit offset for this reference, and displays an error message if the computed offset is greater than 64K or less than 0, or if the beginning of the canonical frame of the target is not addressable.

12.9 LINK Temporary Files

LINK uses available memory during the linking session. If LINK runs out of memory, it creates a disk file to hold intermediate files. LINK deletes this file when it finishes.

When the linker creates a temporary disk file, you see the message

```
Temporary file tempfile has been created.
Do not change diskette in drive, letter.
```

In the message displayed above, *tempfile* is the name of the temporary file and *letter* is the drive containing the temporary file. (The second line appears only for a floppy drive.)

After this message appears, do not remove the disk from the drive specified by *letter* until the link session ends. If the disk is removed, the operation of LINK is unpredictable, and you might see the following message:

```
Unexpected end-of-file on scratch file
```

If this happens, run LINK again.

Location of the Temporary File

If the TMP environment variable defines a temporary directory, LINK creates temporary files there. If the TMP environment variable is undefined or the temporary directory doesn't exist, LINK creates temporary files in the current directory.

Name of the Temporary File

When running under OS/2 or DOS version 3.0 or later, LINK asks the operating system to create a temporary file with a unique name in the temporary-file directory.

Under DOS versions earlier than 3.0, LINK creates a temporary file named VM.TMP. Do not use this name for your files. LINK generates an error message if it encounters an existing file with this name.

12.10 LINK Exit Codes

LINK returns an exit code (also called return code or error code) that you can use to control the operation of batch files or makefiles.

Code	Meaning
0	No error.
2	Program error. Commands or files given as input to the linker produced the error.
4	System error. The linker ■ Ran out of space on output files ■ Was unable to reopen the temporary file ■ Experienced an internal error ■ Was interrupted by the user

12.11 Related Topics in Online Help

In addition to information covered in this chapter, information on the following topics can be found in online help.

Topic	Access
Syntax and procedural information on LINK, BIND, and LIB	Choose these topics from the "Microsoft Advisor Contents" screen
Syntax and procedural information on EXEHDR	Choose "Miscellaneous" from the list of utilities on the "Microsoft Advisor Contents" screen

Chapter 13

Module-Definition Files

This chapter describes the contents of a module-definition file. It begins with a brief overview of the purpose of module-definition files. The rest of the chapter discusses each statement in a module-definition file and describes syntax rules, argument fields, attributes, and keywords for each statement.

13.1 Overview

A module-definition file is a text file that describes the name, attributes, exports, imports, system requirements, and other characteristics of an application or dynamic-link library (DLL) for OS/2 or Microsoft Windows. This file is required for DLLs and is optional (but desirable) for OS/2 and Windows applications.

You use module-definition files in two situations:

- You can specify a module-definition file in LINK's *deffile* field. The module-definition file gives LINK the information it needs to determine how to set up the application or DLL it creates.
- You can provide LINK with the needed information when creating an application by using the Microsoft Import Library Manager utility (IMPLIB) to create an import library from a module-definition file (or from the DLL created by a module-definition file). You then specify the import library in LINK's *libraries* field.

For more information about IMPLIB, see online help.

13.2 Module Statements

A module-definition file contains one or more "module statements." Each module statement defines an attribute of the executable file, such as its name, the attributes of program segments, and the number and names of exported and imported functions and data. Table 13.1 summarizes the purpose of the module statements and shows the order in which they are discussed in this chapter.

Table 13.1 Module Statements

Statement	Purpose
NAME	Names the application (no library created)
LIBRARY	Names the DLL (no application created)
DESCRIPTION	Embeds text in the application or DLL
STUB	Adds a DOS executable file to the beginning of the file
EXETYPE	Identifies the target operating system
PROTMODE	Specifies a protected-mode application or DLL
REALMODE	Supported for compatibility
STACKSIZE	Sets stack size in bytes
HEAPSIZE	Sets local heap size in bytes
CODE	Sets default attributes for all code segments
DATA	Sets default attributes for all data segments
SEGMENTS	Sets attributes for specific segments
OLD	Preserves ordinals from a previous DLL
EXPORTS	Defines exported functions
IMPORTS	Defines imported functions

13.2.1 Syntax Rules

The syntax rules in this section apply to all statements in a module-definition file. Other rules specific to each statement are described in the sections that follow.

- Statement and attribute keywords are not case sensitive. A statement keyword can be preceded by spaces and tabs.
- A **NAME** or **LIBRARY** statement, if used, must precede all other statements.

- Most statements appear at most once in a file and accept one specification of parameters and attributes. The specification follows the statement keyword on the same or subsequent line(s). If repeated with a different specification later in the file, the later statement overrides the earlier one.
- The **SEGMENTS**, **EXPORTS**, and **IMPORTS** statements can appear more than once in the file and take multiple specifications, each on its own line. The statement keyword must appear once before the first specification and can be repeated before each additional specification.
- Comments in the file are designated by a semicolon (;) at the beginning of each comment line. A comment cannot share a line with part or all of a statement but can appear between lines of a multiline statement.
- Numeric arguments can be specified in decimal or in C-language notation.
- Name arguments cannot match a reserved word.

Example

The sample module-definition file below gives a description for a DLL. This sample file includes one comment and five statements.

```
; Sample module-definition file

LIBRARY

DESCRIPTION 'Sample dynamic-link library'

CODE        PRELOAD

STACKSIZE   1024

EXPORTS
    Init   @1
    Begin  @2
    Finish @3
    Load   @4
    Print  @5
```

13.2.2 Reserved Words

The following words are reserved by the linker for use in module-definition files. These names cannot be used as arguments in module-definition statements.

CLASS	IOPL	PRELOAD
CODE	LIBRARY	PRIVATELIB
CONFORMING	LOADONCALL	PROTMODE
DATA	LONGNAMES	PURE
DESCRIPTION	MAXVAL	READONLY
DISCARDABLE	MOVABLE	READWRITE
DOS4 *	MOVEABLE	REALMODE
EXECUTE-ONLY	MULTIPLE	RESIDENT
EXECUTEONLY	NAME	RESIDENTNAME
EXECUTEREAD	NEWFILES	SEGMENTS
EXETYPE	NODATA	SHARED
EXPORTS	NOIOPL	SINGLE
FIXED	NONCONFORMING	STACKSIZE
HEAPSIZE	NONDISCARDABLE	STUB
HUGE *	NONE	UNKNOWN
IMPORTS	NONSHARED	WINDOWAPI
IMPURE	NOTWINDOWCOMPAT	WINDOWCOMPAT
INITGLOBAL	OLD	WINDOWS
INITINSTANCE	OS2	

* DOS4 and HUGE are obsolete but are still reserved by the linker.

In addition to the words listed above, the following words are reserved for use by future or other versions of the linker and should be avoided.

ALIAS	INVALID	PERMANENT
CONTIGUOUS	MIXED1632	PHYSICAL
DEV386	NONAME	RESIDENT
DEVICE	NONPERMANENT	SWAPPABLE
DYNAMIC	OBJECTS	TERMINSTANCE
INCLUDE	ORDER	VIRTUAL

13.3 The NAME Statement

The **NAME** statement identifies the executable file as an application (rather than a DLL). It can also specify the name and application type. The **NAME** or **LIBRARY** statement must precede all other statements. If **NAME** is specified, the **LIBRARY** statement cannot be used. If neither is used, the default is **NAME** and LINK creates an application.

Syntax

NAME [[*appname*]] [[*apptype*]] [[**NEWFILES**]]

Remarks

The fields can appear in any order.

If *appname* is specified, it becomes the name of the application as it is known by OS/2 or Windows. This name can be any valid filename. If *appname* contains a space, begins with a nonalphabetic character, or is a reserved word, surround *appname* with double quotation marks. The name cannot exceed 255 characters (not including surrounding quotation marks). If *appname* is not specified, the base name of the executable file becomes the name of the application.

If *apptype* is specified, it defines the type of application. This information is kept in the executable-file header. The *apptype* field can take one of the following values:

Value	Description
WINDOWAPI	Presentation Manager (PM) or Windows application. The application uses the API provided by PM or Windows and must be executed in the PM or Windows environment. This is equivalent to the LINK option /PM:PM.
WINDOWCOMPAT	Character-mode application to run in a text window in the PM or Windows session. This is equivalent to the LINK option /PM:VIO.
NOTWINDOWCOMPAT	The default. Character-mode application that must run full screen and cannot run in a text window in PM or Windows. This is equivalent to the LINK option /PM:NOVIO.

Specify **NEWFILES** to tell the operating system that the application supports long filenames and extended file attributes (available under OS/2 version 1.2 and later). The synonym **LONGNAMES** is supported for compatibility.

Example

The example below assigns the name `calendar` to an application that can run in a text window in PM or Windows:

```
NAME calendar WINDOWCOMPAT
```

13.4 The LIBRARY Statement

The **LIBRARY** statement identifies the executable file as a DLL. It can also specify the name of the library and the type of library-module initialization required. The **NAME** or **LIBRARY** statement must precede all other statements. If **LIBRARY** is specified, the **NAME** statement cannot be used. If neither is used, the default is **NAME**.

Syntax

LIBRARY [[*libraryname*]] [[*initialization*]] [[**PRIVATELIB**]]

Remarks

The fields can appear in any order.

If *libraryname* is specified, it becomes the name of the library as it is known by OS/2 or Windows. This name can be any valid filename. If *libraryname* contains a space, begins with a nonalphabetic character, or is a reserved word, surround the name with double quotation marks. The name cannot exceed 255 characters. If *libraryname* is not given, the base name of the DLL file becomes the name of the library.

If *initialization* is specified, it determines the type of initialization required. The *initialization* field can take one of the following values:

Value	Description
INITGLOBAL	The default. The library-initialization routine is called only when the library is initially loaded into memory.
INITINSTANCE	The library-initialization routine is called each time a new process gains access to the DLL. This keyword applies only to OS/2.

If **PRIVATELIB** is specified, it tells Windows that only one application may use the DLL.

Example

The following example assigns the name `calendar` to the DLL being defined and specifies that library initialization is performed each time a new process gains access to `calendar`:

```
LIBRARY calendar INITINSTANCE
```

13.5 The DESCRIPTION Statement

The **DESCRIPTION** statement inserts specified text into the application or DLL. This statement is useful for embedding source-control or copyright information into a file.

Syntax

DESCRIPTION '*text*'

Remarks

The *text* is a string of up to 255 characters enclosed in single or double quotation marks (' or "). To include a literal quotation mark in the text, either specify two consecutive quotation marks of the same type or enclose the text with the other type of quotation mark. If a **DESCRIPTION** statement is not specified, the default text is the name of the main output file as specified in LINK's *exefile* field. You can view this string by using the Microsoft EXE File Header Utility (EXEHDR).

The **DESCRIPTION** statement is different from a comment. A comment is a line that begins with a semicolon (;). Comments are not placed in the application or library.

Example

The following example inserts the text `Tester's Version, Test "A"`, including a literal single quotation mark and a pair of literal double quotation marks, into the application or DLL being defined:

```
DESCRIPTION "Tester's Version, Test ""A"""
```

13.6 The STUB Statement

The **STUB** statement adds a DOS executable file to the beginning of an OS/2 or Windows application or DLL. The stub is invoked whenever the file is executed under DOS. Usually, the stub displays a message and terminates execution. By default, LINK adds a standard stub for this purpose. Use the STUB statement when creating a dual-mode program.

Syntax

STUB {'*filename*' | **NONE**}

Remarks

The *filename* specifies the DOS executable file to be added. LINK searches for *filename* first in the current directory and then in directories specified with the PATH environment variable. The *filename* must be surrounded by single or double quotation marks (' or ").

The alternate specification **NONE** prevents LINK from adding a default stub. This saves space in the application or DLL, but the resulting file will hang the system if loaded in DOS.

Example

The following example inserts the DOS executable file STOPIT.EXE at the beginning of the application or DLL:

```
STUB 'STOPIT.EXE'
```

The file STOPIT.EXE is executed when you attempt to run the application or DLL under DOS.

13.7 The EXETYPE Statement

The **EXETYPE** statement specifies under which operating system the application or DLL is to run. This statement is optional and provides an additional degree of protection against the program being run under an incorrect operating system.

Syntax

EXETYPE [[**OS2** | **WINDOWS**[[*version*]] | **UNKNOWN**]]

Remarks

The **EXETYPE** keyword is followed by a descriptor of the operating system, either **OS2** (for OS/2 applications and DLLs), **WINDOWS** (for WINDOWS applications and DLLs), or **UNKNOWN** (for other applications). The default without a descriptor or an **EXETYPE** statement is **OS2**.

EXETYPE sets bits in the header which identify the operating system. Operating-system loaders can check these bits.

Windows Programming

The **WINDOWS** descriptor takes an optional version number. Windows reads this number to determine the minimum version of Windows needed to load the application or DLL. For example, if 3.0 is specified, the resulting application or DLL

can run under Windows versions 3.0 and higher. If *version* is not specified, the default is 3.0. The syntax for *version* is

number[[.[[*number*]]]]

where each *number* is a decimal integer.

In Windows programming, use the **EXETYPE** statement with a **PROTMODE** statement to specify an application or DLL that runs only under protected-mode Windows.

13.8 The PROTMODE Statement

The **PROTMODE** statement specifies that the application or DLL runs only under OS/2 or under Windows 3.0 standard mode and 386 enhanced mode. **PROTMODE** lets LINK optimize to reduce both the size of the file on disk and its loading time. However, an OS/2 program created with **PROTMODE** cannot be bound using **BIND**. Use **PROTMODE** in combination with an **EXETYPE WINDOWS** statement to define an application or DLL that runs only under protected-mode Windows.

Syntax

PROTMODE

Example

The following statement combination defines an application that runs only under protected-mode (standard or 386 enhanced) Windows version 3.0:

```
EXETYPE WINDOWS 3.0
PROTMODE
```

13.9 The REALMODE Statement

The **REALMODE** statement specifies that the application runs only in real mode. This statement is supported for compatibility with existing module-definition files. Use **EXETYPE** instead.

Syntax

REALMODE

13.10 The STACKSIZE Statement

The **STACKSIZE** statement specifies the size of the stack in bytes. It performs the same function as LINK's /STACK option. If both are specified, the **STACKSIZE** statement overrides the /STACK option.

Syntax

STACKSIZE *number*

Remarks

The *number* must be a positive integer, in decimal or C-language notation, up to 64K.

Example

The following example allocates 4,096 bytes of stack space:

```
STACKSIZE 4096
```

13.11 The HEAPSIZE Statement

The **HEAPSIZE** statement defines the size of the application or DLL's local heap in bytes. This value affects the size of the default data segment (DGROUP). The default without **HEAPSIZE** is no local heap.

Syntax

HEAPSIZE {*bytes* | **MAXVAL**}

Remarks

The *bytes* field accepts a positive integer in decimal or C-language notation. The limit is **MAXVAL**; if *bytes* exceeds **MAXVAL**, the excess is not allocated.

MAXVAL is a keyword that sets the heap size to 64K minus the size of DGROUP. This is useful in bound applications when you want to force a 64K requirement for DGROUP for the program in DOS. The bound program fails to load if 64K of memory is not available.

Example

The following example sets the local heap to 4,000 bytes:

```
HEAPSIZE 4000
```

13.12 The CODE Statement

The **CODE** statement defines the default attributes for all code segments within the application or DLL. The **SEGMENTS** statement can override this default for one or more specific segments.

Syntax

CODE [[*attribute*...]]

Remarks

This statement accepts several optional attribute fields: *conforming*, *discard*, *executeonly*, *iopl*, *load*, *movable*, and *shared*. Each can appear once, in any order. These fields are described in Section 13.15, "CODE, DATA, and SEGMENTS Attributes."

Example

The following example sets defaults for the program's code segments. No code segments in the program are loaded until accessed, and all require I/O hardware privilege.

```
CODE LOADONCALL IOPL
```

13.13 The DATA Statement

The **DATA** statement defines the default attributes for all data segments within the application or DLL. The **SEGMENTS** statement can override this default for one or more specific segments.

Syntax

DATA [[*attribute*...]]

Remarks

This statement accepts several optional attribute fields: *instance*, *iopl*, *load*, *movable*, *readonly*, and *shared*. Each can appear once, in any order. These fields are described in Section 13.15, "CODE, DATA, and SEGMENTS Attributes."

Example

The example below defines the application's data segment so that it cannot be shared by multiple copies of the program and cannot be written to. By default, the data segment can be read and written to and a new DGROUP is created for each instance of the application.

```
DATA NONSHARED READONLY
```

13.14 The SEGMENTS Statement

The **SEGMENTS** statement defines the attributes of one or more individual segments in the application or DLL. The attributes specified for a specific segment override the defaults set in the **CODE** and **DATA** statements (except as noted below). The total number of segment definitions cannot exceed the number set using LINK's /SEG option. (The default without /SEG is 128.)

The **SEGMENTS** keyword marks the beginning of the segment definitions, where each definition is on its own line. The **SEGMENTS** statement must appear once before the first specification (on the same or preceding line) and can be repeated before each additional specification. **SEGMENTS** statements can appear more than once in the file.

Syntax

SEGMENTS
    [[']]*segmentname*[[']] [[**CLASS** ' *classname* ']] [[*attribute*...]]

Remarks

Each segment definition begins with *segmentname*, optionally enclosed in single or double quotation marks (' or "). The quotation marks are required if *segmentname* is a reserved word.

The **CLASS** keyword optionally specifies the class of the segment. Single or double quotation marks (' or ") are required around *classname*. If you do not use the **CLASS** argument, the linker assumes that the class is **CODE**.

This statement accepts several optional attribute fields: *conforming*, *discard*, *executeonly*, *iopl*, *load*, *movable*, *readonly*, and *shared*. Each can appear once, in any order. These fields are described in the next section, "CODE, DATA, and SEGMENTS Attributes."

Example

The following example specifies segments named `cseg1`, `cseg2`, and `dseg`. The first segment is assigned the class `mycode` and the second is assigned **CODE** by default. Each segment is given different attributes.

```
SEGMENTS
    cseg1 CLASS 'mycode' IOPL
    cseg2                EXECUTEONLY PRELOAD CONFORMING
    dseg  CLASS 'data'   LOADONCALL READONLY
```

13.15 CODE, DATA, and SEGMENTS Attributes

The following attribute fields apply to the **CODE**, **DATA**, and **SEGMENTS** statements previously described. Refer to "Remarks" in each of the previous sections for the attribute fields that are used by each statement. Most fields are used by all three statements; others are used as noted. Each field can appear once, in any order.

Listed with each attribute field below are keywords that are legal values for the field, along with descriptions of the field and values. The defaults are noted. If two segments with different attributes are combined into the same group, LINK makes decisions to resolve any conflicts and assumes a set of attributes.

Attribute	Description
conforming	{**CONFORMING** | **NONCONFORMING**}
	For **CODE** and **SEGMENTS** statements only. Determines whether a code segment is an 80286 "conforming" segment for device drivers and system-level code. The *conforming* attribute is for OS/2 only.
	CONFORMING specifies that the segment executes at the caller's privilege level. When IOPL=YES is specified in CONFIG.SYS, no call gates are generated for calls or jumps.
	NONCONFORMING (the default) specifies that the segment can be accessed from Ring 2. When IOPL=YES is specified in CONFIG.SYS, call gates are generated.
	For more information, refer to Intel documentation for the 80286 processor and later.
discard	{**DISCARDABLE** | **NONDISCARDABLE**}
	For **CODE** and **SEGMENTS** statements only. Determines whether a code segment can be discarded from memory to fill a different memory request. If the discarded segment is accessed later, it is reloaded from disk. **NONDISCARDABLE** is the default. The *discard* attribute is for Windows only.

Attribute	Description
executeonly	{**EXECUTEONLY** | **EXECUTEREAD**}
	For **CODE** and **SEGMENTS** statements only. Determines whether a code segment can be read as well as executed.
	EXECUTEONLY specifies that the segment can only be executed. The keyword **EXECUTE-ONLY** is an alternate spelling.
	EXECUTEREAD (the default) specifies that the segment is both executable and readable. This attribute is necessary for a program to run under the Microsoft CodeView debugger.
instance	{**NONE** | **SINGLE** | **MULTIPLE**}
	For the **DATA** statement only. Affects the sharing attributes of the default data segment (DGROUP). This attribute interacts with the *shared* attribute.
	NONE tells the loader not to allocate DGROUP. Use **NONE** when a DLL has no data and uses an application's DGROUP.
	SINGLE (the default for DLLs) specifies that one DGROUP is shared by all instances of the DLL or application.
	MULTIPLE (the default for applications) specifies that DGROUP is copied for each instance of the DLL or application.
iopl	{**IOPL** | **NOIOPL**}
	Determines whether a segment has I/O privilege. OS/2 only.
	IOPL specifies that a code segment has I/O privilege and that a data segment can be accessed only from an IOPL code segment.
	NOIOPL (the default) specifies that there is no I/O privilege for code and no protection for data.
load	{**PRELOAD** | **LOADONCALL**}
	Determines when a segment is loaded.

Attribute	Description
(load, continued)	**PRELOAD** specifies that the segment is loaded when the program starts. **LOADONCALL** (the default) specifies that the segment is not loaded until accessed and only if not already loaded.
movable	{**MOVABLE** | **FIXED**} Determines whether a segment can be moved in memory. Windows only. **FIXED** is the default. An alternative spelling for **MOVABLE** is **MOVEABLE**.
readonly	{**READONLY** | **READWRITE**} For **DATA** and **SEGMENTS** statements only. Determines access rights to a data segment. **READONLY** specifies that the segment can only be read. **READWRITE** (the default) specifies that the segment is both readable and writeable.
shared	{**SHARED** | **NONSHARED**} For real-mode Windows and for **READWRITE** data segments under OS/2 only. Determines whether all instances of the program can share **EXECUTEREAD** and **READWRITE** segments. (Under OS/2, all code segments and **READONLY** data segments are shared.) **SHARED** (the default for DLLs) specifies that one copy of the segment is loaded and shared among all processes accessing the application or DLL. This attribute saves memory and can be used for code that is not self-modifying. An alternate keyword is **PURE**. **NONSHARED** (the default for applications) specifies that the segment must be loaded separately for each process. An alternate keyword is **IMPURE**. This attribute and the *instance* attribute interact for data segments. The *instance* attribute has the keywords **NONE**, **SINGLE**, and **MULTIPLE**. If `DATA SINGLE` is specified, LINK assumes **SHARED**; if `DATA MULTIPLE` is specified, LINK assumes **NONSHARED**. Similarly, `DATA SHARED` forces **SINGLE**, and `DATA NONSHARED` forces **MULTIPLE**.

13.16 The OLD Statement

The **OLD** statement directs the linker to search another DLL for export ordinals. This statement preserves ordinal values used from older versions of a DLL. For more information on ordinals, see the sections below on the **EXPORTS** and **IMPORTS** statements.

Exported names in the current DLL that match exported names in the old DLL are assigned ordinal values from the earlier DLL unless

- The name in the old module has no ordinal value assigned, or
- An ordinal value is explicitly assigned in the current DLL.

Only one DLL can be specified; ordinals can be preserved from only one DLL. The **OLD** statement has no effect on applications.

Syntax

OLD *'filename'*

Remarks

The *filename* specifies the DLL to be searched. It must be enclosed in single or double quotation marks (' or ").

13.17 The EXPORTS Statement

The **EXPORTS** statement defines the names and attributes of the functions and data made available to other applications and DLLs, and of the functions that run with I/O privilege. By default, functions and data are hidden from other programs at run time. A definition is required for each function or data item being exported.

The **EXPORTS** keyword marks the beginning of the export definitions, each on its own line. The **EXPORTS** keyword must appear once before the first definition (on the same or preceding line) and can be repeated before each additional definition. **EXPORTS** statements can appear more than once in the file.

Some languages offer a way to export without using an **EXPORTS** statement. For example, in C the **_exports** keyword makes a function available from a DLL.

Syntax

EXPORTS
entryname[[=*internalname*]] [[**@***ord*[[**RESIDENTNAME**]]]] [[**NODATA**]] [[*pwords*]]

Remarks

The *entryname* defines the function or data-item name as it is known to other programs. The optional *internalname* defines the actual name of the exported function or data item as it appears within the exporting program; by default, this name is the same as *entryname*.

The optional *ord* field defines a function's ordinal position within the module-definition table as an integer from 1 to 65,535. If *ord* is specified, the function can be called by either *entryname* or *ord*. Use of *ord* is faster and can save space.

The optional keyword **RESIDENTNAME** specifies that *entryname* be kept resident in memory at all times. This keyword is applicable only if *ord* is used. (If *ord* is not used, the name *entryname* is always kept in memory.)

The optional keyword **NODATA** specifies that there is no static data in the function.

The *pwords* field specifies the total size of the function's parameters in words. This field is required only if the function executes with I/O privilege. When a function with I/O privilege is called, OS/2 consults *pwords* to determine how many words to copy from the caller's stack to the I/O-privileged function's stack.

Example

The following **EXPORTS** statement defines the three exported functions `SampleRead`, `StringIn`, and `CharTest`. The first two functions can be called either by their exported names or by an ordinal number. In the application or DLL where they are defined, these functions are named `read2bin` and `str1`, respectively. The first and last functions run with I/O privilege and therefore are given with the total size of the parameters.

```
EXPORTS
       SampleRead = read2bin @8              24
       StringIn   = str1     @4 RESIDENTNAME
       CharTest                            6
```

13.18 The IMPORTS Statement

The **IMPORTS** statement defines the names and locations of functions and data items to be imported (usually from a DLL) for use in the application or DLL. A definition is required for each function or data item being imported. This statement is an alternative to resolving references through an import library created by the IMPLIB utility; functions and data items listed in an import library do not require an **IMPORTS** definition.

The **IMPORTS** keyword marks the beginning of the import definitions, each on its own line. The **IMPORTS** keyword must appear once before the first definition on the same or preceding line and can be repeated before each additional definition. **IMPORTS** statements can appear more than once in the file.

Syntax

IMPORTS

[[*internalname*=]]*modulename*.*entry*

Remarks

The *internalname* specifies the function or data-item name as it is used in the importing application or DLL. Thus, *internalname* appears in the source code of the importing program, while the function may have a different name in the program where it is defined. By default, *internalname* is the same as the *entry* name. An *internalname* is required if *entry* is an ordinal value.

The *modulename* is the filename of the exporting application or DLL that contains the function or data item.

The *entry* field specifies the name or ordinal value of the function or data item as defined in the *modulename* application or DLL. If *entry* is an ordinal value, *internalname* must be specified. (Ordinal values are set in an **EXPORTS** statement.)

NOTE A given symbol (function or data item) has a name for each of three different contexts. The symbol has a name used by the exporting program (application or DLL) where it is defined, a name used as an entry point between programs, and a name used by the importing program where the symbol is used. If neither program uses the optional *internalname* field, the symbol has the same name in all three contexts. If either of the programs uses the *internalname* field, the symbol may have more than one distinct name.

Example

The following **IMPORTS** statement defines three functions to be imported: `SampleRead`, `SampleWrite`, and a function that has been assigned an ordinal value of 1. The functions are found in the `Sample`, `SampleA`, and `Read` applications or DLLs, respectively. The function from `Read` is referred to as `ReadChar` in the importing application or DLL. The original name of the function, as it is defined in `Read`, may or may not be known and is not included in the **IMPORTS** statement.

```
IMPORTS
                Sample.SampleRead
                SampleA.SampleWrite
    ReadChar = Read.1
```

13.19 Related Topics in Online Help

In addition to information covered in this chapter, information on the following topics can be found in online help.

Topic	Access
Syntax and procedural information on LIB	Choose "LIB" from the list of utilities on the "Microsoft Advisor Contents" screen
Module-definition files and IMPLIB	Choose "LINK" from the list of utilities on the "Microsoft Advisor Contents" screen

Chapter 14

Customizing the Microsoft Programmer's WorkBench

The Microsoft Programmer's WorkBench (PWB) is not just a text editor, but also a full-featured platform for program development. It is both flexible (you can customize it to match your working habits) and extensible (you can add your own functions and features).

This chapter explains three ways to customize the Programmer's WorkBench:

- Setting switches
- Assigning keystrokes
- Writing macros

While this chapter explains customizing techniques, it does not document every customizable feature. Please consult online help for detailed information about these and other PWB features.

This chapter assumes you are familiar with basic PWB operation and terminology. If not, please read "Using the Programmer's WorkBench" in *Installing and Using the Microsoft Macro Assembler Professional Development System.* The Programmer's WorkBench is supplied with both the Macro Assembler and Microsoft C so that you can customize one copy of PWB to work with these and other languages.

14.1 Setting Switches

The Programmer's WorkBench has a number of "switches," or user-configurable options, that control features such as how many lines the screen scrolls or whether you are prompted to save a file when you exit. Each switch has a name and can be assigned a value.

There are two ways to set PWB switches. The easiest way is to choose Editor Settings from the Options menu. Saving the changes made to Editor Settings updates your TOOLS.INI initialization file. You can also directly edit TOOLS.INI. Either method can be used for more elaborate customizations, such as writing macros.

14.1.1 Changing Current Assignments and Switch Settings

You can change the current editor switches and key assignments. Choose Editor Settings or Key Assignments from the Options menu. PWB displays these settings in a new window labeled Current Assignments and Switch Settings.

The <ASSIGN> pseudofile is associated with the Current Assignments and Switch Settings window. A pseudofile exists only in memory; it has no counterpart on disk until you explicitly save it. Saving the <ASSIGN> pseudofile automatically saves any changes you make in the Current Assignments and Switch Settings window.

To change a switch, edit the line on which it appears. For instance, the **vscroll** switch controls how many lines PWB scrolls vertically; its default setting is 1. To change it, move to the corresponding line:

```
vscroll:1
```

Change the 1 to 3 and move the cursor to another line. PWB highlights the line to indicate that the change has been executed. (If you make an illegal change, PWB signals an error.) The change takes effect immediately: PWB now scrolls text three lines at a time.

When you save <ASSIGN>, PWB updates your TOOLS.INI file.

PWB discards all changes at the end of a session unless you explicitly save them. You save changes by saving <ASSIGN> as you would any other file. Select Save from the File menu, or press SHIFT+F2.

You can also use this method for more elaborate customizations, such as writing macros (see Section 14.3, "Writing Macros"). Simply insert a few blank lines in the Current Assignments and Switch Settings window and enter the new information in them.

If you add or modify a line of the Current Assignments and Switch Settings window, PWB immediately alters its behavior accordingly; the new or changed lines are saved in TOOLS.INI when you save the <ASSIGN> file. However, deleting a line has no effect, either on PWB's behavior or the contents of TOOLS.INI; you must edit TOOLS.INI to remove an assignment.

14.1.2 Editing the TOOLS.INI Initialization File

Another way to customize PWB is by editing TOOLS.INI, the initialization file used by PWB and other Microsoft language utilities. This is the most convenient way to perform extensive customizing.

While the Current Assignments and Switch Settings window displays every customizable PWB item, the TOOLS.INI file contains lines only for items you have customized. PWB sets any items you omit from TOOLS.INI to a default value.

TOOLS.INI is made up of sections that start with tags.

Since TOOLS.INI can initialize a number of Microsoft tools, the file is divided into sections, one for each tool. Each section begins with a tag consisting of the tool's base name enclosed in square brackets: `[PWB]` for PWB.EXE, `[NMAKE]` for NMAKE.EXE, and so on.

For example, assume you set the **vscroll** switch to 3 and saved the change, but you have not customized PWB in any other way. Your TOOLS.INI file will contain this section:

```
[PWB]
vscroll:3
```

PWB reads TOOLS.INI at start-up and loads the settings from the `[PWB]` section.

You can also create sections of TOOLS.INI that configure PWB for specific programming languages or operating systems. For instance, your TOOLS.INI file could contain a section beginning with the tag

```
[PWB-.C]
```

for C source files, and

```
[PWB-.ASM]
```

TOOLS.INI sections contain customization information.

for assembly-language (.ASM) source files. Each time you load a file with the designated extension, PWB reads the appropriate section of TOOLS.INI. You can have a different set of macros and other customizations for each file type.

TOOLS.INI can also contain sections specific to an operating system. The following tag introduces a section specific to DOS version 3.31, for instance:

```
[PWB-3.31]
```

You can combine tags as needed. For example, the tag

```
[PWB-3.0 PWB-10.10R]
```

applies to DOS version 3.0 and OS/2 version 1.1 real mode.

14.2 Assigning Functions to Keystrokes

You can assign any PWB function to almost any keystroke. Keystroke assignments, like switches, are displayed in the Current Assignments and Switch Settings window (choose Key Assignments from the Options menu) and can be

changed there. Suppose you want to assign the **home** cursor function to SHIFT+HOME. The default keystroke assignment for **home** is

```
home:Goto
```

If you change the assignment to

```
home:Shift+Home
```

SHIFT+HOME moves the cursor to the home (upper left) window position.

You can assign the same function to more than one keystroke. For example, many keystrokes invoke the **select** function, which selects a text region. The preceding example adds a new keystroke (SHIFT+HOME) for the **home** function, but it does not remove the previous assignment (GOTO, the 5 key on the keypad).

If you aren't sure whether a keystroke is already assigned, select the Current Assignments and Switch Settings window and press PGDN until you reach the Available Keys table. All unassigned keystrokes are displayed; once a keystroke is assigned, it no longer appears in this table.

There are two limitations on keystroke assignments:

- You should not reassign a keystroke that PWB assigns to a menu. For instance, ALT+F displays the File menu; PWB ignores any attempt to reassign ALT+F.
- You should not reassign the ALT plus number keys 1– 6 (ALT+1, ALT+2, and so on). These keystrokes are reserved for the file history menu items.

PWB uses the most recent duplicate key assignment.

A keystroke can invoke only one function. If you accidentally assign a keystroke to more than one function, PWB uses the most recent assignment. For example,

```
home:Ctrl+A
setfile:Ctrl+A
```

assigns the CTRL+A keystroke to two different functions, **home** and **setfile**. The second assignment overrides the first, assigning CTRL+A to **setfile**.

You might occasionally want to "unassign," or disable, a keystroke. This is done by assigning the **unassigned** function to the keystroke. For example,

```
unassigned:Ctrl+A
```

disables CTRL+A. PWB signals an error when you press any unassigned key.

As the list of assigned keystrokes shows, you can use SHIFT+CTRL as a prefix. For PWB to recognize this key combination, SHIFT must come first. For example, to use SHIFT+CTRL with M, you must type SHIFT+CTRL+M, not CTRL+SHIFT+M.

14.3 Writing Macros

If you need a feature or function that is not a part of PWB, the quickest way to create it is by writing a macro in the TOOLS.INI file. A macro can do something as simple as inserting a line of text, or it can perform complex operations by invoking PWB functions and other macros.

14.3.1 Macro Syntax

A macro can consist of any combination of PWB functions, literal text, and calls to previously defined macros. You can define up to 1,024 macros at one time.

Anything inside quotation marks is literal text. Within literal text, quotation marks are represented by a backslash followed by quotation marks (\") and a backslash is represented by two consecutive backslashes (\\). Only literal text is case sensitive; PWB ignores the case of everything else.

The following macro "comments out" a line of MASM source code:

```
comment:=begline "; "
comment:alt+c
```

The first line names the macro (`comment`); the macro commands follow the assignment operator (**:=**). The **begline** editor function moves the cursor to the beginning of the current line. The text inside quotation marks (the MASM comment delimiter) is then inserted. The second line assigns a keystroke (ALT+C) to the macro.

Macros can extend over one line.

If a macro definition takes up more space than you have on one line (about 250 characters in PWB), you can use the backslash (\ ) to continue the definition on the next line. Consider, for instance, the following macro, which comments out a line of C source code:

```
comment:=begline "/* " endline " */"
```

It could be written as

```
comment:=begline  \
"/* " endline  \
" */"
```

Notice the extra space before each backslash. If you want a space between the end of one line and the beginning of the next, you must precede the backslash with two spaces.

You can pass arguments to PWB macros.

You can use the **arg** function to pass arguments to functions. For example, the following macro passes the argument `15` to the **plines** function (which scrolls text down):

```
movedown:=arg "15" plines
```

Because **arg** precedes the literal text, the text isn't written to the screen. Instead, it is passed as an argument to the next function, **plines**. The macro scrolls the current text down 15 lines.

Arguments can also use regular-expression syntax (regular expressions are documented in online help):

```
endword:=arg arg "([ .,;:()[\\]]!$)" psearch cancel
```

The **arg arg** sequence directs the **psearch** function to treat the text argument as a regular-expression search pattern. This search pattern tells PWB to search for the next space, period, comma, semicolon, colon, parentheses, and square brackets. (Note that a backslash must precede any character that has a special meaning in regular expressions—in this case, the right bracket.)

A macro can invoke other macros:

```
lcomment:= "/* "
rcomment:= " */"
commentout:=begline lcomment endline rcomment
commentout:ctrl+o
```

The `commentout` macro invokes the previously defined macros `lcomment` and `rcomment`.

In addition to standard PWB functions, PWB macros can invoke user-defined macro functions. See Section 9.6, "Returning Values with Macro Functions."

14.3.2 Macro Responses

Some PWB functions ask you for confirmation. For example, the **meta exit** (quit without saving) function normally asks if you really want to exit. Such questions always take the answer "yes" (Y) or "no" (N).

When you invoke such a function in a macro, the function assumes an answer of yes and does not ask for confirmation. For example, the macro definition

```
quit:=meta exit
quit:alt+x
```

The meta prefix modifies the action of a function.

invokes **meta exit** when you press ALT+X. Because the **meta exit** function is invoked from a macro, PWB exits without asking for confirmation.

The following operators allow you to restore normal prompting or change the default responses:

Operator	Description
<	Asks for confirmation; if not followed by another < operator, prompts for all further questions
<y	Assumes a response of "yes"
<n	Assumes a response of "no"

A response operator applies to the function immediately preceding it. For example, you can add the < operator to the `quit` macro definition to restore the usual prompt:

```
quit:=meta exit <
quit:alt+x
```

Now the macro prompts for a response before it exits.

14.3.3 Macro Arguments

If you enter an argument in PWB and then invoke a macro, the argument is passed to the first function in the macro that takes an argument:

```
tripleit:=copy paste paste
```

The `tripleit` macro invokes the **copy** and **paste** editing functions. When you highlight a text area and then invoke the macro, your highlighted argument is passed to the **copy** function, which copies the argument to the clipboard. The macro then invokes **paste** twice. The effect is to insert two copies of the highlighted text.

You cannot pass more than one argument from PWB to a macro.

You cannot pass more than one argument from PWB to a macro, even if the macro invokes more than one function that can accept an argument. The argument always goes to the first function in the macro that takes an argument.

You can also prompt for input inside a macro and pass the input as an argument using the **prompt** function as shown below:

```
newfile:=arg "Next file: " prompt setfile <
newfile:alt+n
```

The `newfile` macro prompts for a file name and then switches to the specified file. The sequence `arg "Next file: "` passes the text argument `Next file:` to **prompt**, which prints it in the text-argument dialog box and waits for the user to respond. The response is passed as a text argument to the **setfile** function, which switches to that file.

14.3.4 Macro Conditionals

Macros can take different actions depending on certain conditions. Such macros take advantage of the fact that PWB editing functions return values—a TRUE (nonzero) value if successful or FALSE (zero) if unsuccessful.

Macros can use four conditional operators:

Operator	Description
:>*label*	Defines a label that can be targeted by other operators
=>*label*	Jumps to *label*
+>*label*	Jumps to *label* if the previous function returns TRUE
–>*label*	Jumps to label if the previous function returns FALSE

For example, the `leftmarg` macro moves the cursor to the left margin of the editing window:

```
leftmarg:=:>leftmore left +>leftmore
```

The macro above invokes the **left** function repeatedly (jumping to the label `leftmore`) until it returns FALSE, indicating the cursor has reached the left margin.

Macro execution depends on the status of conditionals.

The label must appear immediately after the conditional operator, with no intervening spaces. A conditional operator without a label exits the macro immediately if the condition is satisfied. If the condition is not satisfied, the macro continues execution. The following example demonstrates this:

```
turnon:=insertmode +> insertmode
```

This macro turns on insert mode regardless of whether insert mode is currently on or off. If insert mode is off, the first invocation of **insertmode** toggles the mode on and returns TRUE, causing the +> operator to terminate the macro. If insert mode is currently on, the first invocation of **insertmode** turns insert mode off and returns FALSE. The macro then invokes **insertmode** a second time, turning insert mode back on.

14.3.5 Recording Macros

You can also create a macro by recording a procedure as you perform it. The keystroke sequence is saved and can be replayed, like any other macro. To record a macro:

1. Choose Set Record from the Edit menu. The Set Macro Record dialog box appears.

2. Type the name you want the macro to have in the Name text box.
3. Tab to the Key Assignment text box and press the key to which you are assigning the macro. (For example, press ALT+T to assign the macro to ALT+T. The name of the keystroke appears in the text box.) If the keystroke (such as ENTER, TAB, or ESC) would normally exit the dialog box or move to the next field, type in the keystroke's name.
4. Click the OK button.
5. Choose Record On from the Edit menu to start the recording.
6. Type the text or perform the actions you want to record. (You can select text or fields with the mouse as well as the keyboard. Mouse selections are automatically converted into equivalent keystrokes.)
7. Choose Record On again to end the recording.

You have now created a named macro available through the assigned keystroke. Pressing this key replays the actions you recorded.

WARNING If you do not select a name for your macro, it is assigned the default name **recordvalue**. Unless you plan to discard the macro when exiting, do not let a recorded macro's name default to **recordvalue**. Any subsequent macro recorded with the **recordvalue** default name will overwrite the first **recordvalue** macro.

A recorded macro is temporary; PWB discards it when you exit. To save a recorded macro:

1. Choose Edit Macro from the Edit menu. This opens the <RECORD> pseudofile and displays the macros you recorded.
2. Make any changes required. For example, you might want to change the macro's name or modify the keystroke sequence.
3. Save the macro using the Save command from the File menu.

The macros defined in the <RECORD> pseudofile are added to your TOOLS.INI file when you save the <RECORD> file. PWB automatically reloads them at the next session.

You can append functions to an existing macro without having to record the original steps again:

1. Choose Set Record from the Edit menu. The Set Macro Record dialog box appears.
2. Type the macro's name in the Name text box.

3. Tab to the Clear First check box and cancel selection. This causes any new actions to be appended to the original macro, rather than replacing (clearing) it.
4. Click the OK button.
5. Choose Record On from the Edit menu to start the recording.
6. Perform the actions you want added to the macro.
7. Choose Record On again to end the recording.

Remember to save the modified macro before exiting, or the new version will be discarded.

You can record a series of actions without executing them.

You can make a "silent" recording, which records a series of actions without executing them. This allows you to create a macro without altering or damaging the file. Start the recording with a **meta record** command (press F9, SHIFT+CTRL+R). When the macro is complete, terminate recording with **record** (press SHIFT+CTRL+R).

PWB gives no visual feedback during silent recording. If you need to see the macro being created, open the <RECORD> pseudofile in a second window as described above. This is an excellent way to get a better understanding of macros and editor functions.

14.3.6 Temporary Macros

You can use the **assign** function to create a macro that lasts only until the end of the current session. For example, the following steps create the `comment` macro described above:

- Press ALT+A
- Type `comment:=begline "; "`
- Press ALT+=

This key sequence tells PWB to open dialog boxes where the macro and key assignments are to be typed. To assign ALT+C to the macro,

- Press ALT+A
- Type `comment:alt+c`
- Press ALT+=

The macro is available immediately and is discarded when you exit PWB.

14.4 Related Topics in Online Help

Information on the following related topics can be found in online help. All the topics listed below are found by choosing "Programmer's WorkBench" from the "Microsoft Advisor's Help System Contents" screen.

Topic	Access
Writing macros	Choose "Writing and Using Macros"
TOOLS.INI	Choose "Using TOOLS.INI"
Regular expressions	Choose "Writing and Using Macros;" then choose "Regular Expressions" from under the "Building Macros" subhead
The **prompt** and **meta** functions	Choose "Using PWB Functions," and from the next screen, choose "Alphabetical List"
Assigning keystrokes	Choose "Setting PWB Switches" and then "Assign Function"

Chapter 15

Debugging Assembly-Language Programs with CodeView

You can diagnose software problems and locate programming errors quickly with the CodeView debugger. This chapter explains how to

- Display and modify variables and memory
- Control the flow of execution
- Use advanced CodeView debugging techniques
- Modify CodeView's behavior with command-line switches and the TOOLS.INI file

CodeView supports the Microsoft mouse (or any fully compatible pointing device). This chapter first describes CodeView operations with the mouse, then with function keys. Command-window commands are not generally discussed, except when there is no comparable mouse or function-key command. Unless a specific mouse button is named, "clicking" means pressing and quickly releasing the left mouse button.

15.1 Understanding Windows in CodeView

CodeView divides the screen into logically separate sections called windows. Windows permit a large amount of information to be displayed in an organized and easy-to-read fashion.

Each window displays a different type of data.

Each CodeView window has a distinct function and operates independently of the others. The name of each window described below appears in the top of the window's frame:

- The Source window displays the source code. You can open a second source window to view an include file, another source file, or the same source file at a different location. Any ASCII text file can be viewed in the Source window.
- The Command window accepts debugging commands from the keyboard.
- The Watch window displays the current values of selected variables.

- The Local window lists the values of all variables local to the current procedure.
- The Memory window shows the contents of memory. You can open a second Memory window to view a different section of memory.
- The Register window displays the contents of the microprocessor's registers, as well as the processor flags.
- The 8087 window displays the registers of the coprocessor or its software emulator.

Figure 15.1 shows all CodeView windows.

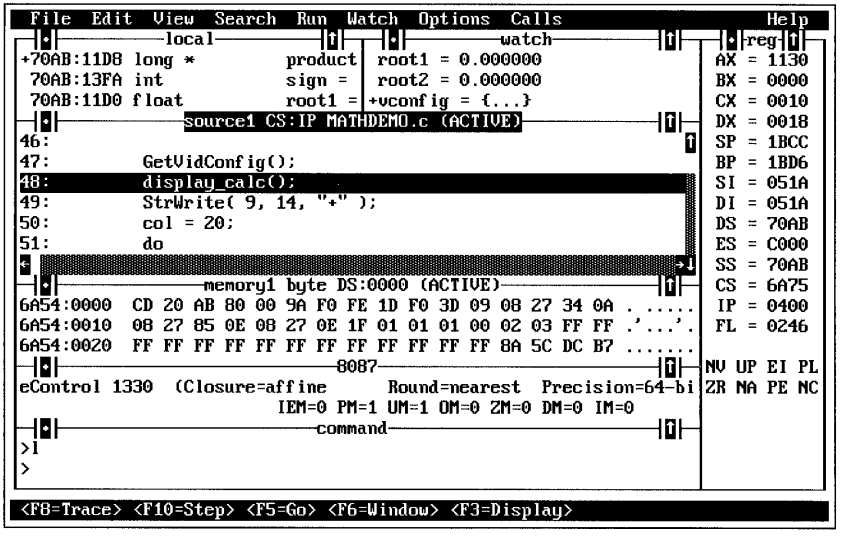


Figure 15.1 CodeView Display of All Possible Windows

The first time you run CodeView, it displays three windows. The Local window is at the top, the Source window fills the middle of the screen, and the Command window is at the bottom. CodeView records which windows were open and how they were positioned at the time you exit. These settings become the default the next time you run CodeView.

There are two ways to open windows. You can choose the desired window from the View menu or press its shortcut key. In addition, some operations (such as selecting a Watch variable) automatically open the appropriate window if it isn't already open.

All displays are updated automatically.

CodeView continually and automatically updates the contents of all windows. However, if you want to interact with a particular window (such as entering a

command, setting a breakpoint, or modifying a variable), you must first select that window.

The selected window is called the "active" window. The active window is marked in three ways:

- The window's name is highlighted.
- The text cursor appears in the window.
- The vertical and horizontal scroll bars move into the window.

Figure 15.2 shows the Source window as the active window.

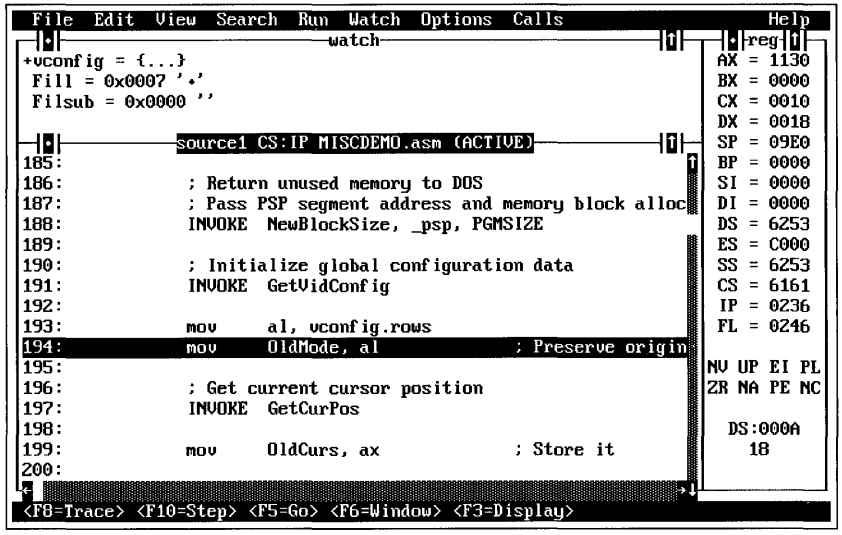


Figure 15.2 Source Window as Active Window

To select a new active window, click that window (position the mouse pointer in the window and press the left mouse button). You can also press F6 or SHIFT+F6 to move from one window to the next.

Windows often contain more information than can be displayed in the area allotted to the window. There are several ways to view these additional contents.

To view additional contents with the mouse:

- Drag the scroll box on the horizontal or vertical scroll bars. (Position the mouse pointer on the scroll box and, while holding down the left mouse button, move the mouse in the appropriate direction.)
- Click the arrows at the top and bottom of the scroll bars.
- Click the gray area to either side of the scroll box in a scroll bar.

To view additional contents with the keyboard:

- Press the direction keys (LEFT, RIGHT, UP, DOWN) to move the cursor.
- Press PGUP, PGDN, CTRL+PGUP (page left), and CTRL+PGDN (page right) to move the cursor to a different page of the window's contents.
- Press CTRL+HOME to move the cursor to the beginning of the window's contents.
- Press CTRL+END to move the cursor to the end of the window's contents.

Typing commands when the Source window is active causes CodeView to temporarily shift its focus to the Command window. Whatever you type is appended to the last line in the Command window. If the Command window is closed, CodeView beeps in response to your entry and ignores the input.

Adjusting the Windows

Although you can't reposition the windows, you can change their size or close them. The Maximize, Size, and Close commands from the View menu perform these functions, or you can press CTRL+F10, CTRL+F8, and CTRL+F4, respectively. Window manipulation is especially easy with a mouse:

- To maximize a window (enlarge it so it fills the screen), click the up arrow at the right end of the window's top border, or double-click the window's title. (Position the mouse pointer anywhere on the title and press the left mouse button twice, rapidly.) To restore the window to its original size, click the double arrow at the right end of the top border or press CTRL+F10.
- To change the size of a window, position the mouse pointer anywhere along the line at the top of the window. Press and hold down the left mouse button, then drag the mouse to enlarge or reduce the window. The same action on a vertical border widens or narrows the window.
- To close a window, click the dot at the left end of the top border. The adjacent windows automatically expand to recover the unused space. You can also close any window whose View menu name has a dot next to it: choose that window from the menu or press the window's shortcut key.

CodeView remembers the last debugging session.

CodeView stores session information in a file called CURRENT.STS, which is created in the directory pointed to by the INIT environment variable (or in the current directory, if there is no INIT variable). The session information includes such items as the name of the program being debugged, the CodeView windows that were open, breakpoint locations, and other status. This information becomes the default status the next time you run CodeView.

15.2 Overview of Debugging Techniques

There is no single best approach to debugging. CodeView offers a variety of debugging tools that let you select a method appropriate for the program or for your work habits. This section describes some approaches to solving debugging problems.

Broadly speaking, two things can go wrong in a program:

- The program doesn't manipulate the data the way you expected it to.
- The flow of execution is incorrect.

These problems usually overlap. Incorrect execution can corrupt the data, and bad data can cause execution to take an unexpected turn. Because CodeView allows you to trace program execution while simultaneously displaying whatever combination of variables you want, you don't have to know ahead of time whether the problem is bad data manipulation, a bad execution path, or some combination of both.

CodeView has specific features that deal with the problems of bad data and incorrect execution:

- You can view and modify any program variable, any section of memory, or any processor register. These features are explained in Section 15.3, "Viewing and Modifying Program Data."
- You can monitor the path of execution and precisely control where execution pauses. These features are explained in Section 15.4, "Controlling Execution."

15.3 Viewing and Modifying Program Data

CodeView offers a variety of ways to display the values of program variables, processor registers, and memory. You can also modify the values of all these items as the program executes. This section shows how to display and modify variables, registers, and memory.

15.3.1 Displaying Variables in the Watch Window

To add a variable to the Watch window, position the cursor on the variable's name, using the mouse or the direction keys (LEFT, RIGHT, UP, DOWN). Then choose the Add Watch command from the Watch menu, or press CTRL+W.

A dialog box appears with the selected variable's name displayed in the Expression field. If you don't want to watch the variable shown, type in the name of another variable. Click the OK button or press ENTER to add this variable to the Watch window.

The Watch window appears at the top of the screen. Selecting a Watch variable automatically opens the Watch window if the window isn't already open.

A newly added variable may be followed by the message:

```
<Watch Expression Not in Context>
```

This message appears when execution has not yet reached the procedure where a local variable is defined. Global variables (those declared outside procedures) never cause CodeView to display this message; they can be watched from anywhere in the program.

To remove a variable from the Watch window, choose the Delete Watch command from the Watch menu or press CTRL+U. Then select the variable to be removed from the list in the dialog box. You can also position the cursor on any line in the Watch window and press CTRL+Y to delete that line.

You can watch an unlimited number of variables.

You can place as many variables as you like in the Watch window; the quantity is limited only by available memory. You can scroll the Watch window to position it at those variables you want to view. CodeView automatically updates all Watch window variables as the program runs, including those not currently visible within the Watch window frame.

A variable can be specified by its address as well as its name. You can give its address in *segment*:*offset* form, where either component can be a register name or a number. You can extract a variable's address by prefixing the **&** operator to its name. Prefixing a variable's address (or any address) with the **BY**, **WO**, or **DW** operator displays the byte, word, or doubleword value starting at that address.

There are several ways to display a variable's value.

By default, CodeView displays variables as decimal values. You can select the radix by typing `n8`, `n10`, or `n16` in the Command window for an octal, decimal, or hexadecimal display. CodeView remembers the current radix when you exit; it becomes the default radix the next time you run CodeView.

15.3.2 Displaying Expressions in the Watch Window

The Watch window is not limited to variables. You can enter an expression (that is, any valid combination of variables, constants, and operators) for CodeView to evaluate and display. You can also select the format in which CodeView displays the expression.

MASM expressions are evaluated using C rules.

CodeView does not include an expression evaluator specifically for MASM. It uses the C expression evaluator instead. This means you must enter MASM variables or expressions in a form the C evaluator recognizes, which is not always the way they appear in a MASM program. (Online help describes the operators and precedence order for C expressions. The last part of this section also gives examples of some of the more commonly used expression forms.)

The Language command from the Options menu offers a choice of Auto, C, Basic, or FORTRAN expression evaluators. However, the Basic and FORTRAN expression evaluators do not support address evaluation, pointer conversions, type casting, or other operations needed when debugging assembly-language code.

Besides arithmetic and memory-reference expressions, CodeView can also display Boolean expressions. For example, if a variable is never supposed to be larger than 100 or less than 25, the expression

```
(var < 25 || var > 100)
```

evaluates to one (TRUE) if `var` goes out of bounds.

Changing Display Format

By default, CodeView displays expression values in decimal form. You can change the display radix to octal or hexadecimal with the Radix (**N**) command described at the end of the previous section.

Another way to change the display format is to append a comma and a single-digit format specifier to any watched variable, expression, or address. For example, to display `varname` in octal form, type `varname,o` in the Watch expression box. (If `varname` is already in the Watch window, simply append a comma and the octal specifier `,o` and then move the cursor off the line.) The following list describes the use of each specifier:

Specifier	Form Displayed
c	Least-significant byte of the variable displayed as a single character
d	Decimal value
e or E	Eight bytes displayed as a double-precision exponential number
f	Four bytes displayed as a single-precision floating-point number
g or G	Eight bytes displayed as a double-precision exponential number
i	Signed integer value
o	Unsigned octal value
s	String; all following bytes displayed as ASCII characters, up to next null character (ASCII 0)
u	Unsigned decimal value
x or X	Hexadecimal value, without leading `0x`

Displaying MASM Expressions

Expressions using registers or indexes are more complex. The following sections show how to substitute CodeView expressions using the C expression evaluator for MASM expressions.

Register Indirection The C expression evaluator does not recognize brackets to indicate the memory location pointed to by a register. Instead, use the **BY**, **WO**, or **DW** operator to reference the corresponding byte, word, or doubleword value.

MASM Expression	CodeView Equivalent
`BYTE PTR [bx]`	`BY bx`
`WORD PTR [bp]`	`WO bp`
`DWORD PTR [bp]`	`DW bp`

Register Indirection with Displacement To perform based, indexed, or based-indexed indirection with a displacement, use the **BY**, **WO**, or **DW** operator combined with addition.

MASM Expression	CodeView Equivalent
`BYTE PTR [di+6]`	`BY di+6`
`BYTE PTR Test [bx]`	`BY &Test+bx`
`WORD PTR [si] [bp+6]`	`WO si+bp+6`
`DWORD PTR [bx] [si]`	`DW bx+si`

Address of a Variable Use the address operator (**&**) instead of the **OFFSET** operator.

MASM Expression	CodeView Equivalent
`OFFSET Var`	`&Var`

PTR Operator Use C type casts, or the **BY**, **WO**, and **DW** operators in conjunction with the address operator (**&**), to replace the **PTR** operator.

MASM Expression	CodeView Equivalents
`BYTE PTR Var`	`BY &Var` `*(unsigned char*)&Var`
`WORD PTR Var`	`WO &Var` `*(unsigned *)&Var`
`DWORD PTR Var`	`DW &Var` `*(unsigned long*)&Var`

Strings Add a comma and the string specifier `,s` after the variable name.

MASM Expression	CodeView Equivalent
`Stringvar`	`Stringvar,s`

Because CodeView uses the C expression evaluator and C strings end with an ASCII null (zero), CodeView displays all characters up to the next null in memory when you request a string display. If you intend to debug a MASM program, you should terminate string variables with a null.

Array and Structure Elements The C expression evaluator equates an array name with the address of its first element. Therefore, you should prefix an array name with the address operator (&), then add the desired offset. The offset can be added directly, or it can appear within parentheses. It can be a number, a register name, or a variable.

The following examples (using byte, word, and doubleword arrays) show how this is done:

MASM Expression	CodeView Equivalents
`String[12]`	`BY &String+12` `*(&String+12)`
`aWords[bx+di]`	`WO &aWords+bx+di` `*(unsigned*)(&aWords+bx+di)`
`aDWords[bx+4]`	`DW &aDWords+bx+4` `*(unsigned long*)(&aDWords+bx+4)`

Pointers MASM 6.0 lets you define pointer-type variables. Since these are the same as C pointers, the C expression evaluator works as it does with C programs.

You dereference a pointer simply by typing its name in the Watch window. The pointer's address is displayed, followed by all the elements of the variable to which the pointer refers. Multiple levels of indirection (that is, pointers referencing other pointers) can be displayed simultaneously.

15.3.3 Displaying Local Variables

When your program is executing within the scope of a procedure, the Local window automatically displays the variables local to that procedure (stack variables). This includes arguments declared in **PROC** directives and variables explicitly declared as **LOCAL** within the procedure.

Note that variables you create on the stack are not displayed in the Local window, since CodeView is aware only of the assembler-created stack. You can display user-defined stack variables in the Watch window by specifying their address in *segment*:*offset* form.

15.3.4 Using Pointers to Display Arrays and Strings

Unlike high-level-language compilers, MASM does not provide symbolic information for arrays. Consequently, CodeView cannot distinguish between a simple variable and an array, and therefore cannot directly display an assembly-language array in expanded form. (See Section 15.3.2, "Displaying Expressions in the Watch Window," to display individual array elements.)

A user-defined pointer lets you view an expanded array.

For debugging purposes, you can overcome MASM's lack of array information by using the **TYPEDEF** directive to define a pointer type, and from that a pointer variable for the array. (Place the directive and pointer definition within a conditional-assembly block, so the pointer won't be added to your release code.) You can then view the array from CodeView by placing the pointer in the Watch window. For example:

```
array   BYTE    20 DUP (0)      ; array of 20 bytes

IF debug
   PBYTE   TYPEDEF PTR BYTE     ; PBYTE type is pointer to bytes
   parray PBYTE array           ; parray points to array
ENDIF
```

If you declare multiple levels of pointers (pointers to pointers to pointers, and so on), multiple levels of indirection can be displayed simultaneously by expanding each subpointer.

If it is inconvenient to view a character array in hexadecimal form, cast the variable's name to a character pointer by placing `(char *)` in front of the name. The character array is then displayed as a string delimited by apostrophes. You can also append the string-format specifier `,s` to the expression.

Note that the C expression evaluator expects a string to terminate with the ASCII null character (0). If you do not include a terminating null in the string's definition, the evaluator continues displaying memory as characters until it encounters a null. The Memory window is an effective way to view nonterminated strings.

15.3.5 Displaying Structures

MASM adds structure and union information to the debugging table. You can display MASM structures in expanded form, just as you would in C, Basic, Pascal, or FORTRAN.

Structures contain multiple data values, often of different data types, arranged in one or more layers. Therefore, they are often referred to as "aggregate" data items. CodeView lets you control how much of a structure is shown; that is, whether all, part, or none of its components are displayed.

The following example defines a structure and pointer types to implement a simple linked list:

```
PTRLINKEDLIST TYPEDEF PTR LINKEDLIST
PTRDATAWORD   TYPEDEF PTR WORD

LINKEDLIST STRUCT
   ptrNext PTRLINKEDLIST 0
   ptrData PTRDATAWORD   0
LINKEDLIST ENDS

rootNode linkedList < >
```

Once `rootNode` has been defined, the program calls the **MALLOC** function (which is available from the libraries of Microsoft high-level languages) to allocate memory for a structure pointer and a data pointer. The addresses of each are assigned to the corresponding pointers in `rootNode`, readying the list for its first entry.

The program stores a list item at the memory location specified by the preceding pointer, then calls **MALLOC** to allocate memory for the next list item. This process is repeated for each new list item, creating a linked list of data structures.

To display the linked list of structures, add `rootNode` to the Watch window. It initially appears in the form:

```
+rootnode = {...}
```

The brackets indicate that this is an aggregate variable (since it's a structure). The plus sign (+) indicates that the structure has not yet been expanded to display its components.

To expand `rootnode`, double-click its display line. (Position the mouse pointer anywhere on the line and press the left mouse button twice, rapidly.) You can also move the cursor to the line and press ENTER. The Watch window display changes to

```
-rootnode
   +ptrnext  = 0F00:1111
   ptrdata   = 0x0032  "2"
```

The address and data values shown here are arbitrary. They depend on the data values stored and on the memory location from where **MALLOC** obtained free space. The minus sign (–) indicates that `rootnode` has been fully expanded; no further expansion is possible. The plus sign (+) indicates that `ptrnext` points to another structure that has not been expanded.

Any structure element can be independently expanded or contracted. To expand the next structure, double-click `ptrnext`, or press ENTER when the cursor is on that line. The Watch window display changes to

```
-rootnode
   -ptrnext  = 0F00:1111
      +ptrnext  = 0F00:2222
      ptrdata  = 0x0034  "4"
   ptrdata  = 0x0032  "2"
```

Note that both the data value and its ASCII equivalent are displayed. To contract the structure, double-click its line a second time or position the cursor on the line and press ENTER.

The process of expanding structures pointed to by `ptrnext` may be repeated indefinitely until you reach the last structure in the list. Its identifier will be prefixed with a minus sign, indicating that no more space for structures has been allocated.

You can view individual elements instead of the entire structure.

If you want to view only one or two elements of a large structure, indicate the specific structure elements in the Expression field of the Add Watch dialog box. Structure elements are separated by a dot (.), so you would type

```
rootnode.ptrnext.ptrnext
```

to view the pointer from the third structure in the list.

15.3.6 Using Quick Watch

Choose the Quick Watch command from the Watch menu (or press SHIFT+F9) to display the Quick Watch dialog box. If the cursor is in the Source, Local, or Watch window, the variable at the current cursor position appears in the dialog box. If it isn't the item you want to display, type in the desired expression or variable; then press ENTER. The Quick Watch window immediately displays the specified item.

The Quick Watch display automatically expands structures and pointers to their first level. You can expand or contract an element just as you would in the Watch window: position the cursor on the appropriate line and press ENTER. If the array needs more lines than the Quick Watch window can display, drag the scroll box with the mouse, or press DOWN or PGDN to view the rest of the array.

You can add Quick Watch variables to the Watch window.

Choose the Add Watch button to add a Quick Watch item to the Watch window. Structures and pointers appear in the Watch window expanded as they were displayed in the Quick Watch dialog box.

Quick Watch is a convenient way to take a quick look at a variable or expression. Since only one Quick Watch variable can be viewed at a time, you would not use Quick Watch for most of the variables you want to view.

15.3.7 Displaying Memory

Choosing the Memory command from the View menu opens a Memory window. Two Memory windows can be open at one time.

By default, memory is displayed as hexadecimal byte values, with 16 bytes per line. At the end of each line is a second display of the same memory in ASCII form. Values that correspond to printable ASCII characters (decimal 32 to 127) are displayed in that form. Values outside this range are shown as dots (.).

You can display memory values in any form.

Byte values are not always the most convenient way to view memory. If the area of memory you're examining contains character strings or floating-point values, you might prefer to view them in a directly readable form. Choosing the Memory Window command from the Options menu displays a dialog box with a variety of display options:

- ASCII characters
- Byte, word, or doubleword binary values
- Signed or unsigned integer decimal values
- Short (32-bit), long (64-bit), or ten-byte (80-bit) floating-point values

Figures 15.3 and 15.4 show two of these different displays.

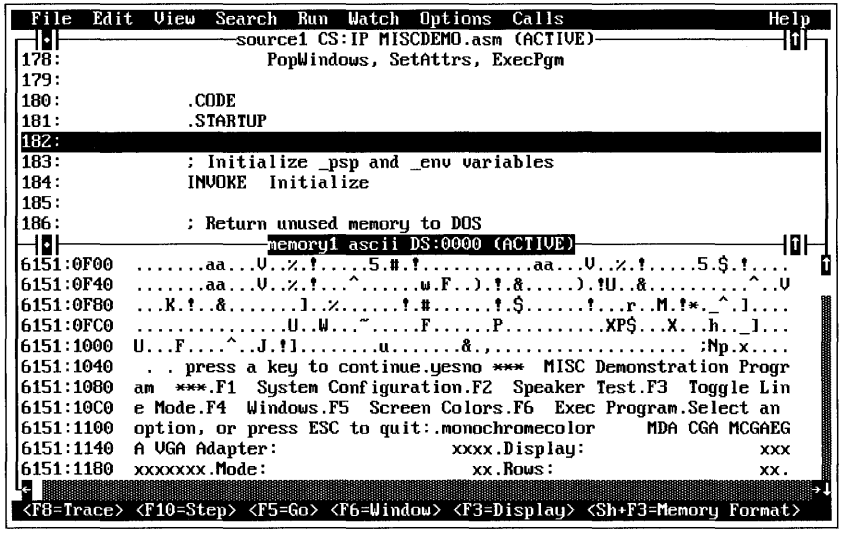


Figure 15.3 Memory Displayed in ASCII Characters

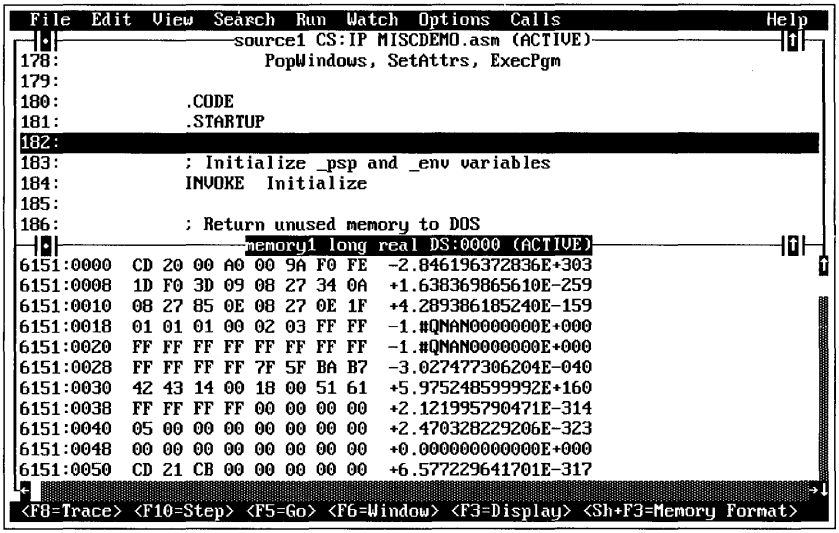


Figure 15.4 Memory Displayed in Long-Real Floating-Point Values

Another way to choose a display format is to cycle through the formats by repeatedly pressing SHIFT+F3.

Not every four-byte or eight-byte sequence represents a valid floating-point number. If a section of memory cannot be displayed in the floating-point format you select, the number displayed includes the characters NAN—"not a number."

You can change the contents of the memory by simply overtyping new values in the Memory window. See Section 15.3.9 for more information on modifying values.

Displaying Variables with a Live Expression

Section 15.3.4 explained how to display a specific array element by adding the appropriate expression to the Watch window. You can also watch a particular array element or structure element in the Memory window. This CodeView display feature is called a "live expression." The term "live" means that CodeView dynamically displays memory starting at the current value of the address expression you specify.

To create a live expression, choose the Memory Window command from the Options menu; then select the Live Expression check box. Type the element you want to view in the Address Expression field. For example, if `array` is a variable whose current value is being indexed by the value in the BI register and you wish to view it, type `array [bi]`. Then choose the OK button or press ENTER.

If no memory windows are open, a new Memory window opens. The first memory location in the window is the first memory location of the live expression. The section of memory displayed changes to the section the live expression currently references.

You can use the Memory Window command from the Options menu to display the memory in a directly readable form. This is especially convenient when the live expression represents strings or floating-point values, which are difficult to interpret in hexadecimal form.

It is usually more convenient to view an item in the Watch window than as a live expression. However, some items are more easily viewed as live expressions. For example, you can examine what is currently on top of the stack by entering SS:SP as the live expression. In fact, any legal combination of register values (such as ES:DI or DS:SI) can be entered in *segment*:*offset* form.

15.3.8 Displaying the Processor Registers

Choosing the Register command from the View menu (or pressing F2) opens a window on the right side of the screen. The microprocessor's current register values appear in this window. At the bottom of the window is a group of mnemonics representing the processor flags. Pressing F2 a second time closes the window.

Video intensity shows changed values.

When you first open the Register window, all register and flag values are shown in normal text. When you change a register or flag, the changed value is highlighted. For example, suppose the overflow flag is not set when the Register window is first opened. The corresponding mnemonic is NV and appears in light gray. If the overflow flag is subsequently set, the mnemonic changes to OV and appears in bright white. If your computer uses an 80386/486 processor and you are running the real-mode version of CodeView choosing the 386 Instructions command from the Options menu displays the registers as 32-bit values. Choosing this command a second time returns to the 16-bit display.

You can also display the registers of an 8087–80387 coprocessor (or the built-in coprocessor of the 80486) in a separate window by choosing the 8087 command from the View menu. If your program uses the coprocessor emulator, the emulated registers are displayed instead.

The Register values reveal program status.

The Register window is a valuable debugging tool. Almost every assembly instruction alters a register or flag. As each line of code is executed, the register values and flags that change are highlighted, so you can see whether each instruction does what you intended it to.

Also, when you execute an instruction whose operand has a memory location (such as a variable), the effective address of the operand, as well as the value stored at that address, is displayed at the bottom of the Register window.

15.3.9 Modifying the Values of Variables, Memory, and Registers

You can easily change the values of variables, memory locations, or registers displayed in the Watch, Local, Memory, Register, or 8087 windows. Simply position the cursor at the value you want to change and edit it to the appropriate value. In the Watch and Local windows, the change is accepted by CodeView when you move the cursor off the line. If you change your mind, press ALT+BKSP to undo the last change you made.

You can also alter expressions in the Watch window by adding an operator or changing the variable displayed. When you have altered the expression and moved the cursor off the line, CodeView will immediately show the new value of the modified expression.

The starting address of each line of memory displayed is shown at the left of the Memory window in *segment*:*offset* form. Altering the address automatically shifts the display to the corresponding section of memory. Under OS/2, if your program does not own that section of memory, memory values are displayed as double question marks (??).

It's easy to change memory values...

You can also change the values of memory locations by modifying the right side of the memory display (where memory values are shown in ASCII form). For example, to change a byte from decimal 75 to decimal 85, place the cursor over the letter K, which corresponds to the position where the memory value is 75 (K is ASCII 75), and type in U (ASCII 85).

...or flags.

To toggle a processor flag, double-click its mnemonic. You can also position the cursor on a mnemonic, then press any key (except ENTER, TAB, or SPACE). Press ALT+BKSP (undo) to restore the flag to its previous setting.

Be cautious when modifying memory or a register.

The effect of changing a register, flag, or memory location can vary from no effect at all to crashing the operating system. Be cautious when altering these values.

15.4 Controlling Execution

There are two forms of program execution under CodeView:

- Continuous; the program executes until either a previously specified breakpoint has been reached or the program terminates.
- Single-step; the program pauses after each line of code has been executed.

Sections 15.4.1 and 15.4.2 explain how each form of execution works and the most effective way to use each.

As you are debugging, you can display the program in source-code form or assembly form. Section 15.4.3 explains the advantages of each.

15.4.1 Continuous Execution

Continuous execution lets you quickly execute the bug-free sections of code which would otherwise take a long time to execute one instruction at a time.

The simplest form of continuous execution is to click the line of code you want to debug or examine in more detail with the right mouse button. The program executes up to the start of this line, then pauses. An alternative method is to position the cursor on this line, then press F7.

You can also pause execution at a specific line of code with a "breakpoint." There are several types of breakpoints. Breakpoints are explained in the following section.

Selecting Breakpoint Lines

Breakpoints can be tied to lines of code.

You can skip over those parts of the program that you don't want to examine by specifying one or more lines as breakpoints. The program executes up to the first breakpoint, then pauses. Pressing F5 continues program execution up to the next breakpoint, and so on. (You can halt execution at any time by pressing CTRL+C.)

There is no limit to the number of breakpoints.

You can set as many breakpoints as you like (limited only by available memory). There are several ways to set breakpoints:

- Double-click anywhere on the desired breakpoint line. The selected line is highlighted to show that it is a breakpoint. To remove the breakpoint, double-click the line a second time.
- Position the cursor anywhere on the line at which you want execution to pause. Press F9 to select the line as a breakpoint and highlight it. Press F9 a second time to remove the breakpoint and highlighting.
- Display the Set Breakpoint dialog box by choosing Set Breakpoint from the Watch menu. Select one of the breakpoint options that permits a line ("location") to be specified. The line at the cursor is the default breakpoint line in the Location field. If this line is not the desired location, enter the line number desired. (You must place a period in front of the line number, or CodeView will interpret the number as an absolute address.) To remove the breakpoint, use F9 or choose Edit Breakpoints from the Watch menu to display the Edit Breakpoints dialog box.

Not every line can be a breakpoint.

A breakpoint line must be a program line that represents executable code. You cannot select a blank line, a comment, or a declaration (such as a variable declaration or a segment specifier) as a breakpoint.

A breakpoint can also be set at an address. Type the address in *segment*:*offset* form in the Set Breakpoint dialog box. (Address breakpoints, unlike line breakpoints, are not saved in CodeView's status file, and therefore are not restored when you restart a debugging session.)

A breakpoint can be set to the name of a procedure if the procedure was declared with the **PROC** directive. If not, the procedure must contain a labeled line. Type the procedure's name or the line's label in the Set Breakpoint dialog box.

Once execution has paused, you can continue execution by clicking the F5=Go button in the display or by pressing F5. Execution continues to the next breakpoint. If there are no more breakpoints, execution continues to the end of the program, or until a fatal error occurs.

NOTE The Set Breakpoint dialog box contains a Commands text box. You can type Command-window commands in this box, separated by semicolons. These commands are executed when the breakpoint is reached. See the Command Window section of CodeView online help for a full description of Command-window commands.

Conditional Breakpoints

Breakpoints are not limited to specific lines of code. CodeView can also pause when a variable reaches a particular value or just changes value. This is a "conditional breakpoint." In previous versions of CodeView, conditional breakpoints are called "watchpoints" and "tracepoints."

You can associate a conditional breakpoint with a specific line of code, so that execution pauses at that line only if the variable has simultaneously reached a particular value or changed value. The check boxes in the Set Breakpoint dialog box select these other breakpoint types.

To pause execution when a variable reaches a particular value, type an expression that is usually false in the Expression field of the Set Breakpoint dialog box. For example, if you want to pause when the variable `looptest` equals 17, type `looptest == 17`.

To pause execution when a variable changes value, you need to type only the name of the variable in the Expression field. For large variables (such as arrays or character strings), you can specify the number of bytes you want checked (up to 32K) in the Length field. Execution pauses when any one of these values changes.

NOTE CodeView checks every conditional breakpoint after executing each line of source code. Unless you have enabled the use of the debug registers with the CodeView /R command-line option, this computational overhead greatly slows execution. (Execution is even slower if you are executing in Mixed mode or Assembly mode, because conditional breakpoints are checked after each machine instruction.)

For maximum speed when debugging, either associate conditional breakpoints with specific lines, or set conditional breakpoints only after you have reached the section of code that needs to be debugged. You can also use the Disable button in the Edit Breakpoints dialog box to temporarily suspend evaluation of a previously set conditional breakpoint.

Using Breakpoints

One of the most common bugs is a loop that executes too many or too few times. If you set a breakpoint on the statement that controls the loop statements, the program pauses after each iteration. With the loop variable or critical program variables in the Watch or Local windows, it should be easy to see what's going wrong in the loop.

You can specify how many times a breakpoint is reached before stopping.

You do not have to pause at a breakpoint the first time execution reaches it. CodeView lets you specify the number of times you want to ignore the breakpoint condition before pausing. Type the number in the Pass Count field of the Set Breakpoint dialog box. This feature can eliminate a lot of tedious single-stepping.

Another programming error is erroneously assigning a value to a variable that should not change. Type the variable in the Expression field of the Set Breakpoint dialog box. Execution breaks whenever this variable changes—even unintentionally.

You can assign new values to variables while execution is paused.

Breakpoints are a convenient way to pause the program so you can assign new values to variables. For example, if a limit value is set by a variable, you can change the value to see whether program execution is affected.

15.4.2 Single-Stepping

In single-stepping, CodeView pauses after each line of code is executed. The next line to be executed is highlighted.

There are two ways to single-step.

You can single-step through a program with the Step and Trace commands. Step (executed by pressing F10) steps over procedure calls. All the code in the procedure is executed, but it appears to you as if the procedure executed in a single step. Trace (executed by pressing F8) traces through every step of all procedures. Each line of the procedure is executed as a separate step.

You can alternate between Trace and Step as you like. The method you use depends only on whether you want to see what happens within a particular procedure. (Note that interrupt calls are always stepped over; you do not see individual steps of the execution.)

If CodeView cannot locate the source code for a procedure in the current directory, it pauses and asks for the name of the file that contains the source. If you cannot supply a source file, CodeView disassembles the executable code and displays that instead. (If you are executing in Source mode, and the source code for a procedure is not available, CodeView steps over the procedure, even if you use the Trace command.)

Note that breakpoints are active during both step and trace mode. If the procedure you step over contains a breakpoint, execution stops at the breakpoint.

You can trace through the program continuously (without having to press F8 at each step), using the Animate command from the Run menu. The speed of execution is controlled by the Trace Speed command from the Options menu. You can halt animated execution at any time by pressing any key.

15.4.3 Changing the Program Display Mode

The F3 function switches the display between Source mode, Mixed mode, and Assembly mode. You can also switch display modes by choosing the Source Window command from the Options menu and then selecting a display mode in the Source Window Options dialog box. (If the source-code text file cannot be located, CodeView automatically disassembles the executable file and displays it in assembly-language form.)

The Source mode shows the program as you wrote it. The Mixed mode and Assembly mode each expand macros and code-generating directives (such as **.STARTUP**) into assembly-language instructions. You can execute these instructions one at a time (rather than as a single item), and verify that the assembler has created the correct instructions from the macro or the directive.

Figures 15.5 and 15.6 show Mixed mode and Assembly mode, respectively, for the same code.

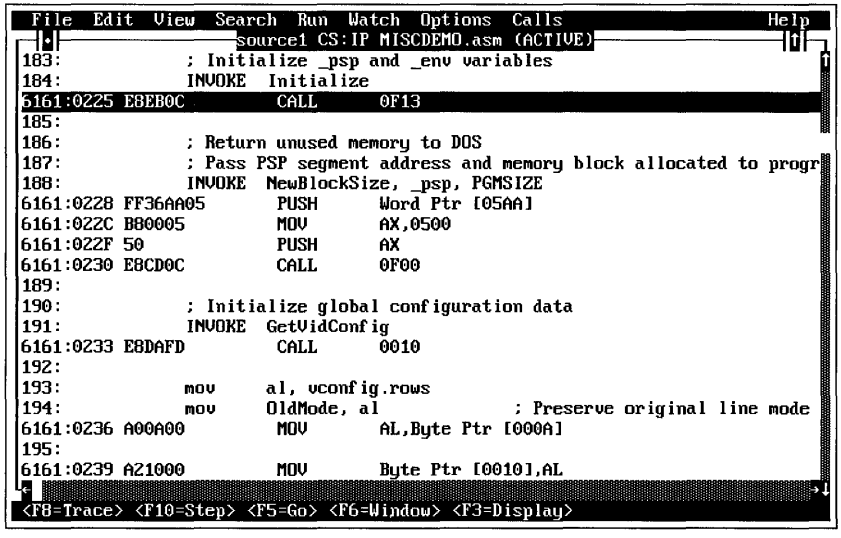


Figure 15.5 Source Window in Mixed Mode

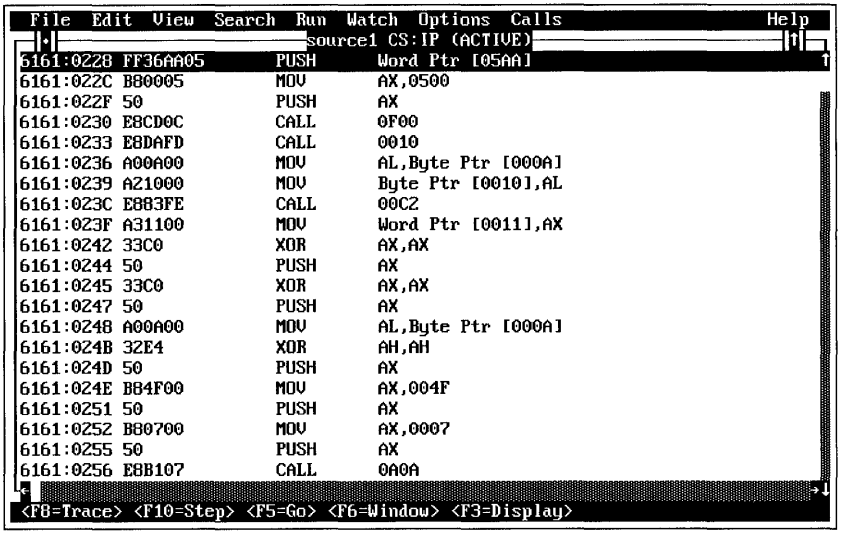


Figure 15.6 Source Window in Assembly Mode

15.5 Replaying a Debug Session

CodeView can automatically create a "tape" (a disk file) with the debugging instructions and input data you entered when testing a program. The tape can then be "replayed" to repeat the debugging process. You initiate recording by choosing the History On command from the Run menu. Choosing History On a second time terminates recording. The recording is saved in the .CVH file in the current directory.

Dynamic replay has several uses. The most obvious is repeating a debug session for the corrected version of a program. Dynamic replay usually works with slightly modified programs. However, the more you change the program, the less likely the new version will replay reliably.

You can also use the recording as a bookmark. You can quit after a long debugging session, then pick up the session later in the same place.

Dynamic replay makes it easy to correct a mistake.

Most importantly, dynamic replay allows you to back up when you make an error or overshoot the section of code with the bug. This feature is important because not all bugs appear on the first path of execution you try.

For example, you might have to manually execute a procedure many times before its bug appears. If you then enter a command that alters the machine's or program's status, thereby losing the information you need to find the cause of the bug, you would have to restart the program and manually repeat every debugging step to return to that point. Even worse, if you don't remember the exact sequence of events that exposed the bug, it could take hours to reproduce them.

Dynamic replay of a recorded tape eliminates this problem. Choose the Undo command from the Run menu to automatically restart the program and continuously execute every command up to (but not including) the last one you entered. You can repeat this process as many times as you like until you return to the desired point in execution.

You can add additional steps to an existing tape. Choose History On, then choose Replay. When replay has completed, perform whatever new debugging steps you want, then choose History On a second time to terminate recording. The new tape contains both the original and the added commands.

NOTE CodeView records only those mouse commands that apply to CodeView. Mouse commands recognized by the application being debugged are not recorded.

Replay Limitations under OS/2

There are some limitations to dynamic replay when debugging under OS/2:

- The program must not respond to asynchronous events. Replay under Presentation Manager is not currently supported because of this restriction.
- Breakpoints must be specified at specific source lines or for specific symbols (rather than by absolute addresses), or replay may fail.
- Single-thread programs behave normally during replay. However, one of the threads in a multithread program may cause an asynchronous event, violating the first restriction in this list. Multithread programs are therefore more likely to fail during replay.
- Multiprocess replay will fail. Each new process invokes a new CodeView session. The existence of multiple sessions makes it impractical to record the sequence of events if you execute commands in a session other than the original session.

15.6 Advanced CodeView Techniques

Once you are comfortable displaying and changing variables, stepping through the program, and using dynamic replay, you might want to experiment with the advanced techniques explained below.

Debugging OS/2 Programs

You can debug protected-mode and bound programs under CodeView. See the Debug Multiple Processes and Debug Multiple Threads sections of CodeView online help for information about executing threads and multiple processes.

Setting Command-Line Arguments

If your program retrieves command-line arguments, you can specify them with the Set Runtime Arguments command from the Run menu. Type the arguments in the Command Line field before you begin execution. (Arguments entered after execution begins cause an automatic restart.)

Opening Multiple Source Windows

You can open two Source windows at the same time. The windows can display two different sections of the same program, or one window can show the calling program and the other a procedure file. You can move freely between the windows, executing lines of code as you like.

Calling Procedures

Any procedure in your program (whether user-written or from a library) can be called from the Command window or the Watch window. In the Command window, use the Display Expression command as follows:

?*procname* (*arglist*)

The procedure *procname* is evaluated with the *arglist* arguments and the returned value is displayed in the Command window. (Note that CodeView cannot evaluate a function that returns an aggregate type.) In the Watch window, simply enter the procedure call. If the procedure does not return a value, the value displayed is the value of the AX register upon return from the procedure.

You can evaluate any procedure, not just those called by your program. All object code specified to the linker is linked into the program. Any public functions in this code can be evaluated from the Command window.

You can use this feature to call functions from within CodeView that you would not normally include in the final version of your program. For example, you could include the OS/2 API functions that control semaphores, then execute them from the Command window to manipulate the run-time environment at any point in the debugging process. (Remember that altering the environment during program execution may have unexpected side effects.)

Executing Faster when Using Breakpoints

Breakpoints can slow execution. You can increase CodeView's speed with the /R command-line option if you have an 80386/486-based computer and are running CodeView under DOS. This option enables the four debug registers, which support breakpoint-checking in hardware rather than in software. (The CodeView options are described in Section 15.7.)

Printing Selected Items

You can print all or part of the contents of any window with the Print command from the File menu. In the Print dialog box, a check box lets you print selected

text from the window, the material currently displayed in the window, or the complete contents of the window. Select text by dragging the mouse across it, or by holding down the SHIFT key and pressing the direction keys (LEFT, RIGHT, UP, DOWN).

By default, print output is to the file CODEVIEW.LST in the current directory. You can choose whether the new material is appended to an existing file or overwrites it, using the Append/Overwrite check box. If you want print output to go to a different file, type its name in the To File Name field. If you want the output to go to a printer, enter the appropriate device name such as LPT1 or COM2.

Redirecting CodeView Input and Output

The Command window accepts DOS-like commands that redirect input and output. These commands can also be included on the command line that invokes CodeView. Whatever items follow the /C option on the command line are treated as CodeView commands to be immediately executed at start-up.

```
CV /c "<infile; t>outfile" myprog
```

In the example above, input is redirected from `infile`, which can contain start-up commands for CodeView. When CodeView exhausts all commands in the input file, focus automatically shifts to the Command window. Output is sent to `outfile` and echoed to the Command window. The `t` must precede the `>` command for output to be sent to the Command window.

Redirection is a useful way to automate CodeView start-up. It also lets you keep a viewable record of command-line input and output, a feature not available with dynamic replay. No record is kept of mouse operations. Some applications (particularly interactive ones) may need modification to allow for redirection of input to the application itself.

Executing Faster with Additional Memory

If you are running DOS and your computer uses expanded or extended memory, you can increase CodeView's execution speed by selecting the /X or /E option. CodeView moves as much as it can of itself and the symbolic CodeView information to higher memory (above the first megabyte).

The /X option uses extended memory and gives the greatest speed increase. This option requires the HIMEM.SYS driver, which is included on your distribution disks. Add `DEVICE = HIMEM.SYS` to your CONFIG.SYS file to load HIMEM.SYS at boot time.

The /E option uses expanded memory. The speed increase is not as great as that supplied by the /X option. The expanded memory manager (EMM) must be LIM 4.0, and no single module's debug information can exceed 48K. If the symbol table exceeds this limit, try reducing file-name information by not specifying full path names at compile time and by specifying CodeView information (/Zi) only with those program modules that need debugging.

If you do not specify either /X or /E (or the /D disk-overlay option), CodeView automatically searches for the HIMEM.SYS driver and extended memory so it can implement the /X option. If it fails, CodeView searches for expanded memory to implement the /E option. If that search fails, CodeView uses a default disk overlay of 64K. (See the description of the /D option in the next section.)

15.7 CodeView Command-Line Options

The following options can be added to the command line that invokes CodeView. The Starting Up CodeView section of CodeView online help contains more information about these options.

Option	Description
/2	Two-monitor debugging. The display adapters must be configured for different addresses, such as Hercules® and VGA. The application is displayed on the primary monitor (the monitor the operating system normally directs output to), while CodeView's output appears on the secondary monitor.
/25	Display in 25-line mode.
/43	Display in 43-line mode.
/50	Display in 50-line mode.
/B	Display in black and white. This assures that the display is readable when a color display is not used. You should also specify this option along with the /2 option when the secondary monitor is black and white.
/C*commands*	Execute *commands* immediately on start-up. The commands must be separated with a semicolon. If any commands require a space, enclose the entire list in double quotation marks.
/D[[*buffersize*]]	Use disk overlays to increase the size of the program that can be debugged, where *buffersize* is the decimal size of the overlay buffer, in kilobytes. Smaller buffers leave more room for the program being debugged, while larger buffers increase the speed of execution. The acceptable range is 16K to 128K. The default size is 64K. (DOS only.)

Option	Description
/E	Use expanded memory for symbolic information and CodeView overlays. (DOS only.)
/F	Flip screen video pages (rather than swap). When your application does not use graphics, eight video screen pages are available. Switching from CodeView to the output screen is accomplished by directly selecting the appropriate video page. Cannot be used with /S. (DOS only.)
/G	Suppress "snow" on a CGA display. (DOS only.)
/I[[0 | 1]]	Control trapping of nonmaskable interrupts and 8259 interrupts. A value of 0 forces interrupt trapping on machines CodeView doesn't recognize as IBM-compatible. A value of 1 (the default) disables interrupt trapping. (DOS only.)
/K	Disable keyboard monitors (under OS/2) and keyboard interrupts (under DOS). This allows you to regain control of the computer under deadlock conditions, but prevents CodeView from recording keyboard entries when recording a debug session.
/L*dll*	Load symbolic information for the specified dynamic-link libraries (DLL). (OS/2 only.) This option is required only for DLLs loaded with **DOSLOADMODULE**. CodeView automatically loads debug information for statically linked DLLs.
/M	Disable CodeView's use of the mouse. This simplifies debugging programs that accept mouse commands.
/N[[0 | 1]]	Identical to /I, but applies only to nonmaskable interrupts. (DOS only.)
/O	Debug child processes ("offspring"). (OS/2 only.)
/R	Use 80386/486 hardware debug registers to speed execution. (DOS only.)

Option	Description
/S	Swap screen in buffers (rather than flip). When your program uses graphics, all eight video pages must be used. Switching from CodeView to the output screen is accomplished by saving the previous screen in a buffer. Cannot be used with /F. (DOS only.)
/TSF	Toggle (invert) the sense of the Statefileread switch in TOOLS.INI. If Statefileread is set to no (do not read the status file), the status file is read, and vice-versa.
/X	Use extended memory for CodeView and symbolic information. (DOS only.)

15.8 Customizing CodeView with the TOOLS.INI File

The TOOLS.INI file customizes the behavior and user interface of several Microsoft products. The TOOLS.INI file is a plain ASCII text file. You should place it in a directory pointed to the INIT environment variable. (If you do not use the INIT environment variable, CodeView looks for TOOLS.INI only in the CodeView source directory.)

The CodeView section of TOOLS.INI is preceded by the following line:

```
[cv]
```

If you run the protected-mode version of CodeView, use `[cvp]` instead. If you run both versions, include both: `[cv cvp]`. You can have separate sections for `cv` and `cvp` if you want different customizations.

Most of the TOOLS.INI customizations for CodeView control screen colors, but you can also specify such things as start-up commands or the default name of the file that receives CodeView output. See the Configure CodeView section of CodeView online help for full information about all TOOLS.INI switches that control CodeView.

15.9 Related Topics in Online Help

In addition to information covered in this chapter, information on the following topics can be found in online help.

Topic	Access
CodeView information	Choose "CodeView Debuggers" from the "Microsoft Advisor Contents" screen
ML command-line options	Choose "Macro Assembler" from the "Command Line" section of the "Microsoft Advisor Contents" screen

Chapter 16

Converting C Header Files to MASM Include Files

The H2INC utility translates C header files into MASM-compatible include files. C header files normally have the extension .H; MASM include files normally have the extension .INC. This is the origin of the program's name: "H to INC."

H2INC simplifies porting data structures from your C programs to MASM programs. This is especially useful when you have

- A program that mixes C code and MASM code with globally accessible data structures
- A program prototyped in C that you're translating to MASM for compactness and fast execution

The H2INC program translates data declarations, function prototypes, and type definitions. H2INC does not convert C code into MASM code. When H2INC encounters a C statement that would compile into executable code, H2INC ignores the statement and issues a warning message to the standard output.

H2INC accepts C source code compatible with Microsoft C 6.0 and creates include files suitable for MASM 6.0. These include files will not work with versions of MASM prior to 6.0.

H2INC is designed to translate project header files that you have written specifically for translation to MASM 6.0 include files. It is not designed to translate header files such as PM.H and WINDOWS.H.

This chapter explains how H2INC performs the C code translation and how the command-line options control the conversions.

16.1 Basic H2INC Operation

H2INC is designed to provide automatic translation of C declarations that you need to include in the MASM portions of an application. However, the set of C statements processed by H2INC must be those needed by and interpretable by MASM. H2INC converts only function prototypes, some preprocessor directives,

and C declarations outside the scope of procedures. For example, H2INC translates the C statement

```
#define MAX_EMPLOYEES 400
```

into this MASM statement:

```
MAX_EMPLOYEES EQU 400t
```

The *t* specifies the decimal radix.

H2INC does not translate C code into MASM code. Statements such as the following are ignored:

```
printf( "This is an executable statement.\n" );
```

H2INC translates declarations, not executable code.

By default, H2INC creates a single .INC file. If the C header file includes other header files, the statements from the original and nested files are translated and combined into one .INC file. This behavior can be changed with the /Ni option (see Section 16.2).

The program also preprocesses some statements, just as the C preprocessor would. For example, given the following statements, if `VERSION` is not defined, H2INC ignores the **#ifdef** block.

```
#ifdef VERSION
#define BOX_VALUE 4
#endif
```

If `VERSION` is defined, H2INC translates the statements inside the block from C syntax to MASM syntax.

H2INC normally discards comments. If you use the /C option, C comments are passed to the output file. If the line starts with a `/*` or `//`, the comment specifier is converted to a semicolon (;). If the line is part of a multiline comment, a semicolon is prefixed to each line.

H2INC ignores anything that is not a comment or that cannot be translated. These items do not appear in the output file. If H2INC encounters an error, it stops translating and deletes the resulting .INC file.

16.2 H2INC Syntax and Options

To run H2INC, type `H2INC` at the command-line prompt, followed by the options desired and the names of the .H files you want to convert:

H2INC [[*options*]] *file*.**H** ...

You can specify more than one *file*.H. File names are separated by a space. The contents of each *file*.H are translated into a single file in the current directory with the name *file*.INC. The original *file*.H is not altered.

The following lists describe the available options. You can specify more than one option. Note that the options are case sensitive except for /HELP.

H2INC recognizes /? to display a summary of H2INC syntax, and /HELP to invoke QuickHelp for H2INC. If QuickHelp is not available, H2INC displays a short list of H2INC options. This option is not case sensitive.

H2INC recognizes but ignores C 6.0 options that aren't specified in the following two lists.

Options Directly Affecting H2INC Output

This first list describes the options that directly affect the H2INC output:

Option	Action
/C	Passes comments in the .H file to the .INC file.
/Fa [[*filename*]]	Specifies that the output file contain only equivalent MASM statements. This is the default. If specified, the filename overrides the default, keeping the base name of the C header files and adding the .INC extension.
/Fc [[*filename*]]	Specifies that the output file contain equivalent MASM statements plus original C statements converted to comment lines.
/Mn	Assumes the **.MODEL** directive is not specified for the MASM source or the generated .INC files. Instructs H2INC to declare explicitly the distances for all pointers and functions.
/Ni	Suppresses the expansion of nested include files.
/Zu	Makes all structure and union tag names unique.

Options Indirectly Affecting H2INC Output

This second list describes the options that indirectly affect the H2INC output:

Option	Action
/AT	Specifies tiny memory model (.COM).
/AS	Specifies small memory model, the default.
/AC	Specifies compact memory model.
/AM	Specifies medium memory model.
/AL	Specifies large memory model.
/AH	Specifies huge memory model.
/D[[*const*[[=*value*]]]]	Defines a constant or macro.
/G0	Enables 8086/8088 instructions (default).
/G1	Enables 80186/80188 instructions.
/G2	Enables 80286 instructions.
/G3	Enables 80386 instructions. Changes the default word size to **DWORD**.
/G4	Enables 80486 instructions. Changes the default word size to **DWORD**.
/Gc	Specifies Pascal as the default calling convention.
/Gd	Specifies C as the default calling convention for functions (default).
/Gr	Specifies the **_fastcall** calling convention for functions. Generates a warning since H2INC does not translate **_fastcall** functions and prototypes.
/Ht	Enables generation of text equates. By default, text items are not translated.
/I*paths*	Searches named paths for include files before searching the paths in the INCLUDE environment variable. Paths are separated with a semicolon (;).
/J	Changes default character type from **signed char** to **unsigned char**.
/nologo	Suppresses display of the sign-on banner.

Option	Action
/Tc [[*filename*]]	Enables the processing of files whose name does not end in .H.
/u*ident*	"Undefines" one of the predefined identifiers. (See Section 16.3.1.)
/U	"Undefines" all predefined identifiers. (See Section 16.3.1.)
/w	Suppresses compiler warning messages; same as /W0.
/W0	Suppresses all warning messages.
/W1	Displays level 1 warning messages (default).
/W2	Displays level 1 and level 2 warning messages.
/W3	Displays level 1, 2, and 3 warning messages.
/W4	Displays all warning messages.
/X	Excludes search for include files in the standard places.
/Za	Disables language extensions (allows ANSI standard only).
/Zc	Causes functions declared as **_pascal** to be case insensitive.
/Ze	Enables language extensions (default).
/Zn *string*	Adds string to all names generated by H2INC. Used to eliminate name conflicts with other H2INC-generated include files.
/Zp{1 | 2 | 4}	Packs structure on a 1-, 2-, or 4-byte boundary, following C packing rules. Default is /Zp2.

16.3 Converting Data and Data Structures

The primary use of H2INC is to convert data automatically from C format into MASM format. This section shows how H2INC converts constants, variables, pointers, and other C data structures to definitions recognizable to MASM.

Since the names of the items translated by H2INC may be distinguished only by the case of the names, you should specify **OPTION CASEMAP:NONE** in any MASM files that include .INC files generated with H2INC.

16.3.1 User-Defined and Predefined Constants

H2INC translates constants from C to MASM format. For example, C symbolic constants of the form

```
#define CORNERS 4
```

are translated to MASM constants of the form

```
CORNERS EQU 4t
```

in cases where `CORNERS` is an integer constant or is preprocessed to an integer constant. See Section 1.2.4, "Integer Constants and Constant Expressions," for more information on integer constants in MASM.

TEXTEQU is new to MASM 6.0.

When the defined expression evaluates to a noninteger value, such as a floating-point number or a string, H2INC defines the expression with **TEXTEQU** and adds angle brackets to create text macros. By default, however, these **TEXTEQU** expressions are not added to the include file. Set the /Ht option to tell H2INC to generate **TEXTEQU** expressions.

```
/* #define PI 3.1415 */
PI TEXTEQU <3.1415>
```

H2INC uses this form when the expression is anything other than a constant integer expression. H2INC does not check the constant or string for validity. For example, although the following C definitions are valid, H2INC creates invalid string equates without generating an error.

These C statements

```
#define INT 6
#define FOREVER for(;;)
```

generate these MASM statements:

```
INT EQU 6t
FOREVER TEXTEQU <for(;;)>
```

The first **#define** statement is invalid because **INT** is a MASM instruction; in MASM 6.0, instructions are reserved and cannot be used as identifiers. The **for** loop definition is invalid because MASM cannot assemble C code.

Predefined constants control the contents of .INC files.

You can make use of the following predefined constants in your C code to conditionally generate the code in .INC files. The predefined constants and the conditions under which they are defined are

Predefined Constant	When Defined
_H2INC	Always defined
M_I86	Always defined
MSDOS	Always defined
_MSC_VER	Defined as 600 for this release
M_I8086	Defined if /G0 is specified
M_I286	Defined if /G0 is not specified
NO_EXT_KEYS	Defined if /Za is specified
_CHAR_UNSIGNED	Defined if /J is specified
M_I86SM	Defined if /AS is specified
M_I86MM	Defined if /AM is specified
M_I86CM	Defined if /AC is specified
M_I86LM	Defined if /AL is specified
M_I86HM	Defined if /AH is specified

For example, if your C header file includes definitions which are specific to the C portion of the program or otherwise are not appropriate for translation by H2INC, you can bracket the C-specific code with

```
#ifndef _H2INC
     /* C-specific code */
#endif
```

In this case, only the C compiler processes the bracketed code.

The /u and /U options affect these predefined constants. The /u*arg* option undefines the constant specified as the argument. The /U option disables the definition of all predefined constants. Neither /u or /U affects constants defined by the /D option.

H2INC places an **OPTION EXPR32** directive in the .INC file so that MASM correctly handles long integers within expressions. This means that the .INC files as well as all the .ASM files which include .INC files created with H2INC will resolve integer expressions in 32 bits instead of 16 bits.

16.3.2 Variables

H2INC translates variables from C to MASM format. For example, this C declaration

```
int my_var;
```

is translated into the MASM declaration

```
EXTERNDEF my_var:SWORD
```

H2INC converts C variable types to MASM types as follows:

C Type	MASM Type
char	**BYTE** or **SBYTE** (controlled by /J option)
signed char	**SBYTE**
unsigned char	**BYTE**
short	**SWORD**
unsigned short	**WORD**
int	**SWORD** (**SDWORD** with /G3 or /G4 option)
unsigned int	**WORD** (**DWORD** with /G3 or /G4 option)
long	**SDWORD**
unsigned long	**DWORD**
float	**REAL4**
double	**REAL8**
long double	**REAL10**

H2INC assumes that a variable is external unless the variable is explicitly declared as static. For example, the C declaration

```
long big_data;
```

is converted to this MASM declaration:

```
EXTERNDEF big_data:SDWORD
```

See Sections 1.2.6, "Data Types," and 4.1.1, "Allocating Memory for Integer Variables," for more information on MASM data types, and Section 8.2.2, "Declaring Symbols Public and External," for information on **EXTERNDEF**.

H2INC does not allocate space for arrays since all variables are assumed to be external. For example, the C declaration

```
int two_d[10][20];
```

translates to

```
EXTERNDEF two_d:SWORD
```

H2INC does not translate static variables, since the scope of these variables extends only to the file where they are declared.

16.3.3 Pointers

H2INC translates C pointer variables into their MASM equivalents. The C declarations

```
int *ptr_var;
char NEAR *pCh;
```

are translated into these MASM statements:

```
EXTERNDEF ptr_var:PTR SWORD
EXTERNDEF pCh:NEAR PTR SBYTE
```

If you set the /Mn option, H2INC specifies all distances explicitly (for example, **NEAR PTR** instead of **PTR**). If /Mn is not set, the distances are generated only when they differ from the default values implied by the memory model specified by the /A command-line option.

H2INC converts **_segment** and **_based** variables to type **WORD** in MASM.

See Sections 1.2.6, "Data Types," and 3.3, "Accessing Data with Pointers and Addresses," for information about MASM pointers.

16.3.4 Structures and Unions

H2INC translates C structures and unions into their MASM equivalents. H2INC modifies the C structure or union definition to account for differences from MASM structure and union definitions. This list describes these modifications.

- C allows a structure or union variable to have the same name as the type name, but MASM does not. The H2INC /Zu option prevents the structure name from matching a variable or instance by prefixing every MASM structure name with `@tag_`.
- If a C structure or union definition does not have a name, H2INC supplies one for the MASM conversion. These generated structure names take the form `@tag_`*n*, where *n* is an integer that starts at zero and is incremented for each structure name H2INC generates.

- If the /Zn option is specified, H2INC inserts the given string between the underscore and the number in the generated structure names. This eliminates name conflicts with other H2INC-generated include files.
- H2INC adds the alignment value to the converted structure definition.

The following examples show how these rules are applied when converting structures. (Union conversions are not shown; they are handled identically.) These examples assume that the C header file defines an alignment value of 2. (See Section 5.2.1, "Declaring Structure and Union Types," for information on alignment values.)

The following named C structure definition

```
struct file_info
{
   unsigned char  file_addr;
   unsigned int   file_size;
};
```

is converted to the following MASM form. Except for explicitly specifying the alignment value, the conversion is direct:

```
file_info          STRUCT 2t
file_addr          BYTE          ?
file_size          WORD          ?
file_info          ENDS
```

If the same C structure definition is converted using the /Zu option, the `@tag_` prefix is added to the structure's name so that the name does not duplicate the name of a structure component:

```
@tag_file_info     STRUCT 2t
file_addr          BYTE          ?
file_size          WORD          ?
@tag_file_info     ENDS
```

If the original C structure definition is modified to be an unnamed-type declaration of a specific instance (`myfile`)

```
struct
{
   unsigned char  file_addr;
   unsigned int   file_size;
} myfile ;
```

its MASM conversion looks like the following example. (The specific integer added to the `@tag_` prefix is determined by the sequence in which H2INC creates tag names.)

```
@tag_7          STRUCT 2t
file_addr       BYTE ?
file_size       WORD ?
@tag_7          ENDS
EXTERNDEF       C myfile:@tag_7
```

Nested structures may have as many levels as desired; they are not limited to one level. Nested structures are "unnested" (expanded) in the correct hierarchical sequence, as shown with the C structure and H2INC-generated code in this example.

```
/* C code: */
struct phone
{
    int  areacode;
    long number;
};

 struct person
{
    char   name[30];
    char   sex;
    int    age;
    int    weight;
    struct phone;
} Jim;

; H2INC generated code:
phone           STRUCT 2t
areacode        SWORD          ?
number          SDWORD         ?
phone           ENDS

person          STRUCT 2t
name            SBYTE          30t DUP (?)
sex             SBYTE          ?
age             SWORD          ?
weight          SWORD          ?
STRUCT
  areacode      SWORD          ?
  number        SDWORD         ?
ENDS
person          ENDS

EXTERNDEF       C Jim:person
```

See Section 5.2 for information on MASM structures and unions.

16.3.5 Bit Fields

H2INC translates C bit fields into MASM records. H2INC looks at a structure definition; if it consists only of bit fields of the same type and if the total size of the bit fields does not exceed the type of the bit fields, then H2INC outputs a **RECORD** definition with the name of the structure. All bit-field names are modified to include the structure name for uniqueness, since record fields have global scope in MASM.

For example,

```
struct s
{
   int i:4;
   int j:4;
   int k:4;
}
```

becomes:

```
s       RECORD  @tag_0:4,
                k@s:4,
                j@s:4,
                i@s:4
```

The `@tag` variable pads out the record to the type size of the bit fields so alignment of the structures will be correct.

If the bit fields are too large, are not of the same type, or are mixed with fields that are not bit fields, H2INC generates a **RECORD** definition inside the structure and then uses the definition.

For example,

```
struct t
{
    int i;
    unsigned char a:4;
    int j:9;
    int k:9;
    long l;
}  m;
```

becomes:

```
t          STRUCT 2t
i          SWORD       ?
rec@t_0    RECORD      @tag_1:4,
           a@t:4
@bit_0     rec@t_0     <>
rec@t_1    RECORD      @tag_2:7,
           j@t:9
@bit_1     rec@t_1     <>
rec@t_2    RECORD      @tag_3:7,
           k@t:9
@bit_2     rec@t_2     <>
l          SDWORD      ?
t          ENDS

EXTERNDEF C m:t
```

Notice that `j` and `k` are not packed because their total size exceeds the 16 bits of an integer in C.

Since the `@bit` field names are local to the structure, these begin with `0` for each structure type; the `@rec` variables have global scope and so their number always increases.

The C bit-field declaration

```
struct SCREENMODE
{
   unsigned int disp_mode : 4;
   unsigned int fg_color  : 3;
   unsigned int bg_color  : 3;
};
```

is converted into the following MASM record:

```
SCREENMODE      RECORD          disp_mode@SCREENMODE:4,
                                fg_color@SCREENMODE:3,
                                bg_color@SCREENMODE:3
```

See Section 5.3 for information about MASM records.

16.3.6 Enumerations

H2INC converts C enumeration declarations into MASM **EQU** definitions that are treated as standard integer constants. If the C declaration is not assigned a value, the H2INC generates an **EQU** statement that supplies a value equivalent to its position in the list. For example, the C enumeration declaration

```
enum tagName
{
   id1,
   id2,
   id3 = 42,
   id4
};
```

is converted into the following **EQU** statements:

```
id1     EQU     0t
id2     EQU     1t
id3     EQU     42t
id4     EQU     43t
```

See Section 1.2.4 for information on MASM integer constants.

16.3.7 Type Definitions

All type definitions using C base types are translated directly. For example, H2INC converts the C type definitions

```
typedef int INTEGER;
typedef float FLOAT;
```

to these MASM forms:

```
INTEGER TYPEDEF SWORD
FLOAT   TYPEDEF REAL4
```

Pointer types are converted in a similar fashion. The following declarations

```
typedef int *PINT
typedef int **PINT
typedef int far *PINT
```

become (respectively)

```
PINT TYPEDEF PTR SWORD
PINT TYPEDEF PTR PTR SWORD
PINT TYPEDEF FAR PTR SWORD
```

Addressing mode determines pointer size.

The number of bytes allocated for the pointer is set by the addressing mode you have selected unless if is specifically overridden in the type definition.

C statements using **typedef** which convert to a type with the same name as the type do not generate errors, but are not converted. For example, H2INC does not convert

```
typedef int SWORD
typedef unsigned char BYTE
```

since these typedef statements would generate these MASM statements:

```
SWORD   TYPEDEF SWORD
BYTE    TYPEDEF BYTE
```

See Section 3.3, "Accessing Data with Pointers and Addresses," for information on using **TYPEDEF** in MASM 6.0.

16.4 Converting Function Prototypes

When H2INC converts C function prototypes into MASM function prototypes, the elements of the C syntax are converted into the corresponding elements of the MASM syntax.

The syntax of a C function prototype is

[[*storage*]] [[*distance*]] [[*ret_type*]] [[*langtype*]] *label* ([[*parmlist*]])

In C syntax, *storage* can be **STATIC** or **EXTERN**. H2INC does not translate static function prototypes because static functions are visible only within the current source module, and standard include files do not contain executable code.

Procedures for returning values depend on the *langtype* specified.

In C, the *ret_type* is the data type of the return value. Because the MASM **PROTO** directive does not specify how to handle return values, H2INC does not translate the return type. However, H2INC checks the *langtype* specified in the C prototype to determine how particular languages return the value—through the stack or through registers.

For the Pascal, FORTRAN, or Basic *langtype* specifications, H2INC appends an additional parameter to the argument list if the return type is longer than four bytes. This parameter is always a near pointer with the type of the return value. If the value of the return value type is not supported, this parameter is an untyped near pointer.

For the **_cdecl** *langtype* specification in the C prototype, all returned data is passed in registers (AX or AX plus DX). There is no restriction on the return type. Additional parameters are not necessary.

The *langtype* represents the naming and passing conventions for a language type. H2INC accepts the following C language types and converts them to their corresponding MASM language types:

C Language Type	MASM Language Type
_cdecl	**C**
_fortran	**FORTRAN**
_pascal	**PASCAL**
_stdcall	**STDCALL**
_syscall	**SYSCALL**

H2INC explicitly includes the *langtype* in every function prototype. If no language type is specified in the .H file prototype, the default language is **_cdecl** (unless the default is overridden by the /Gc command-line option).

In the MASM prototype syntax, the *label* is the name of the function or procedure.

If you select the /Mn option, H2INC specifies the *distance* of the function (near or far), whether or not the C prototype specifies the distance. If /Mn is not set, H2INC specifies the distance only when it is different from the default distance specified by the memory model.

If the C prototype's parameter list ends with a comma plus an ellipsis (`, ...`), the function can accept a variable number of arguments. H2INC converts this to the MASM form: a comma followed by the **:VARARG** keyword (`, :VARARG`) appended to the last parameter.

H2INC does not translate **_fastcall** functions. Functions explicitly declared **_fastcall** (or invoking H2INC with the /Gr option) generate a warning indicating that the function declaration has been ignored.

The following examples show how the preceding rules control the conversion of C prototypes to MASM prototypes (when the memory model default is small). The example function is `my_func`. The **TYPEDEF** generated by H2INC for the **PROTO** is given along with the **PROTO** statement.

```
/* C prototype */
            my_func (float fNum, unsigned int x);
; MASM TYPEDEF
            @proto_0 TYPEDEF PROTO C :REAL4, :WORD
; MASM prototype
            my_func  PROTO   @proto_0
```

```
/* C prototype */
            extern my_func1 (char *argv[]);
; MASM TYPEDEF
            @proto_1  TYPEDEF  PROTO C :PTR PTR SBYTE
; MASM prototype
            my_func1  PROTO    @proto_1

/* C prototype */
            struct vconfig _far * _far pascal my_func2 (int, scri );
; MASM TYPEDEF
            @proto_2  TYPEDEF  PROTO FAR PASCAL :SWORD, :scri
; MASM prototype
            my_func2  PROTO    @proto_2

/* C prototype */
            long pascal my_func3 (double y, struct vconfig vc);
; MASM TYPEDEF
            @proto_3  TYPEDEF  PROTO PASCAL :REAL8, :vconfig
; MASM prototype
            my_func3    PROTO    @proto_3

/* C prototype */
            void _far _cdecl myfunc4 ( char _huge *, short);
; MASM TYPEDEF
            @proto_4  TYPEDEF  PROTO FAR C :FAR PTR SBYTE, :SWORD
; MASM prototype
            myfunc4  PROTO  @proto_4

/* C prototype */
            short my_func5 (void *);
; MASM TYPEDEF
            @proto_5  TYPEDEF  PROTO C :PTR
; MASM prototype
            my_func5  PROTO  @proto_5

/* C prototype */
            char my_func6 (int, ...);
; MASM TYPEDEF
            @proto_6  TYPEDEF  PROTO C :SWORD, :VARARG
; MASM prototype
            my_func6  PROTO  @proto_6

/* C prototype */
            typedef char * ptrchar;
            ptrchar _cdecl my_func7 (char *);
; MASM TYPEDEF
            @proto_7  TYPEDEF  PROTO C :PTR SBYTE
; MASM prototype
            my_func7  PROTO  @proto_7
```

See Section 7.3.6, "Declaring Procedure Prototypes," for more information on prototypes and Chapter 20, "Mixed-Language Programming," for information on calling conventions and mixed-language programs.

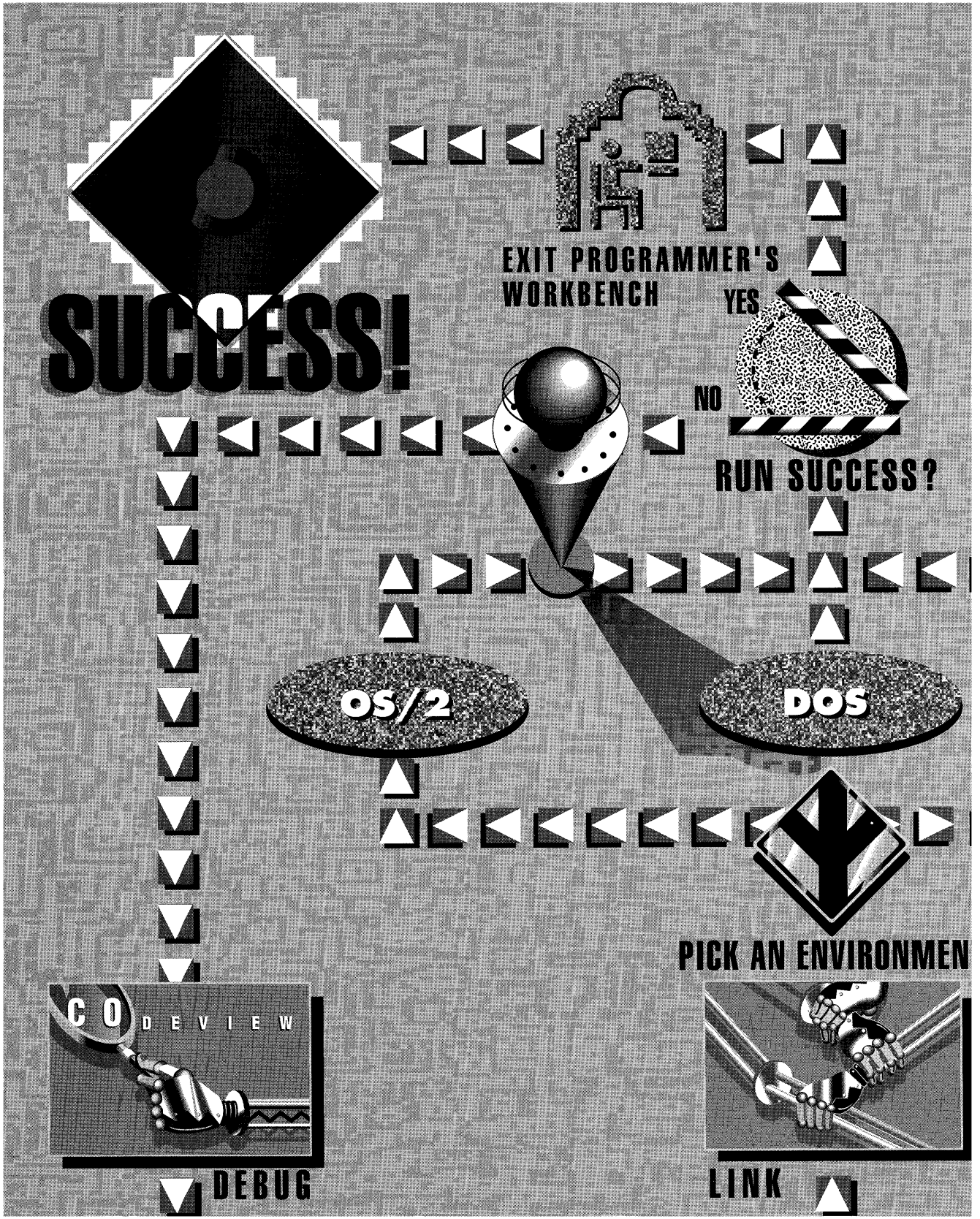
16.5 Related Topics in Online Help

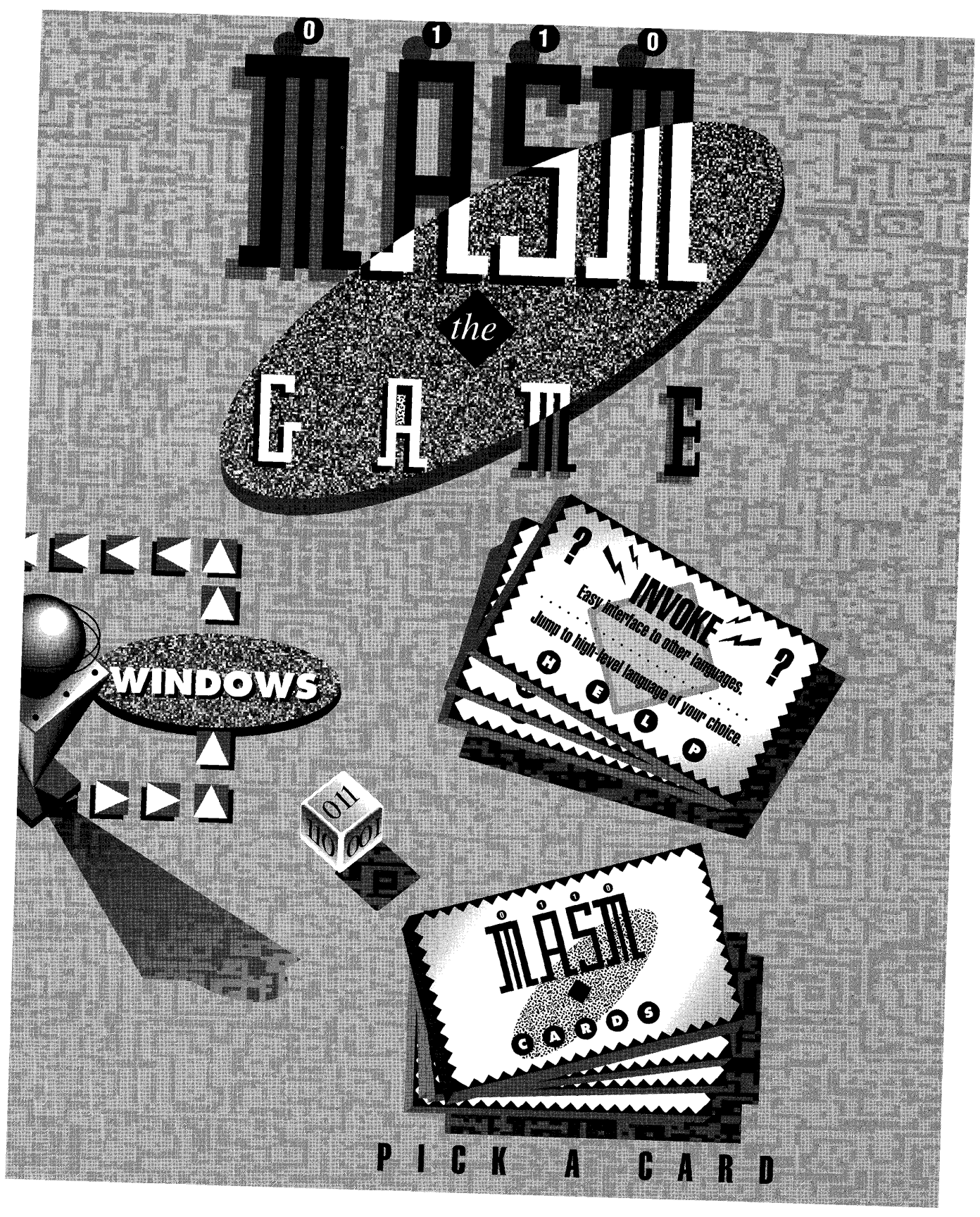
In addition to information covered in this chapter, information on the following topics can be found in online help.

Topic	Access
INCLUDE Directive	From the "MASM 6.0 Contents" screen, choose "Directives" and then "Miscellaneous"
Include files	From the "MASM 6.0 Contents" screen, choose "Example Code"; then choose "INCLUDE Files" to see a list of the include files provided with MASM 6.0
MASM data types (constants, variables, structures, unions, real numbers, records)	From the "MASM 6.0 Contents" screen, choose "Directives"; then choose "Data Allocation" or "Complex Data Types"
TYPEDEF	From the "MASM 6.0 Contents" screen, choose "Directives" and then "Complex Data Types"
Procedures and prototypes	From the "MASM 6.0 Contents" screen, choose "Directives"; then choose "Procedure and Code Labels"

Part 3
Advanced Topics

Chapters


SUCCESS!
EXIT PROGRAMMER'S WORKBENCH
YES
NO
RUN SUCCESS?
OS/2
DOS
PICK AN ENVIRONMEN
CODEVIEW
DEBUG
LINK


MASM
the
GAME
WINDOWS
INVOKE
Easy interface to other languages.
Jump to high-level language of your choice.
HELP
MASM
CARDS
PICK A CARD

Chapter 17
Writing OS/2 Applications

Microsoft Operating System/2 (OS/2) takes full advantage of 80286 and later processors. It supports memory far beyond the DOS 640K limit and offers a rich set of multitasking system calls. Although OS/2 is much more powerful than DOS, you may ultimately find it easier to program for OS/2.

This chapter shows how to develop an OS/2 application and how to write dual-mode programs to run under both OS/2 and DOS.

To write OS/2 applications, you must learn OS/2 system calls. While this chapter mentions a few of these calls, you should consult the references listed in the book's introduction to learn more about OS/2 system functions.

OS/2 supports two modes—real mode, which emulates the DOS environment, and protected mode, which supports all the advanced features. For simplicity's sake, the rest of this chapter equates OS/2 with protected mode.

NOTE Examples in this chapter support OS/2 1.*x*. Future versions of OS/2 may support different calling conventions.

17.1 OS/2 Overview

There are three steps in developing OS/2 or dual-mode applications:

1. Write the source code, using procedure calls rather than interrupts to call system functions.
2. Assemble and link the program with OS2.LIB.
3. Optionally, convert the program so that it can run under both OS/2 and DOS.

This chapter explains each of these steps, first looking at specific differences in how you write DOS and OS/2 code. Then it illustrates the development of a simple OS/2 program. Finally, the chapter discusses register initialization and additional OS/2 utilities.

17.2 Differences between DOS and OS/2

Assembly language is assembly language. Most machine instructions you use in a DOS program are the same instructions you use in an OS/2 program. When you start making calls to the operating system, however, things change.

You should understand the following differences between the two operating systems before attempting to write an OS/2 program.

System Calls

System calls control I/O and screen access.

OS/2 is similar to DOS in that it offers a series of system calls that perform tasks such as opening or closing a disk file. The OS/2 system calls that handle keyboard input (**KbdCharIn**, for example) correspond to the interrupt 16h instructions in DOS. The OS/2 system calls for screen output (**VioScrollDn**, for example) correspond to DOS interrupt 10h calls. And the OS/2 disk and operating-system calls (**DosGetDateTime**, for example) correspond to DOS interrupt 21h calls.

The effect is similar, but the way you actually make the calls is different. In DOS, you issue an interrupt. In OS/2, you make the system call with the **INVOKE** directive or the **CALL** instruction.

New Instructions

OS/2 is designed for advanced processors, and you may want to write programs that take advantage of the new instructions available on the 80286–80486. To use the new instructions and still target OS/2 1.*x*, place a **.286** directive at the beginning of your source code.

In general, you should avoid the directives that enable privileged instructions (**.286P**, **.386P**, and **.486P**), unless you are writing system-level code.

Many OS/2 programs can be converted to run under DOS as well. To write programs to run on all DOS and OS/2 systems, use the default processor setting (**.8086**).

The OS/2 Library

MASM 6.0 provides OS2.INC and OS2.LIB.

OS/2 programs must be linked to the system-call import library, OS2.LIB. The best way to perform this task is to use the **INCLUDELIB** directive, as shown in the example in the next section. In addition, you can include the OS2.INC file as an alternative to adding the prototypes for the OS/2 functions to your file.

The OS2.LIB file makes system calls possible; it contains import definitions for all system calls. An import definition specifies the name of a procedure and the dynamic-link library (DLL) where the procedure resides. You can learn more about DLLs in Chapter 18, "Creating Dynamic-Link Libraries." To create an OS/2 application, however, you need to know only that OS2.LIB is required.

Start-Up Code

Unlike DOS, OS/2 automatically initializes all segment registers as required by the standard segment model. No special start-up sequence is required, although OS/2 places useful information in AX, BX, and CX (see Section 17.6, "Register and Memory Initialization") that you may want to save.

Calling Conventions

OS/2 1.*x* uses the Pascal calling convention.

OS/2 system calls follow the Pascal calling and naming conventions. One way to enforce these conventions is to specify **PASCAL** in the **.MODEL** directive, then use the **INVOKE** directive to generate the correct code. Another is to include the OS2.INC file, which uses the **PROTO** directive to prototype the functions to follow the Pascal conventions. The prototypes specify Pascal as the calling convention. OS/2 functions return a value in AX. A nonzero value indicates an error. All registers except AX are preserved.

The OS/2 2.*x* operating system uses different calling conventions. See the documentation provided with that product.

Exit Code

To exit an OS/2 program, call the OS/2 system function **DosExit**. If you use the **.EXIT** directive and the **OS_OS2** attribute of the **.MODEL** statement, the assembler automatically generates the proper system call if you have a prototype for **DosExit**.

Segment Restrictions

Although OS/2 makes some operations easier, it does impose restrictions on the programmer. You cannot do segment arithmetic. That is, you cannot attempt to measure the distance between segments by subtracting one segment from another. In general, you also cannot add values to segment registers. Either operation may cause a protection violation, which would immediately terminate the program.

Under OS/2, segment registers do not hold physical addresses; they hold "segment selectors." A segment selector is an index into the system's descriptor tables that hold the actual addresses. You can copy the segment selector or use it to access data, but you should not try to modify it.

Huge pointer arithmetic is therefore different under OS/2. Under DOS, you can handle huge pointers easily by checking the **OVERFLOW?** flag after you increment or add to an offset address. If the result overflows (exceeds 64K), then you increment the segment address. Under OS/2, manipulation of huge pointers requires special techniques. See your OS/2 documentation for more information.

17.3 A Sample Program

The following program prints `Hello, world`. It runs under OS/2 protected mode.

```
; HELLO.ASM
;
                .MODEL  small, pascal, OS_OS2
                .286

                INCLUDELIB  os2.lib
                INCLUDE     os2.inc

                .STACK
                .DATA
message         BYTE    "Hello, world.", 13, 10  ; Message to print
bytecount       DWORD   ?                        ; Holds number of
                                                 ;  bytes written
                .CODE

                .STARTUP
                push        1                   ; Select standard output
                push        ds                  ; Pass address of message
                push        OFFSET message
                push        LENGTHOF message    ; Pass length of message
                push        ds                  ; Pass address of count
                push        OFFSET bytecount    ;  returned by function
                call        DosWrite            ; Call system write
                                                ;  function
                .EXIT       0                   ; Exit with 0 return code
                END
```

.STARTUP and .EXIT automatically generate code.

The **.STARTUP** and **.EXIT** directives are very useful because they automatically produce correct code for the operating-system type specified with the **.MODEL** directive (see Section 2.2, "Using Simplified Segment Directives"). As described in Section 17.6, OS/2 initializes all segment registers; therefore, **.STARTUP** does nothing but indicate the starting point. To correctly exit an OS/2 program, you must call the **DosExit** function. The **DosExit** prototype is always available to MASM programs.

In the example above, **.EXIT** automatically generates the following code under OS/2:

```
                                    .EXIT     0
0011  6A 01               *   push    +000000001h ; Action 1 ends all threads
0013  6A 00               *   push    +000000000h ; Pass 0 return code
0015  9A ---- 0000 E      *   call    DosExit     ; Call system function
                                     END
```

Between **.STARTUP** and **.EXIT**, the entire program consists of a single call to the **DosWrite** function. The program pushes the parameters on the stack and then

makes the call. No **POP** or **ADD** instructions are needed to restore the stack after **DosWrite** returns; **DosWrite** observes the Pascal calling convention and restores the stack itself before returning.

The **.MODEL** statement helps ensure that the assembler produces correct code for calling **DosWrite**:

```
        .MODEL  small, pascal, OS_OS2
```

When you run HELLO.EXE, OS/2 looks at the import definitions in the executable-file header and makes sure that all needed DLLs are in memory. It then loads any needed DLLs not already in memory.

The assembler must be informed that **DosWrite** and **DosExit** are far and observe the Pascal calling convention. This information is in the prototype.

In the call to **DosWrite**, note that although `OFFSET message` is an immediate operand, the program pushes it directly onto the stack. This operation is legal on 80186–80486 processors but not on the 8086 or 8088:

```
        push    OFFSET message
```

The processors you want to target determine the instructions you should use.

Since OS/2 programs can execute only on the 80286 or later processors, it is reasonable to use extended operations not supported by the 8086. However, if you want to write a program that can be converted to run under both OS/2 and DOS (as shown in Section 17.5), then you should write code that can run on the 8086. For example,

```
mov ax, OFFSET msg
        push    ax
```

The following revision of the sample program illustrates the usefulness of the **INVOKE** directive. This version does everything the previous example did with far fewer statements:

```
; HELLO.ASM

            .MODEL  small, pascal, OS_OS2

            INCLUDE     os2.inc
            INCLUDELIB  os2.lib

            .STACK
            .DATA
message     BYTE   "Hello, world.", 13, 10  ; Message to print
bytecount   DWORD   ?                       ; Holds number of
                                            ;  bytes written
            .CODE
            .STARTUP
```

```
INVOKE  DosWrite,
        1,
        ADDR message,
        LENGTHOF message,
        ADDR bytecount

.EXIT 0                         ; Exit with return code 0
END
```

The **INVOKE** directive generates a call to the given procedure after first pushing all other arguments on the stack. Like a call statement in a high-level language, the **INVOKE** directive handles types in a sophisticated way.

17.4 Building an OS/2 Application

The easiest way to assemble and link the program is from the Programmer's WorkBench (PWB). From the Options Menu, select Link Options and choose OS/2 Application. When you select Build from the Make menu, PWB calls ML and LINK, passing the proper options.

From the command line, type

```
ML hello.asm
```

The next section discusses how to "bind" the program—that is, convert it so that it runs under either DOS or OS/2.

17.5 Binding OS/2 MASM Programs

You can convert many OS/2 programs to run under both OS/2 and DOS 3.*x*. This conversion is called "binding" because it binds system calls to the API.LIB file provided with MASM 6.0. This file simulates OS/2 functions under DOS. The program must use a restricted set of system calls or it cannot be bound.

OS/2 function calls are known collectively as the applications program interface (API). If you restrict your system calls to a subset of these functions known as the Family API, the program can be bound. See the *Microsoft Operating System/2 Programmer's Reference* for a list of the Family API functions.

Online help also provides information on these utilities.

If you use PWB, binding is easy. Select Bound Application from the LINK Options command in the Options menu. PWB does the rest, calling the BIND.EXE utility.

If you want to bind the program to run under either OS/2 or DOS, use this command line:

```
ML /Fb hello.asm
```

You can use system calls outside the Family API provided that you never use them when running under DOS. The program can check the operating system and, if running under OS/2, can execute system calls that do not belong to the Family API. To follow this strategy, list OS/2-only calls with the BIND's /N option. It is the program's responsibility to make sure these calls are never made under DOS; otherwise, execution is terminated.

17.6 Register and Memory Initialization

When you execute an OS/2 program, OS/2 stores information about the program directly in registers. With DOS programs, the information is kept in a separate program segment prefix (PSP). The registers hold these values when an OS/2 program begins:

Register	Contents at Program Start
AX	Segment address of program's environment
BX	Offset of command-line arguments within the environment
CX	Length of near data area (DGROUP)
SP	Offset of the top of the stack within the stack segment
CS:IP	Program's entry point
DS	Segment address of near data area (DGROUP)
SS	Segment address of stack

Note that OS/2 automatically initializes SS:SP correctly. If the **.MODEL** directive specifies **FARSTACK**, SS is initialized to its own segment address. If the model is **NEARSTACK**, OS/2 sets SS to DGROUP and SP to the top of the stack within DGROUP.

You may want to save the AX, BX, and CX registers at startup.

Upon start-up, AX, BX, and CX all contain information highly useful to some programs. If you want to access the program's command-line arguments or know the size of DGROUP, you must save the contents of these registers immediately:

```
FPBYTE TYPEDEF FAR PTR BYTE

        .DATA

args    FPBYTE  0
cmds    FPBYTE  0

        .CODE
```

```
mov WORD PTR args[0], ax ; Save segment of args
mov WORD PTR args[2], 0  ; Offset is 0
mov WORD PTR cmds[0], ax ; Save segment of cmds
mov WORD PTR cmds[2], bx ; Save offset of cmds
```

The AX register points to the segment value of the start of the program's environment. AX:BX points to the starting address of arguments within the environment, the first of which is the program name. This name is followed by a null (zero) byte and the command-line arguments exactly as typed at the command prompt. A second null marks the end of the arguments.

If you use simplified segments, .DATA is equivalent to DGROUP.

Under OS/2, the data segment register, DS, contains the segment of the near data area, DGROUP. If you use simplified segment directives, this is the **.DATA** segment. You must place one data segment in a group called DGROUP if you do not use the simplified directives:

```
_DATA  SEGMENT WORD PUBLIC 'DATA'
       .
       .
       .
_DATA  ENDS

DGROUP GROUP _DATA
       ASSUME DS:DGROUP
```

Calling the group anything other than DGROUP, or not having a DGROUP, causes an error. Only the memory required by the program is allocated by OS/2. This means that the system has space in reserve for later memory requests and for other programs.

17.7 Other OS/2 Utilities

In addition to LINK and BIND, MASM 6.0 provides other utilities useful for working with OS/2.

EXEHDR

The EXEHDR utility examines and can modify a DOS, Windows, or OS/2 executable file header. In the case of OS/2 and Windows, EXEHDR reports a great deal more information: specifically, it displays the contents of segment tables and lists the attributes of the individual segments.

IMPLIB

The IMPLIB utility creates an import library that you can use when linking with a DLL or group of DLLs. Generally, there are three steps in using a DLL:

1. Copy the DLL to a directory listed in your CONFIG.SYS LIBPATH setting.
2. Run IMPLIB on the DLL to create an import library, or write a module-definition file.
3. Link the import library or module-definition file with any application that uses the DLL.

An import library does not contain executable code but does contain the name and location of dynamic-link calls. These calls are resolved during run time.

Chapter 18 goes into more detail about how to write DLLs.

17.8 Module-Definition Files

You can create a module-definition file for an application. A module-definition file is a text file that contains statements that give directions to the linker. These statements can alter the attributes of individual segments—for example, whether multiple instances of the program share data. Module-definition files are optional. If you use one, begin the file with the **NAME** statement. The following sample module-definition file specifies an application, `MYPROG`, that shares the `CONSTDAT` segment:

```
NAME MYPROG

SEGMENTS CONSTDAT SHARED
```

17.9 Related Topics in Online Help

In addition to information covered in this chapter, information on the following topics can be found in online help:

Topic	Access
BIND	See the "Microsoft Advisor Contents" screen
OS/2 Include files	Choose from the "MASM 6.0 Contents" screen

Topic	Access
PROTO, INVOKE	From the "MASM 6.0 Contents" screen, choose "Directives" and then "Procedure and Code Labels"
INCLUDE, INCLUDELIB	From the "MASM 6.0 Contents" screen, select "Directives" and then "Miscellaneous Language Directives"
EXEHDR	From the "Microsoft Advisor Contents" screen, select "Miscellaneous" under "Microsoft Utilities"
INCL_NOCOMMON	Select "OS/2 Include Files" from the "MASM 6.0 Contents" screen; from the next screen, select "Category Summary"
CALL	From the "MASM 6.0 Contents" screen, choose "Processor Instruction" and then "Control Flow"
SHOW.EXE	From the "MASM 6.0 Contents" screen, choose "Example Code" and then "SHOW (Text Viewer)"

Chapter 18

Creating Dynamic-Link Libraries

A "dynamic-link library" (DLL) links to the main program at run time (hence the term *dynamic link*). The program that calls the DLL is known as the "client program." One DLL can supply services for several clients simultaneously.

The client program can choose to load the DLL into memory at the same time the main program loads, or it can choose to load the DLL only when it is needed.

DLLs are available only in OS/2 and Windows. In non-Windows DOS programs, all object modules are statically linked to the program at link time. This chapter discusses DLL programming for OS/2 1.*x* only.

After an overview of DLLs, this chapter describes the following stages in developing a DLL:

- Understanding general DLL programming considerations
- Writing an interface to the DLL's exported procedures and data
- Writing initialization and termination code
- Building the DLL

The last step requires use of a module-definition file and an import library.

18.1 DLL Overview

Like a standard (object-code) library, a DLL contains procedures that one or more programs can call. Yet unlike standard-library procedures, DLL procedures are never copied into an application's executable file. They reside only on disk in the DLL file.

DLLs have several advantages:

- Dynamic link libraries save significant space since the DLL's code and data exist in only one place, no matter how many different programs call the DLL. Applications that need a particular DLL can share it.

  In contrast, a standard library routine (the **printf** function in C, for example) becomes part of the executable code for each application that uses it. For example, if three different programs use the statically linked **printf** function, three copies of the **printf** code are on disk. Furthermore, if all three programs

run at once, the **printf** code occurs three times in memory. If the same function were part of a DLL, it would exist in only one location on disk and in memory.

- Dynamic linking makes applications and libraries more independent, and therefore they are easier to maintain. You can update a DLL without having to relink any of the programs that use it.
- Applications link faster because the executable code for a dynamic link function is not copied into the application's .EXE file. Instead, only an import definition is copied.

The purpose of a DLL is to supply ("export") procedures and data to client programs at run time. Items not exported are visible only within the DLL.

Exported procedures are visible to the client program.

The concept of exporting is analogous to the action of the **PUBLIC** directive, but goes further. A public item is available only to other source modules within the same program or DLL. An exported item is available to all programs running on the system. In addition to global procedures and data, a DLL can contain other procedures and data definitions to support the operations of exported procedures.

Finally, a DLL can contain initialization and termination code to allocate and release resources needed by the procedures. Resources are typically files or dynamic memory. System services for OS/2 and Windows are provided through DLLs.

18.2 DLL Programming Requirements

Four programming requirements arise from the nature of DLLs. These requirements apply to all code used in a dynamic-link call—both in an exported procedure and in any procedure it may call:

- You cannot assume that the SS and DS registers hold the same value, unless you explicitly set SS equal to DS.
- You should avoid using the math coprocessor or emulator routines unless you are certain a coprocessor or emulator library is available.
- The DLL should be "re-entrant," because there is no guarantee that only one program will use the DLL. A re-entrant procedure is one that can be called by different programs concurrently. This creates problems for static data in the DLL, unless you declare data to be **NONSHARED** in the module-definitions file.
- Be careful how you place data and code in segments. The location of data and code in different segments and the contents of the module-definition file also determine the content of the executable file.

This section discusses these requirements.

18.2.1 Separate Stack and Data Requirement

The separate stack and data requirement involves both assembler assumptions and coding techniques. If you used the **FARSTACK** keyword as described in Section 18.3.1, "Choosing Module Attributes," the assembler makes correct assumptions about the contents of DS and SS.

Do not assume that SS equals DS.

In your own code, avoid any optimizing techniques that use SS to access items in the data segment or DS to access stack data. For example, the following code uses the **ASSUME** statement to be sure the correct stack is accessed:

```
ASSUME  DS:DGROUP
        .
        .
        .
        push    ds
        lds     si, sourcead  ; Load DS for string ops
ASSUME  DS:NOTHING
        .
        .
        .
ASSUME  SS:STACK
        mov     bx, ss:thing  ; Access near data thing through SS
ASSUME  SS:NOTHING
```

Thread-specific variables can be stored on the stack, as shown in the example above.

18.2.2 Floating-Point Math Requirement

Don't assume the math coprocessor is available to the DLL.

A stand-alone DLL—that is, a DLL created for general use by many programs—can make few assumptions about the calling program. Therefore, the safest way to perform floating-point calculations is to use alternate math routines. If you link to a Microsoft high-level language, you can access these routines through a language library. These routines give the fastest results possible without a coprocessor. See Section 6.3, "Using Emulator Libraries," for more information.

Floating-point operations in DLLs can use a coprocessor or emulator routines if you are certain that a coprocessor or emulator libraries are available.

18.2.3 Re-entrance Requirement

A procedure may be called by any number of different programs concurrently. That is, program A may call a DLL procedure while program B is still executing the same procedure. The basic problem of re-entrance is how data is shared.

Be aware that re-entering the DLL can modify its data.

For example, suppose you have a DLL that contains an accounting package; one of the functions adds up an employee's salary for a whole year. First it initializes the total to zero; then it increments this total one week at a time. While program A is in the middle of this function, program B could enter the procedure; its first action would be to initialize the total to zero. Control could then pass back to program A, which would then have zero total for salary. The problem is that two instances of the DLL share the same variable for totals.

A procedure in a DLL must therefore follow this rule: it can access static data items but must not alter them. Otherwise, one instance of a procedure could corrupt data relied on by another instance of the procedure.

There are several exceptions to this rule. First, if data is declared **NONSHARED** in the module-definitions file, each instance has its own copy of the data segment, and there is no conflict. Second, you can use semaphores to allow mutually exclusive access to data items. Finally, there may be some items you deliberately want all instances to alter—such as a global counter to keep track of number of instances.

Section 18.4.1, "Writing the Module-Definition File," explains how to declare some data items as **SHARED** while declaring others to be **NONSHARED**.

18.2.4 Segment Strategy in a DLL

Be careful how you place different kinds of data and code in different segments. When loading the DLL, OS/2 checks to see if the DLL is already in memory. If so, it loads only new copies of **NONSHARED** segments; it does not reload **SHARED** segments. Code segments are always **SHARED**.

Control of DLL data and code works at the segment level. The **DATA** statement assigns default attributes for all data segments in the DLL, but the module-definition **SEGMENTS** statement overrides these attributes for any given segment.

You may want to create a DLL that has some data shared between all programs that call the DLL and some data that is private to each instance. The following module-definition statement specifies that all data in `GLOBDAT` is shared and all data in `PRIVDAT` is not:

```
SEGMENTS
    GLOBDAT SHARED 'data'
    PRIVDAT NONSHARED 'data'
```

The segments have class 'code' unless you specifically define the class as shown in this example. See Section 18.4.1 for more information on module-definition files.

18.3 Writing the DLL Code

When you write the code for the DLL module, you need to select the correct module attributes, define the procedures and data in your DLL, and write the initialization and termination code. This section discusses these tasks.

18.3.1 Choosing Module Attributes

As noted in Chapter 2, there are four fields for the **.MODEL** directive: memory model, language type, operating system, and stack type. When you write a DLL, you can choose the attributes you would normally use for the first two fields. OS/2 system calls use the Pascal calling convention, so you may find it convenient to make all your modules use this convention as well.

DLLs use the OS_OS2 and FARSTACK attributes.

The operating system and stack fields should be **OS_OS2** and **FARSTACK**, respectively. You should use the **NEARSTACK** attribute only if you switch execution to your own stack.

A usable declaration is therefore

```
.MODEL  large, pascal, os_os2, farstack
```

If you are using full segment definitions, remember to generate an **ASSUME** directive for DS but not for SS.

```
ASSUME  DS:DGROUP ; Necessary with full segment definitions
```

18.3.2 Defining Procedures and Data

Procedures and data in DLLs can be either global (available to the client process) or local (used only by the DLL). To create a global data item, make sure that it is public:

```
EXTERNDEF   dllvar
            .DATA
dllvar      WORD    0
```

The variable must then be exported in a module-definition file, as shown in Section 18.4.1, "Writing the Module-Definition File." When executable files other than the DLL access the variable, they must treat it as far data, as in the following example:

```
mov     ax, SEG dllvar
mov     es, ax
mov     bx, es:dllvar
```

An exported procedure (often called a dynamic-link procedure) must follow these rules:

- It must be declared far and public. The MASM keyword **EXPORT** does both of these.
- The procedure should initialize DS upon entry (unless you are not going to be accessing any static near data).
- Data pointers in the parameter list should be far.

The easiest way to realize most of these requirements is to use the **EXPORT** keyword and **LOADDS** in the procedure's *prologuearg* list (see Section 7.3.8). **LOADDS** generates instructions to save DS and load it with the value of the DLL's data segment. The **EXPORT** keyword makes the procedure **FAR** and **PUBLIC**, overriding the memory model. You may also need to use **FORCEFRAME**, which instructs the assembler to generate a stack frame even if there are no parameters or locals.

The example DLL used in the chapter, CSTR.DLL, illustrates how DLLs can be shared by several processes. The procedures in the DLL write a string and keep track of the number of times the string is written. When more than one process uses the DLL, they all increment the global variable `GCount`, but each process increments its own private instance of the `PCount` variable.

The only initialization code this DLL needs is code to set up the exit code. The next section shows how to write a module-definition file to create an import library and how to create a DLL from this code.

The code for the CSTR.DLL example looks like this:

```
        .MODEL  small, pascal, os_os2, farstack
        .286

INCL_NOCOMMON   EQU     1
INCL_DOSPROCESS EQU     1
INCL_VIO        EQU     1

        INCLUDE    OS2.INC
        INCLUDELIB OS2.LIB

        .DOSSEG

VioWrtCStr PROTO FAR PASCAL, pchString:PCH, hv:HVIO
GetGCount  PROTO     PASCAL
GetPCount  PROTO     PASCAL
CStrExit   PROTO FAR

        .STACK
        .DATA               ; Default segment is SHARED
```

```
GCount  WORD    0           ; Count of all calls

@CurSeg ENDS

PRIVDAT SEGMENT WORD        ; Private segment is NONSHARED

PCount  WORD    0           ; Count of all this process
                            ;  calls to VioWrtCStr
PRIVDAT ENDS

        .CODE
        .STARTUP

           pusha

        ; Initialization goes here. In this case, the only
        ;  initialization is setting up the exit behavior.

        INVOKE  DosExitList, EXLST_ADD, CStrExit
        INVOKE  DosExitList, EXLST_EXIT,0

        popa
        retf

VioWrtCStr PROC FAR PASCAL EXPORT <LOADDS> USES cx di si,
        pchString:PCH,
        hv:HVIO

        sub     al, al          ; Search for zero
        mov     cx, 0FFFFh      ; Set maximum length
        les     di, pchString   ; Load pointer
        mov     si, di          ; Copy it
        repne   scasb           ; Find null
        .IF     zero?           ; Continue if found
        sub     di, si          ; Calculate length
        xchg    di, si          ; Restore address and save length

        INVOKE  VioWrtTTy,      ; Let OS/2 do output
                es:di,          ; Address of string
                si,             ; Calculated length
                hv              ; Video handle

        inc     GCount          ; Count as one of total calls
```

```
        ASSUME  DS:PRIVDAT
        mov     ax, PRIVDAT
        mov     ds, ax
        inc     PCount          ; Count as one of process calls
        ASSUME  DS:DGROUP
        sub     ax, ax          ; Success
        .ELSE
        mov     ax, 1           ; Error
        .ENDIF
        ret
VioWrtCStr ENDP

GetGCount PROC  FAR PASCAL EXPORT <LOADDS, FORCEFRAME>
        mov     ax, GCount
        ret
GetGCount ENDP

GetPCount PROC  FAR PASCAL EXPORT <LOADDS, FORCEFRAME> USES ds
        ASSUME  DS:PRIVDAT
        mov     ax, PRIVDAT
        mov     ds, ax
        mov     ax, PCount
        ASSUME  DS:NOTHING
        ret
GetPCount ENDP

        .DATA
szOut   BYTE    13, 10, "Exiting DLL...", 13, 10, 0
        .CODE

CStrExit PROC   FAR  <LOADDS, FORCEFRAME>
        INVOKE  VioWrtCStr,
                ADDR szOut,
                0
        INVOKE  DosExitList, EXLST_EXIT, 0
CStrExit ENDP

        END
```

These generated code for the `VIOWrtCStr` procedure follows. The code marked with asterisks is generated by the assembler.

```
VioWrtCStr PROC FAR PASCAL EXPORT <LOADDS> USES cx di si,
        pchString:PCH,
        hv:HVIO
 0000  55         *          push   bp
 0001  8B EC      *          mov    bp, sp
 0003  1E         *          push   ds
 0004  B8 ---- R  *          mov    ax, DGROUP
 0007  8E D8      *          mov    ds, ax
 0009  51         *          push   cx
 000A  57         *          push   di
 000B  56         *          push   si
        .
        .                        ; Procedure code here
        .
                                 ret
 000C  5E         *          pop    si
 000D  5F         *          pop    di
 000E  59         *          pop    cx
 000F  1F         *          pop    ds
 0010  C9         *          leave
 0011  CA 0006    *          ret    00006h
 0014                  VioWrtCStr ENDP
```

The DLL should establish its own data segment.

The DLL should change DS in this manner because each client program has its own private version of DGROUP. When a program calls your dynamic-link procedure, DS points to the program's data area, not yours. The solution is to initialize DS so that it points to your own default data area.

However, one side effect of this approach is that it alters DS so that it no longer is equal to SS. Consequently, all data pointers in the parameter list must be far pointers, even if the data was stack data or near data.

18.3.3 Creating Initialization and Termination Code

Begin initialization code with the .STARTUP directive.

A DLL can contain procedures that require special resources, such as temporary files or dynamic memory blocks. Resources allocated during initialization exist for the lifetime of the client program and are removed when the client program exits. Usually the best method for managing these resources is to write initialization and termination code.

A DLL can have a starting point just as an application does. In the case of a DLL, this starting point marks the beginning of the initialization code. A DLL does not need a starting point if it has no need for initialization. Do not use **.EXIT**, since **.EXIT** will terminate the client program.

Attributes of the initialization code are defined in the module-definition file (see Section 18.4.1). Initialization code can have the **INITGLOBAL** or **INITINSTANCE** attribute.

INITGLOBAL specifies that the initialization code executes only once—when the DLL is first loaded into memory. **INITINSTANCE** specifies that initialization code should execute once for each program that uses the DLL. **INITGLOBAL** is the default. You should use termination code only for DLLs that have been defined with **INITINSTANCE** unless you know that the first process to use the DLL is the last to terminate.

To specify **INITINSTANCE**, place the **LIBRARY** statement in your module-definition file:

```
LIBRARY CSTR INITINSTANCE
```

In the statement above, CSTR is the name of the DLL.

To include a termination procedure, invoke **DosExitList** in the initialization code. **DosExitList** is a system function that attaches a termination procedure to a program. When the program terminates, OS/2 executes the procedure as part of the program exit sequence. In the termination procedure itself, release any system resources (such as memory or files) allocated during initialization.

This is the termination code for the CSTR.DLL module:

```
CStrExit PROC   FAR  <LOADDS, FORCEFRAME>
        INVOKE  VioWrtCStr,
                ADDR szOut,
                0
        INVOKE  DosExitList, EXLST_EXIT, 0
CStrExit ENDP
```

The termination code in CSTR.DLL uses the **INVOKE** directive to set up a call to the **DosExitList** function. You can perform a similar operation by simply pushing arguments on the stack and observing the correct calling convention.

The effect of **DosExitList** in the initialization code is to make OS/2 call the termination procedure when the current process exits. The "current process" in this case is the client program, not the DLL or the DLL initialization code.

18.4 Building the DLL

To create a DLL, you need to assemble the DLL code, write a module-definition file, use LINK to create the DLL, generate an import library, and then link the DLL to the client program.

18.4.1 Writing the Module-Definition File

A module-definition file is required for DLLs.

The module-definition file is an ASCII text file that lists attributes of a library or application (in the case of an application, this file is optional). The module-definition file gives directions to the linker that supplement the information on the command line.

This module-definition file tells the linker to create a DLL called CSTR.DLL with **INITINSTANCE** data. The library has exported procedure `VioWrtCStr`, `GetPCount`, and `GetGCount`, and the data segment `PRIVDAT` is not shared between programs:

```
LIBRARY CSTR INITINSTANCE

EXPORTS
    VioWrtCstr
    GetGCount
    GetPCount

DATA SINGLE NONSHARED
```

The **LIBRARY** statement need not specify a name. If the name is omitted, the linker gives the library the base filename of the module-definition file. The default file extension is .DLL. The **INITINSTANCE** attribute is optional and is significant only if you have initialization code. If you specify **INITINSTANCE**, then the library initialization is called each time a new process gains access to the library. Otherwise, it will be called once only.

At least one procedure must be listed after EXPORTS.

The **EXPORTS** statement lists identifiers (procedures and variables) that can be accessed directly by client programs. Note that if you give a procedure the **EXPORTS** attribute from within the source code, you do not need to list the procedure here. The **EXPORTS** keyword automatically exports the procedure by name, so putting the names of the procedures in the module-definition file is not required. However, exported variables must be listed in a module-definition file.

The **DATA** statement lists attributes for data segments (DGROUP) in the DLL. The default for DLLs, **SINGLE**, specifies that one DGROUP is shared by all instances of the DLL. **NONSHARED** specifies that all other data segments are not to be shared. See Section 13.15, "CODE, DATA, and SEGMENTS Attributes."

18.4.2 Generating an Import Library with IMPLIB

The DLL exports a procedure; the client program imports it.

Just as a procedure is exported by a DLL, it must be imported by an application. An application's EXE header must indicate what dynamic-link procedures are used and where they reside. The easiest way to specify this information is with an "import library," which is a .LIB file that contains the import information in object-record form. The IMPLIB utility automates this process for you.

To create an import library, run the IMPLIB utility on the module-definition file:

```
IMPLIB MYDYNLIB.LIB MYDYNLIB.DEF
```

The result is the import library, MYDYNLIB.LIB, which you then link to any program that calls CSTR.DLL. You would then list MYDYNLIB.LIB in the libraries field (the fourth field) of the LINK command. Or, in assembly-language programs, you can link to this library automatically by just adding the following statement to the source code of your program:

```
INCLUDELIB MYDYNLIB.LIB
```

18.4.3 Creating and Using the DLL

Now you can use LINK to create the DLL. The LINK utility uses the object module of the DLL code and the module definition to create the CSTR.DLL:

```
LINK CSTR.OBJ , , , , MYDYNLIB.DEF
```

If linking is successful, the linker creates a file with a .DLL extension.

You can link several modules together to create a DLL. The following command line links several object modules and an object-code library (BIGLIB.LIB) to form a DLL. The module-definition file is MYDYNLIB.DEF:

```
LINK  MOD1 MOD2 MOD3,,, BIGLIB, MYDYNLIB
```

To use the DLL, copy the .DLL file to a directory listed in the LIBPATH setting in your CONFIG.SYS file.

To create an executable file using the DLL, link the client program with the import library as shown:

```
LINK CALLDLL.OBJ , , , MYDYNLIB.LIB
```

By running CALLDLL.EXE in separate OS/2 windows, you can see that both client programs access the DDL at the same time. When the last process exits the DLL, the DLL is removed from memory.

18.5 Related Topics in Online Help

In addition to information covered in this chapter, information on the following topics can be found in online help.

Topic	Access
LINK	From the "Microsoft Advisor Contents" screen, select LINK
Module-definition files	Select Module-Definition Files from the "LINK Contents" screen
EXPORT	Select from the MASM Language Index
EXTERNDEF	From the "MASM 6.0 Contents" screen, select "Directives"; then select "Scope and Visibility" from the next screen
LOADDS, FORCEFRAME	Choose "Proc" from the MASM Language Index
IMPLIB	Select "IMPLIB Summary" from the "LINK Contents" screen

Chapter 19

Writing Memory-Resident Software

Through its memory-management system, DOS allows a program to remain resident in memory after terminating. The resident program can later regain control of the processor to perform tasks such as background printing or "popping up" a calculator on the screen. Such a program is commonly called a TSR, from the Terminate-and-Stay-Resident function it uses to return to DOS.

This chapter explains the techniques of writing memory-resident software. The first two sections present introductory material. Following sections describe important DOS and BIOS interrupts and focus on how to write safe, compatible, memory-resident software. Two example programs illustrate the techniques described in the chapter. These programs are also available as sample programs on the MASM 6.0 disks.

19.1 Terminate-and-Stay-Resident Programs

DOS maintains a pointer to the beginning of unused memory. Programs load into memory at this position. They terminate execution by returning control to DOS. Normally, the pointer remains unchanged, allowing DOS to reuse memory when loading other programs.

A terminating program can, however, prevent other programs from loading on top of it. It does this by returning to DOS through the terminate-and-stay-resident function, which resets the free-memory pointer to a higher position. This leaves the program resident in a protected block of memory, even though it is no longer running.

The terminate-and-stay-resident function (Function 31h) is one of the DOS services invoked through Interrupt 21h. The following fragment shows how a TSR program terminates using Function 31h and remains resident in a 1000h-byte block of memory:

```
mov     ah, 31h         ; Request DOS Function 31h
mov     al, err         ; Set return code
mov     dx, 100h        ; Reserve 100h paragraphs
                        ;  (1000h bytes)
int     21h             ; Terminate-and-stay-resident
```

NOTE In current versions of DOS, Interrupt 27h also provides a terminate-and-stay-resident service. However, Microsoft cannot guarantee future support for Interrupt 27h and does not recommend its use.

19.1.1 Structure of a TSR

TSRs consist of two distinct parts that execute at different times. The first part is the installation section, which executes only once, when DOS loads the program. The installation code performs any initialization tasks required by the TSR and then exits through the terminate-and-stay-resident function.

A TSR consists of an installation section and a resident section.

The second part of the TSR, called the resident section, consists of code and data left in memory after termination. Though often identified with the TSR itself, the resident section makes up only part of the entire program.

The TSR's resident code must be able to regain control of the processor and execute after the program has terminated. Methods of executing a TSR are classified as either passive or active.

19.1.2 Passive TSRs

The simplest way to execute a TSR is to transfer control to it explicitly from another program. Because the TSR in this case does not solicit processor control, it is said to be passive. If the calling program can determine the TSR's memory address, it can grant control via a far jump or call. More commonly, a program activates a passive TSR through a software interrupt. The installation section of the TSR writes the address of its resident code to the proper position in the interrupt vector table (see Section 7.4, "DOS Interrupts"). Any subsequent program can then execute the TSR by calling the interrupt.

Passive TSRs often replace existing software interrupts. For example, a passive TSR might replace Interrupt 10h, the BIOS video service. By intercepting calls that read or write to the screen, the TSR can access the video buffer directly, increasing display speed.

Passive TSRs allow limited access since they can be invoked only from another program. They have the advantage of executing within the context of the calling program, and thus run no risk of interfering with another process, which could happen with active TSRs.

19.1.3 Active TSRs

The second method of executing a TSR involves signaling it through some hardware event, such as a predetermined sequence of keystrokes. This type of TSR is called active because it must continually search for its start-up signal. The advantage of active TSRs lies in their accessibility. They can take control from any running application, execute, and return, all on demand.

An active TSR, however, must not seize processor control blindly. It must contain additional code that determines the proper moment at which to execute. The extra code consists of one or more routines called "interrupt handlers," described in the following section.

19.2 Interrupt Handlers in Active TSRs

The memory-resident portion of an active TSR consists of two parts. One part contains the body of the TSR—the code and data that perform the program's main tasks. The other part contains the TSR's interrupt handlers.

An interrupt handler is a routine that takes control when a specific interrupt occurs. Although sometimes called an "interrupt service routine," a TSR's handler usually does not service the interrupt. Instead, it passes control to the original interrupt routine, which does the actual interrupt servicing.

Collectively, interrupt handlers ensure that a TSR operates compatibly with the rest of the system. Individually, each handler fulfills at least one of the following functions:

- Auditing hardware events that may signal a request for the TSR
- Monitoring system status
- Determining whether a request for the TSR should be honored, based on current system status

19.2.1 Auditing Hardware Events for TSR Requests

Active TSRs commonly use a special keystroke sequence or the timer as a request signal. A TSR invoked through one of these channels must be equipped with handlers that audit keyboard or timer events.

A keyboard handler receives control at every keystroke. It examines each key, searching for the proper signal or "hot key." Generally, a keyboard handler should not attempt to call the TSR directly when it detects the hot key. If the TSR cannot safely interrupt the current process at that moment, the keyboard handler is forced to exit to allow the process to continue. Since the handler cannot regain control until the next keystroke, the user has to press the hot key repeatedly until the handler can comply with the request.

Instead, the handler should merely set a request flag when it detects a hot-key signal and then exit normally. Examples in the following paragraphs illustrate this technique.

For computers other than the IBM PS/2® series, an active TSR audits keystrokes through a handler for Interrupt 09, the keyboard interrupt:

```
Keybrd  PROC    FAR
        sti                     ; Interrupts are okay
        push    ax              ; Save AX register
        in      al, 60h         ; AL = scan code of current key
        call    CheckHotKey     ; Check for hot key
        .IF     !carry?         ; If not hot key:
```

```
; Hot key pressed. Reset the keyboard to throw away keystroke.
        cli                     ; Disable interrupts while resetting
        in      al, 61h         ; Get current port 61h state
        or      al, 10000000y   ; Turn on bit 7 to signal clear keybrd
        out     61h, al         ; Send to port
        and     al, 01111111y   ; Turn off bit 7 to signal break
        out     61h, al         ; Send to port
        mov     al, 20h         ; Reset interrupt controller
        out     20h, al
        sti                     ; Reenable interrupts
        pop     ax              ; Recover AX
        mov     cs:TsrRequestFlag, TRUE ; Raise request flag
        iret                    ; Exit interrupt handler
        .ENDIF                  ; End hot-key check

; No hot key was pressed, so let normal Int 09 service
; routine take over

        pop     ax              ; Recover AX and fall through
        cli                     ; Interrupts cleared for service
KeybrdMonitor LABEL FAR         ; Installed as Int 09 handler for
                                ;  PS/2 or for time-activated TSR
                                ; Signal that interrupt is busy
        mov     cs:intKeybrd.Flag, TRUE
        pushf                   ; Simulate interrupt by pushing flags,
                                ;  far-calling old Int 09 routine
        call    cs:intKeybrd.OldHand
        mov     cs:intKeybrd.Flag, FALSE
        iret
Keybrd  ENDP
```

A TSR running on a PS/2 computer cannot reliably read key-scan codes using the above method. Instead, the TSR must search for its hot key through a handler for Interrupt 15h (Miscellaneous System Services). The handler determines the current keypress from the AL register when AH equals 4Fh, as shown here:

```
MiscServ PROC   FAR
        sti                     ; Interrupts okay
        .IF     ah == 4Fh       ; If Keyboard Intercept Service:
        push    ax              ; Preserve AX
        call    CheckHotKey     ; Check for hot key
        pop     ax
        .IF     !carry?         ; If hot key:
        mov     cs:TsrRequestFlag, TRUE ; Raise request flag
        clc                     ; Signal BIOS not to process the key
        ret     2               ; Simulate IRET without popping flags
        .ENDIF                  ; End carry flag check
        .ENDIF                  ; End Keyboard Intercept check
        cli                     ; Disable interrupts and fall through
SkipMiscServ LABEL FAR          ; Interrupt 15h handler if PC/AT
        jmp     cs:intMisc.OldHand
MiscServ ENDP
```

The example program in Section 19.8 demonstrates how a TSR tests for a PS/2 machine and then sets up a handler for either Interrupt 09 or Interrupt 15h to audit keystrokes.

Setting a request flag in the keyboard handler allows other code, such as the timer handler (Interrupt 08), to recognize a request for the TSR. The timer handler gains control at every timer interrupt; the interrupts occur an average of 18.2 times per second. The following fragment shows how a timer handler tests the request flag and continually polls until it can safely execute the TSR.

```
TestFlag PROC FAR
        .
        .
        .
        cmp     TsrRequestFlag, FALSE ; Has TSR been requested?
        je      exit                  ; If not, exit
        call    CheckSystem           ; Can system be interrupted
                                      ;  safely?
        jc      exit                  ; If not, exit
        call    ActivateTsr           ; If okay, call TSR
```

Figure 19.1 illustrates the process. It shows a time line for a typical TSR signaled from the keyboard. When the keyboard handler detects the proper hot key, it sets a request flag called `TsrRequestFlag`. Thereafter, the timer handler continually checks the system status until it can safely call the TSR.

The timer itself can serve as the start-up signal if the TSR executes periodically. Screen clocks that continuously show seconds and minutes are examples of TSRs that use the timer this way. ALARM.ASM, a program described in the next section, shows another example of a timer-driven TSR.

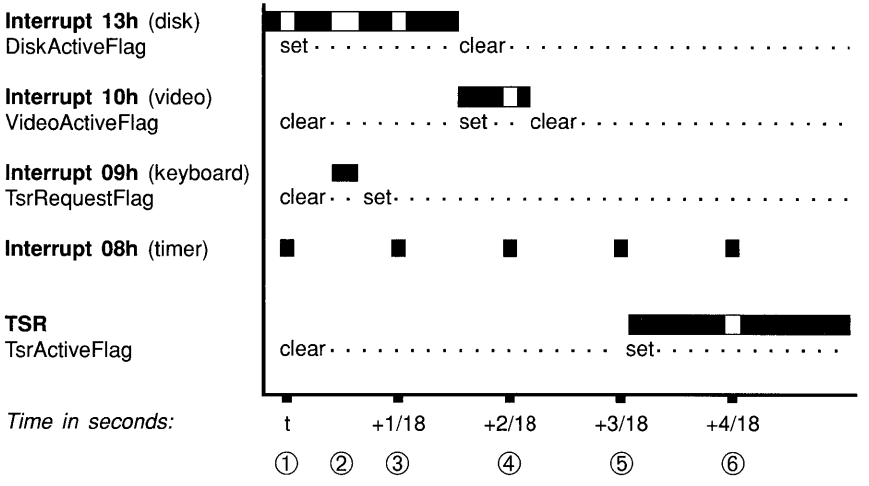


Figure 19.1 Time Line of Interactions between Interrupt Handlers for a Typical TSR

19.2.2 Monitoring System Status

A TSR that uses a hardware device such as the video or disk must not interrupt while the device is active. A TSR monitors a device by handling the device's interrupt. Each interrupt handler need only set a flag to indicate that the device is in use and then clear the flag when the interrupt finishes.

The following shows a typical monitor handler:

```
NewHandler PROC    FAR
        mov     ActiveFlag, TRUE  ; Set active flag
        pushf                     ; Simulate interrupt by
                                  ;  pushing flags,
        call    OldHandler        ;  then calling original routine
        mov     ActiveFlag, FALSE ; Clear active flag
        iret                      ; Return from interrupt
NewHandler ENDP
```

Only hardware used by the TSR requires monitoring. For example, a TSR that performs disk input/output (I/O) must monitor disk use through Interrupt 13h. The disk handler sets an active flag that prevents the TSR from executing during a read or write operation. Otherwise, the TSR's own I/O would move the disk head. This would cause the suspended disk operation to continue with the head incorrectly positioned when the TSR returned control to the interrupted program.

In the same way, an active TSR that displays to the screen must monitor calls to Interrupt 10h. The Interrupt 10h BIOS routine does not protect critical sections of code that program the video controller. The TSR must therefore ensure that it does not interrupt such nonreentrant operations.

The activities of the operating system also affect the system status. With few exceptions, DOS functions are not reentrant and must not be interrupted. However, monitoring DOS is somewhat more complicated than monitoring hardware. Discussion of this subject is deferred until Section 19.4.

The following comments describe the chain of events depicted in Figure 19.1. Each comment refers to one of the numbered pointers in the figure.

1. At time = t, the timer handler activates. It finds the flag `TsrRequestFlag` clear, indicating that the TSR has not been requested. The handler terminates without taking further action. Notice that Interrupt 13h is currently processing a disk I/O operation.

2. Before the next timer interrupt, the keyboard handler detects the hot key, signalling a request for the TSR. The handler sets `TsrRequestFlag` and returns.

3. At time = t + 1/18 second, the timer handler again activates and finds `TsrRequestFlag` set. The handler checks other active flags to determine if the TSR can safely execute. Since Interrupt 13h has not yet completed its

disk operation, the timer handler finds `DiskActiveFlag` set. The handler therefore terminates without invoking the TSR.

4. At time = t + 2/18 second, the timer handler again finds `TsrRequestFlag` set and repeats its scan of the active flags. `DiskActiveFlag` is now clear, but in the interim, Interrupt 10h has activated as indicated by the flag `VideoActiveFlag`. The timer handler accordingly terminates without invoking the TSR.
5. At time = t + 3/18 second, the timer handler repeats the process. This time it finds all active flags clear, indicating that the TSR may safely execute. The timer handler calls the TSR, which sets its own active flag to ensure that it will not interrupt itself if requested again.
6. The timer and other interrupts continue to function normally while the TSR executes.

19.2.3 Determining Whether to Invoke the TSR

Once a handler receives a request signal for the TSR, it checks the various active flags maintained by the handlers that monitor system status. If any of the flags are set, the handler ignores the request and exits. If the flags are clear, the handler invokes the TSR, usually through a near or far call. Figure 19.1 illustrates how a timer handler detects a request and then periodically scans various active flags until all the flags are clear.

A TSR that changes stacks must not interrupt itself. Otherwise, the second execution would overwrite the stack data belonging to the first. A TSR prevents this by setting its own active flag before executing, as shown in Figure 19.1. A handler must check this flag along with the other active flags when determining whether the TSR can safely execute.

19.3 Example of a Simple TSR: ALARM

This section presents a simple alarm clock TSR that demonstrates some of the material covered so far. The program accepts an argument from the command line that specifies the alarm setting in military form, such as 1635 for 4:35 P.M. For the sake of simplicity, the argument must consist of four digits, including leading zeros. To set the alarm at 7:45 A.M., for example, enter:

```
ALARM 0745
```

The installation section of the program begins with the `Install` procedure. `Install` computes the number of five-second intervals that must elapse before the alarm sounds and stores this number in the word `CountDown`. The procedure then obtains the vector for Interrupt 08 (timer) through Interrupt 21h Function 35h and stores it in the far pointer `OldTimer`. Interrupt 21h Function 25h

replaces the vector with the far address of the new timer handler `NewTimer`. Once installed, the new timer handler executes at every timer interrupt. These interrupts occur 18.2 times per second or 91 times every five seconds.

Each time it executes, `NewTimer` subtracts one from a secondary counter called `Tick91`. By counting 91 timer ticks, `Tick91` accurately measures a period of five seconds. When `Tick91` reaches zero, it's reset to 91 and `CountDown` is decremented by one. When `CountDown` reaches zero, the alarm sounds.

```
;* ALARM.ASM - A simple memory-resident program that beeps the speaker
;* at a prearranged time. Can be loaded more than once for multiple
;* alarm settings. During installation, ALARM establishes a handler
;* for the timer interrupt (interrupt 08). It then terminates through
;* the terminate-and-stay-resident function (function 31h). After the
;* alarm sounds, the resident portion of the program retires by setting
;* a flag that prevents further processing in the handler.
;*
;* NOTE: You must assemble this program as a .COM file, either as a PWB
;*       build option or with the ML /AT option.

        .MODEL tiny, pascal, os_dos
        .STACK
        .CODE
        ORG     5Dh                     ; Location of time argument in PSP,
CountDown       LABEL   WORD            ;  converted to number of 5-second
                                        ;  intervals to elapse
        .STARTUP
        jmp     Install                 ; Jump over data and resident code

; Data must be in code segment so it won't be thrown away with Install code.

OldTimer        DWORD   ?               ; Address of original timer routine
tick_91         BYTE    91              ; Counts 91 clock ticks (5 seconds)
TimerActiveFlag BYTE    0               ; Active flag for timer handler

;* NewTimer - Handler routine for timer interrupt (interrupt 08).
;* Decrements CountDown every 5 seconds. No other action is taken
;* until CountDown reaches 0, at which time the speaker sounds.

NewTimer PROC   FAR
        .IF     cs:TimerActiveFlag != 0 ; If timer busy or retired:
        jmp     cs:OldTimer             ; Jump to original timer routine
        .ENDIF
        inc     cs:TimerActiveFlag      ; Set active flag
        pushf                           ; Simulate interrupt by pushing flags,
        call    cs:OldTimer             ;  then far-calling original routine
        sti                             ; Enable interrupts
        push    ds                      ; Preserve DS register
        push    cs                      ; Point DS to current segment for
        pop     ds                      ;  further memory access
        dec     tick_91                 ; Count down for 91 ticks
```

```
        .IF     zero?                   ; If 91 ticks have elapsed:
        mov     tick_91, 91             ; Reset secondary counter and
        dec     CountDown               ;  subtract one 5-second interval
        .IF     zero?                   ; If CountDown drained:
        call    Sound                   ; Sound speaker
        inc     TimerActiveFlag         ; Alarm has sounded, set flag
        .ENDIF
        .ENDIF

        dec     TimerActiveFlag         ; Decrement active flag
        pop     ds                      ; Recover DS
        iret                            ; Return from interrupt handler
NewTimer ENDP

;* Sound - Sounds speaker with the following tone and duration:

BEEP_TONE       EQU     440             ; Beep tone in hertz
BEEP_DURATION   EQU     6               ; Number of clocks during beep,
                                        ;  where 18 clocks = approx 1 second

Sound   PROC    USES ax bx cx dx es     ; Save registers used in this routine
        mov     al, 0B6h                ; Initialize channel 2 of
        out     43h, al                 ;  timer chip
        mov     dx, 12h                 ; Divide 1,193,180 hertz
        mov     ax, 34DCh               ;  (clock frequency) by
        mov     bx, BEEP_TONE           ;  desired frequency
        div     bx                      ; Result is timer clock count
        out     42h, al                 ; Low byte of count to timer
        mov     al, ah
        out     42h, al                 ; High byte of count to timer
        in      al, 61h                 ; Read value from port 61h
        or      al, 3                   ; Set first two bits
        out     61h, al                 ; Turn speaker on

; Pause for specified number of clock ticks

        mov     dx, BEEP_DURATION       ; Beep duration in clock ticks
        sub     cx, cx                  ; CX:DX = tick count for pause
        mov     es, cx                  ; Point ES to low memory data
        add     dx, es:[46Ch]           ; Add current tick count to CX:DX
        adc     cx, es:[46Eh]           ; Result is target count in CX:DX
        .REPEAT
        mov     bx, es:[46Ch]           ; Now repeatedly poll clock
        mov     ax, es:[46Eh]           ;  count until the target
        sub     bx, dx                  ;  time is reached
        sbb     ax, cx
        .UNTIL  !carry?

        in      al, 61h                 ; When time elapses, get port value
        xor     al, 3                   ; Kill bits 0-1 to turn
        out     61h, al                 ;  speaker off
        ret
Sound   ENDP
```

```
;* Install - Converts ASCII argument to valid binary number, replaces
;* NewTimer as the interrupt handler for the timer, then makes program
;* memory-resident by exiting through function 31h.
;*
;* This procedure marks the end of the TSR's resident section and the
;* beginning of the installation section. When ALARM terminates through
;* function 31h, the above code and data remain resident in memory. The
;* memory occupied by the following code is returned to DOS.

Install PROC

; Time argument is in hhmm military format. Converts ASCII digits to
; number of minutes since midnight, then converts current time to number
; of minutes since midnight. Difference is number of minutes to elapse
; until alarm sounds. Converts to seconds-to-elapse, divides by 5 seconds,
; and stores result in word CountDown.

DEFAULT_TIME    EQU     3600                ; Default alarm setting = 1 hour
                                            ;  (in seconds) from present time
        mov     ax, DEFAULT_TIME
        cwd                                 ; DX:AX = default time in seconds
        .IF     BYTE PTR CountDown != ' ' ; If not blank argument:
        xor     CountDown[0], '00'          ; Convert 4 bytes of ASCII
        xor     CountDown[2], '00'          ;  argument to binary

        mov     al, 10                      ; Multiply 1st hour digit by 10
        mul     BYTE PTR CountDown[0]       ;  and add to 2nd hour digit
        add     al, BYTE PTR CountDown[1]
        mov     bh, al                      ; BH = hour for alarm to go off
        mov     al, 10                      ; Repeat procedure for minutes
        mul     BYTE PTR CountDown[2]       ; Multiply 1st minute digit by 10
        add     al, BYTE PTR CountDown[3] ; and add to 2nd minute digit
        mov     bl, al                      ; BL = minute for alarm to go off
        mov     ah, 2Ch                     ; Request function 2Ch
        int     21h                         ; Get Time (CX = current hour/min)
        mov     dl, dh
        sub     dh, dh
        push    dx                          ; Save DX = current seconds

        mov     al, 60                      ; Multiply current hour by 60
        mul     ch                          ;  to convert to minutes
        sub     ch, ch
        add     cx, ax                      ; Add current minutes to result
                                            ; CX = minutes since midnight
        mov     al, 60                      ; Multiply alarm hour by 60
        mul     bh                          ;  to convert to minutes
        sub     bh, bh
        add     ax, bx                      ; AX = number of minutes since
                                            ;  midnight for alarm setting
        sub     ax, cx                      ; AX = time in minutes to elapse
                                            ;  before alarm sounds
```

```
        .IF     carry?                  ; If alarm time is tomorrow:
        add     ax, 24 * 60             ; Add minutes in a day
        .ENDIF

        mov     bx, 60
        mul     bx                      ; DX:AX = minutes-to-elapse-times-60
        pop     bx                      ; Recover current seconds
        sub     ax, bx                  ; DX:AX = seconds to elapse before
        sbb     dx, 0                   ;  alarm activates
        .IF     carry?                  ; If negative:
        mov     ax, 5                   ; Assume 5 seconds
        cwd
        .ENDIF
        .ENDIF

        mov     bx, 5                   ; Divide result by 5 seconds
        div     bx                      ; AX = number of 5-second intervals
        mov     CountDown, ax           ;  to elapse before alarm sounds

        mov     ax, 3508h               ; Request function 35h
        int     21h                     ; Get Vector for timer (interrupt 08)
        mov     WORD PTR OldTimer[0], bx ; Store address of original
        mov     WORD PTR OldTimer[2], es ;  timer interrupt
        mov     ax, 2508h               ; Request function 25h
        mov     dx, OFFSET NewTimer     ; DS:DX points to new timer handler
        int     21h                     ; Set Vector with address of NewTimer

        mov     dx, OFFSET Install      ; DX = bytes in resident section
        mov     cl, 4
        shr     dx, cl                  ; Convert to number of paragraphs
        inc     dx                      ;  plus one
        mov     ax, 3100h               ; Request function 31h, error code=0
        int     21h                     ; Terminate-and-stay-resident
Install ENDP
        END
```

Note the following points about ALARM:

- The constant `BEEP_TONE` specifies the alarm tone. Practical values for the tone range from approximately 100 to 4,000 hertz.
- The `Install` procedure marks the beginning of the installation section of the program. Execution begins here when ALARM.COM is loaded. A TSR generally places its installation code after the resident section. This allows the TSR to include the installation code and data and to return memory to DOS when the program terminates. Since the installation section executes only once, the TSR can discard it after becoming resident.
- You can install ALARM any number of times in quick succession, each time with a new alarm setting. The timer handler does not restore the original timer vector after the alarm sounds. In effect, the multiple installations are

daisy-chained in memory. The address in `OldTimer` for one installation is the address of `NewTimer` in the preceding installation.

- Until a system reboot, `NewTimer` remains in place as the Interrupt 08 handler, even after the alarm sounds. To save unnecessary activity, the byte `TimerActiveFlag` remains set after the alarm sounds. This forces an immediate jump to the original handler for all subsequent executions of `NewTimer`.
- `NewTimer` and `Sound` alter registers DS, AX, BX, CX, DX, and ES. To preserve the original values in these registers, the procedures first push them onto the stack and then restore the original values before exiting. This ensures that the process interrupted by `NewTimer` continues with valid registers after `NewTimer` returns.
- ALARM requires little stack space. It assumes that the current stack is adequate and makes no attempt to set up a new one. More sophisticated TSRs, however, should as a matter of course provide their own stacks to ensure adequate stack depth. The example program presented in Section 19.8 demonstrates this safety measure.

19.4 Using DOS in Active TSRs

This section explains how to write active TSRs that can safely call DOS functions. The material explores the problems imposed by DOS's nonreentrance and explains how a TSR can resolve those problems. The solution consists of four parts:

- Understanding how DOS uses stacks
- Determining when DOS is active
- Determining whether a TSR can safely interrupt an active DOS function
- Monitoring the Critical Error flag

19.4.1 Understanding DOS Stacks

DOS functions set up their own stacks, which makes them nonreentrant. If a TSR interrupts a DOS function and then executes another function that sets up the same stack, the second function will overwrite everything placed on the stack by the first function. The problem occurs when the second function returns and the first is left with unusable stack data. A TSR that calls a DOS function must not interrupt any function that uses the same stack.

With few exceptions, DOS functions use their own stacks when they execute.

DOS versions 2.0 and later use three internal stacks: an I/O stack, a disk stack, and an auxiliary stack. The current stack depends on the DOS function. Functions 01 through 0Ch set up the I/O stack. Functions higher than 0Ch (with few exceptions) use the disk stack, as do Interrupts 25h and 26h. DOS normally uses the auxiliary stack only when it executes Interrupt 24h (Critical Error Handler).

19.4.2 Determining DOS Activity

A TSR's handlers can determine when DOS is active by consulting a one-byte flag called the InDos flag. Every DOS function sets this flag upon entry and clears it upon termination. During installation, a TSR locates the flag through Function 34h (Get Address of InDos Flag), which returns the address as ES:BX. The installation portion then stores the address so that the handlers can later find the flag without again calling Function 34h.

Theoretically, a TSR can wait to execute until the InDos flag is clear, thus sidestepping the entire issue of interrupting DOS. However, several low-order functions—such as Function 0Ah (Get Buffered Keyboard Input)—wait idly for an expected keystroke before they terminate. If a TSR were allowed to execute only after DOS returned, it too would be forced to wait for the terminating event.

The solution lies in determining when the low-order functions are active. DOS provides another service for this purpose: Interrupt 28h, the Idle Interrupt.

19.4.3 Interrupting DOS Functions

DOS continually calls Interrupt 28h from the low-order polling functions as they wait for keyboard input. This signal says that DOS is idle and that a TSR may interrupt provided it does not overwrite the I/O stack.

A TSR may interrupt DOS Functions 01 through 0Ch provided it does not call them.

An active TSR that calls DOS must monitor Interrupt 28h with a handler. When the handler gains control, it checks the TSR request flag. If the flag indicates the TSR has been requested and if system hardware is inactive, the handler executes the TSR. Since control must eventually return to the idle DOS function which has stored data on the I/O stack, the TSR in this case must not call any DOS function that also uses the I/O stack. Table 19.1 shows which functions set up the I/O stack for various versions of DOS.

Table 19.1 DOS Internal Stacks

Function	Critical Error flag	DOS Version 2.x	3.0	3.1+
01–0Ch	Clear	I/O*	I/O	I/O
	Set	Aux*	Aux	Aux
33h	Clear	Disk*	Disk	Caller*
	Set	Disk	Disk	Caller
50h–51h	Clear	I/O	Caller	Caller
	Set	Aux	Caller	Caller
59h	Clear	n/a*	I/O	Disk
	Set	n/a	Aux	Disk
5D0Ah	Clear	n/a	n/a	Disk
	Set	n/a	n/a	Disk
62h	Clear	n/a	Caller	Caller
	Set	n/a	Caller	Caller
All others	Clear	Disk	Disk	Disk
	Set	Disk	Disk	Disk

* I/O = I/O stack, Aux = auxiliary stack, Disk = disk stack, Caller = caller's stack, n/a = function not available.

TSRs that perform tasks of long or indefinite duration should themselves call Interrupt 28h. For example, a TSR that polls for keyboard input should include an **INT 28h** instruction in the polling loop, as shown here:

```
poll:   int     28h             ; Signal idle state
        mov     ah, 1
        int     16h             ; Key waiting?
        jnz     poll            ; If not, repeat polling loop
        sub     ah, ah
        int     16h             ; Otherwise, get key
```

This courtesy gives other TSRs a chance to execute if the InDos flag happens to be set.

19.4.4 Monitoring the Critical Error Flag

DOS sets the Critical Error flag to a nonzero value when it detects a critical error. It then invokes Interrupt 24h (Critical Error Handler) and clears the flag when Interrupt 24h returns. DOS functions higher than 0Ch are illegal during critical

error processing. Therefore, a TSR that calls DOS must not execute while the Critical Error flag is set.

DOS versions 3.1 and later locate the Critical Error flag in the byte preceding the InDos flag. A single call to Function 34h (Get Address of InDos Flag) thus effectively returns the addresses of both flags. For earlier versions of DOS or for the compatibility version of DOS in OS/2, a TSR must call Function 34h and then scan the segment returned in the ES register for one of the two following sequences of instructions:

```
; Sequence of instructions in DOS Versions 2.0 - 3.0
        cmp     ss:[CriticalErrorFlag], 0
        jne     @F
        int     28h

; Sequence of instructions in OS/2's compatibility
;  version of DOS
        test    [CriticalErrorFlag], 0FFh
        jnz     @F
        push    ss:[ ? ]
        int     28h
```

The question mark inside brackets in the **PUSH** statement above indicates that the operand for the **PUSH** instruction can be any legal operand.

In either version of DOS, the operand field in the first instruction gives the flag's offset. The value in ES determines the segment address. The example program presented in Section 19.8 demonstrates how to locate the Critical Error flag with this technique.

19.5 Preventing Interference

This section describes how an active TSR can avoid interfering with the process it interrupts. Interference occurs when a TSR commits an error or performs an action that affects the interrupted process after the TSR returns. Examples of interference range from the relatively harmless, such as moving the cursor, to the serious, such as overrunning a stack.

Although a TSR can potentially interfere with another process in many different ways, protection against interference involves only three steps:

1. Recording a current configuration
2. Changing the configuration so it applies to the TSR
3. Restoring the original configuration before terminating

The example program in Section 19.8 demonstrates all the noninterference safeguards described in this section. These safeguards by no means exhaust the

subject of noninterference. More sophisticated TSRs may require more sophisticated methods. However, noninterference methods generally fall into one of the following categories:

- Trapping errors
- Preserving an existing condition
- Preserving existing data

19.5.1 Trapping Errors

A TSR committing an error that triggers an interrupt must handle the interrupt to trap the error. Otherwise, the existing interrupt routine, which belongs to the underlying process, would attempt to service an error the underlying process did not commit.

For example, a TSR that accepts keyboard input should include handlers for Interrupts 23h and 1Bh to trap keyboard break signals. When DOS detects CTRL+C from the keyboard or input stream, it transfers control to Interrupt 23h (CTRL+C Handler). Similarly, the BIOS keyboard routine calls Interrupt 1Bh (CTRL+BREAK Handler) when it detects a CTRL+BREAK key combination. Both routines normally terminate the current process.

A TSR that calls DOS should also trap critical errors through Interrupt 24h (Critical Error Handler). DOS functions call Interrupt 24h when they encounter certain hardware errors. The TSR must not allow the existing interrupt routine to service the error, since the routine might allow the user to abort service and return control to DOS. This would terminate both the TSR and the underlying process. By handling Interrupt 24h, the TSR retains control if a critical error occurs.

An error-trapping handler differs in two ways from a TSR's other handlers:

1. It is temporary, in service only while the TSR executes. At start-up, the TSR copies the handler's address to the interrupt vector table; it then restores the original vector before terminating.
2. It provides complete service for the interrupt; it does not pass control on to the original routine. However, if the error is not a TSR error, the handler needs to pass the error to the original routine.

Error-trapping handlers often set a flag to let the TSR know that the error has occurred. For example, a handler for Interrupt 1Bh might set a flag when the user

presses CTRL+BREAK. The TSR can check the flag as it polls for keyboard input, as shown here:

```
BrkHandler PROC FAR              ; Handler for Interrupt 1Bh

        mov     BreakFlag, TRUE  ; Raise break flag
        iret                     ; Terminate interrupt

BrkHandler ENDP
        .
        .
        .
        mov     BreakFlag, FALSE ; Initialize break flag
poll:   .
        .
        .
        cmp     BreakFlag, TRUE  ; Keyboard break pressed?
        je      exit             ; If so, break polling loop
        mov     ah, 1
        int     16h              ; Key waiting?
        jnz     poll             ; If not, repeat polling loop
```

19.5.2 Preserving an Existing Condition

A TSR and its interrupt handlers must preserve register values so that all registers are returned intact to the interrupted process. This is usually done by pushing the registers onto the stack before changing them, then popping the original values before returning.

Setting up a new stack is another important safeguard against interference. A TSR should usually provide its own stack to avoid the possibility of overrunning the current stack. Exceptions to this rule are simple TSRs such as the sample program ALARM that make minimal stack demands.

A TSR that alters the video configuration should return the configuration to its original state upon return. Video configuration includes cursor position, cursor shape, and video mode. The services provided through Interrupt 10h enable a TSR to determine the existing configuration and alter it if necessary.

However, some applications set video parameters by directly programming the video controller. When this happens, BIOS remains unaware of the new configuration and consequently returns inaccurate information to the TSR. Unfortunately, there is no solution to this problem if the controller's data registers provide write-only access and thus cannot be queried directly. For more information on video controllers, refer to Richard Wilton, *Programmer's Guide to the PC & PS/2 Video Systems*. (See "Books for Further Reading" in the Introduction.)

19.5.3 Preserving Existing Data

A TSR requires its own disk transfer area (DTA) if it calls DOS functions that access the DTA. These include file control block functions, as well as Functions 11h, 12h, 4Eh, and 4Fh. The TSR must switch to a new DTA to avoid overwriting the one belonging to the interrupted process. On becoming active, the TSR calls Function 2Fh to obtain the address of the current DTA. The TSR stores the address and then calls Function 1Ah to establish a new DTA. Before returning, the TSR again calls Function 1Ah to restore the address of the original DTA.

DOS versions 3.1 and later allow a TSR to preserve extended error information. This prevents the TSR from destroying the original information if it commits a DOS error.

The TSR retrieves the current extended error data by calling DOS Function 59h. It then copies registers AX, BX, CX, DX, SI, DI, DS, and ES to an 11-word data structure in the order given. DOS reserves the last three words of the structure, which should each be set to zero. Before returning, the TSR calls Function 5Dh, with AL equalling 0Ah and DS:DX pointing to the data structure. This call restores the extended error data to their original state.

19.6 Communicating through the Multiplex Interrupt

The Multiplex interrupt (Interrupt 2Fh) provides the Microsoft-approved way for a program to verify the presence of an installed TSR and to exchange information with it. DOS version 2.*x* uses Interrupt 2Fh only as an interface for the resident print spooler utility PRINT.COM. Later DOS versions standardize calling conventions so that multiple TSRs can share the interrupt.

A TSR chains to the Multiplex interrupt by setting up a handler. The TSR's installation code records the Interrupt 2Fh vector and then replaces it with the address of the new multiplex handler.

19.6.1 The Multiplex Handler

A program communicates with a multiplex handler by calling Interrupt 2Fh with an identity number in the AH register. As each handler in the chain gains control, it compares the value in AH with its own identity number. If the handler finds that it is not the intended recipient of the call, it passes control to the previous handler. The process continues until control reaches the target handler. When the target handler finishes its tasks, it returns via an **IRET** instruction to terminate the interrupt.

The target handler determines its tasks from the function number in AL. Convention reserves Function 0 as a request for installation status. A multiplex handler must respond to Function 0 by setting AL to 0FFh, to inform the caller of the handler's presence in memory. The handler should also return other information to provide a completely reliable identification. For example, it might return in ES:BX a far pointer to the TSR's copyright notice. This assures the caller it has located the intended TSR and not another TSR that has already claimed the identity number in AH.

Identity numbers range from 192 to 255, since DOS reserves lesser values for its own use. During installation, a TSR must verify the uniqueness of its number. It must not set up a multiplex handler identified by a number already in use. A TSR usually obtains its identity number through one of the following methods:

- The programmer assigns the number in the program.
- The user chooses the number by entering it as an argument in the command line, placing it into an environment variable, or by altering the contents of an initialization file.
- The TSR selects its own number through a process of trial and error.

The last method offers the most flexibility. It finds an identity number not currently in use among the installed multiplex handlers and does not require intervention from the user.

To use this method, a TSR calls Interrupt 2Fh during installation, with AH = 192 and AL = 0. If the call returns AL = 0FFh, the program tests other registers to determine if it has found a prior installation of itself. If the test fails, the program resets AL to zero, increments AH to 193, and again calls Interrupt 2Fh. The process repeats with incrementing values in AH until the TSR locates a prior installation of itself—in which case it should abort with an appropriate message to the user—or until AL returns as zero. The TSR can then use the value in AH as its identity number and proceed with installation.

The SNAP.ASM program in Section 19.8 demonstrates how a TSR can use this trial-and-error method to select a unique identity number. During installation, the program calls Interrupt 2Fh to verify that SNAP is not already installed. When deinstalling, the program again calls Interrupt 2Fh to locate the resident TSR in memory. SNAP's multiplex handler services the call and returns the address of the resident code's program-segment prefix. The calling program can then locate the resident code and deinstall it, as explained in Section 19.7.

19.6.2 Using the Multiplex Interrupt Under DOS Version 2.*x*

A TSR can use the Multiplex interrupt under DOS version 2.*x* with certain limitations. Under version 2.*x*, only DOS's print spooler PRINT, itself a TSR program, provides an Interrupt 2Fh service. The Interrupt 2Fh vector remains null until PRINT or another TSR is installed that sets up a multiplex handler.

Therefore, a TSR running under version 2.*x* must first check the existing Interrupt 2Fh vector before installing a multiplex handler. The TSR locates the current Interrupt 2Fh handler through Function 35h (Get Interrupt Vector). If the function returns a null vector, the TSR's handler will be last in the chain of Interrupt 2Fh handlers. The handler must terminate with an **IRET** instruction rather than pass control to a nonexistent routine.

PRINT in DOS version 2.*x* does not pass control on to the previous handler. If the user intends to run PRINT under version 2.*x*, the program must be installed before other TSRs that also handle Interrupt 2Fh. This places PRINT's multiplex handler last in the chain of handlers.

19.7 Deinstalling TSRs

A TSR should provide a means for the user to remove or "deinstall" it from memory. Deinstallation returns occupied memory to the system, offering these benefits:

- The freed memory becomes available to subsequent programs which may require additional memory space.
- Deinstallation restores the system to a normal state. This allows sensitive programs that may be incompatible with TSRs a chance to execute without the presence of installed routines.

A deinstallation program must first locate the TSR in memory, usually by requesting an address from the TSR's multiplex handler. When it has located the TSR, the deinstallation program should then compare addresses in the vector table with the addresses of the TSR's handlers. A mismatch indicates that another TSR has chained a handler to the interrupt routine. In this case, the deinstallation program should deny the request to deinstall. If the addresses of the TSR's handlers match those in the vector table, deinstallation can safely continue.

Deinstall the TSR in three steps:

1. Restore to the vector table the original interrupt vectors replaced by the handler addresses.
2. Read the segment address stored at offset 2Ch of the resident TSR's program segment prefix (PSP). This address points to the TSR's "environment block," a list of environment variables that DOS copies into memory when it loads a program. Place the block's address in the ES register and call DOS Function 49h (Release Memory Block) to return the block's memory to the operating system.
3. Place the resident PSP segment address in ES and again call Function 49h. This call releases the block of memory occupied by the TSR's code and data.

The example program in the next section demonstrates how to locate a resident TSR through its multiplex handler and deinstall it from memory.

19.8 Example of an Advanced TSR: SNAP

This section presents SNAP, a memory-resident program that demonstrates most of the techniques discussed in the chapter. SNAP takes a snapshot of the current screen and copies the text to a specified file. SNAP accommodates screens with various column and line counts, such as CGA's 40-column mode or VGA's 50-line mode. The program ignores graphics screens.

Once installed, SNAP occupies approximately 7.5K (kilobytes) of memory. When it detects the ALT+LEFT SHIFT+S key combination, SNAP displays a prompt for a file specification. The user can type a new file name, accept the previous file name by pressing ENTER, or press ESC to cancel the request.

SNAP reads text directly from the video buffer and copies it to the specified file. The program sets the file pointer to the end of the file so that text is appended without overwriting previous data. SNAP copies each line only to the last character, ignoring trailing spaces. The program adds a carriage return–linefeed sequence (0D0Ah) to the end of each line. This makes the file accessible to any text editor that can read ASCII files.

To demonstrate how a program accesses resident data through the Multiplex interrupt, SNAP can reset the display attribute of its prompt box. After installing SNAP, run the main program with the /C option to change box colors:

```
SNAP /Cxx
```

The argument *xx* specifies the desired attribute as a two-digit hexadecimal number—for example, 7C for red on white, or 0F for monochrome high intensity. For a list of color and monochrome display attributes, refer to a description of Basic's **COLOR** command or to the "Tables" section of the *Macro Assembler Reference*.

SNAP can deinstall itself, provided another TSR has not been loaded after it. Deinstall SNAP by executing the main program with the /D option:

```
SNAP /D
```

If SNAP successfully deinstalls, it displays the following message:

```
TSR deinstalled
```

19.8.1 Building SNAP.EXE

SNAP combines four modules: SNAP.ASM, COMMON.ASM, HANDLERS.ASM, and INSTALL.ASM. Source files are located on one of your distribution disks. Each module stores temporary code and data in the segments INSTALLCODE and INSTALLDATA. These segments apply only to SNAP's installation phase; DOS recovers the memory they occupy when the program exits through the terminate-and-stay-resident function. The following briefly describes each module:

- SNAP.ASM contains the TSR's main code and data.
- COMMON.ASM contains procedures used by other example programs.
- HANDLERS.ASM contains interrupt handler routines for Interrupts 08, 09, 10h, 13h, 15h, 28h, and 2Fh. It also provides simple error-trapping handlers for Interrupts 1Bh, 23h, and 24h. Additional routines set up and deinstall the handlers.
- INSTALL.ASM contains an exit routine that calls the terminate-and-stay-resident function and a deinstallation routine that removes the program from memory. The module includes error-checking services and a command-line parser.

This building-block approach allows you to create other TSRs by replacing SNAP.ASM and linking with the HANDLERS and INSTALL object modules. The library of routines accommodates both keyboard-activated and time-activated TSRs. A time-activated TSR is a program that activates at a predetermined time of day, similar to the example program ALARM introduced in Section 19.3. The header comments for the `Install` procedure in HANDLERS.ASM explain how to install a time-activated TSR.

You can write new TSRs in assembly language or any high-level language that conforms to the Microsoft conventions for ordering segments. Regardless of the language, the new code must not invoke a DOS function that sets up the I/O stack (see Section 19.4.3). Code in Microsoft C, for example, must not call **getche** or **kbhit**, since these functions in turn call DOS Functions 01 and 0Bh.

Code written in a high-level language must not check for stack overflows. Compiler-generated stack probes do not recognize the new stack setup when the TSR executes, and therefore must be disabled. The example program BELL.C, included on disk with the TSR library routines, demonstrates how to disable stack checking in Microsoft C using the **check_stack** pragma.

19.8.2 Outline of SNAP

The following sections outline in detail how SNAP works. Each part of the outline covers a specific portion of SNAP's code. Headings refer to earlier sections of this chapter, providing cross-references to SNAP's key procedures. For example, the part of the outline that describes how SNAP searches for its start-up signal refers to Section 19.2.1, "Auditing Hardware Events for TSR Requests."

Figures 19.2 through 19.4 are flow charts of the SNAP program. Each chart illustrates a separate phase of SNAP's operation, from installation through memory-residency to deinstallation.

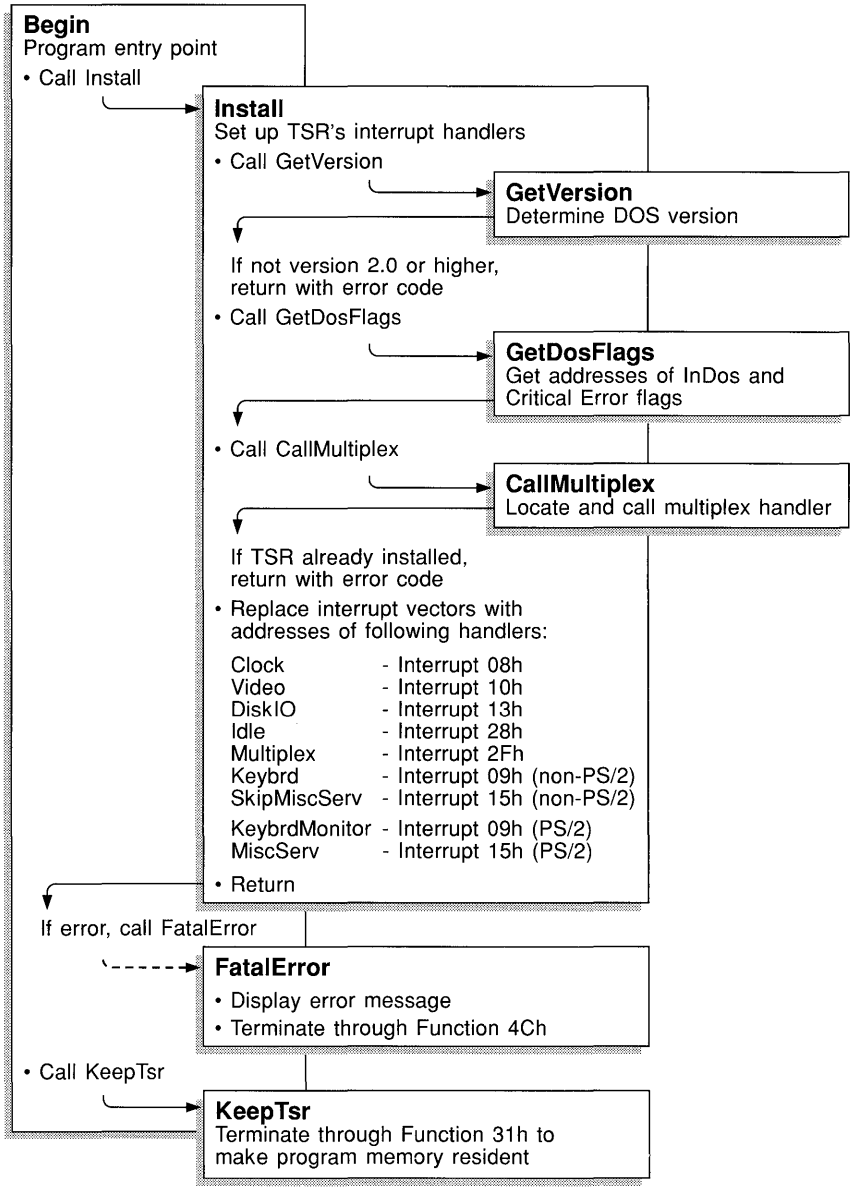


Figure 19.2 Flow Chart for SNAP.EXE: Installation Phase

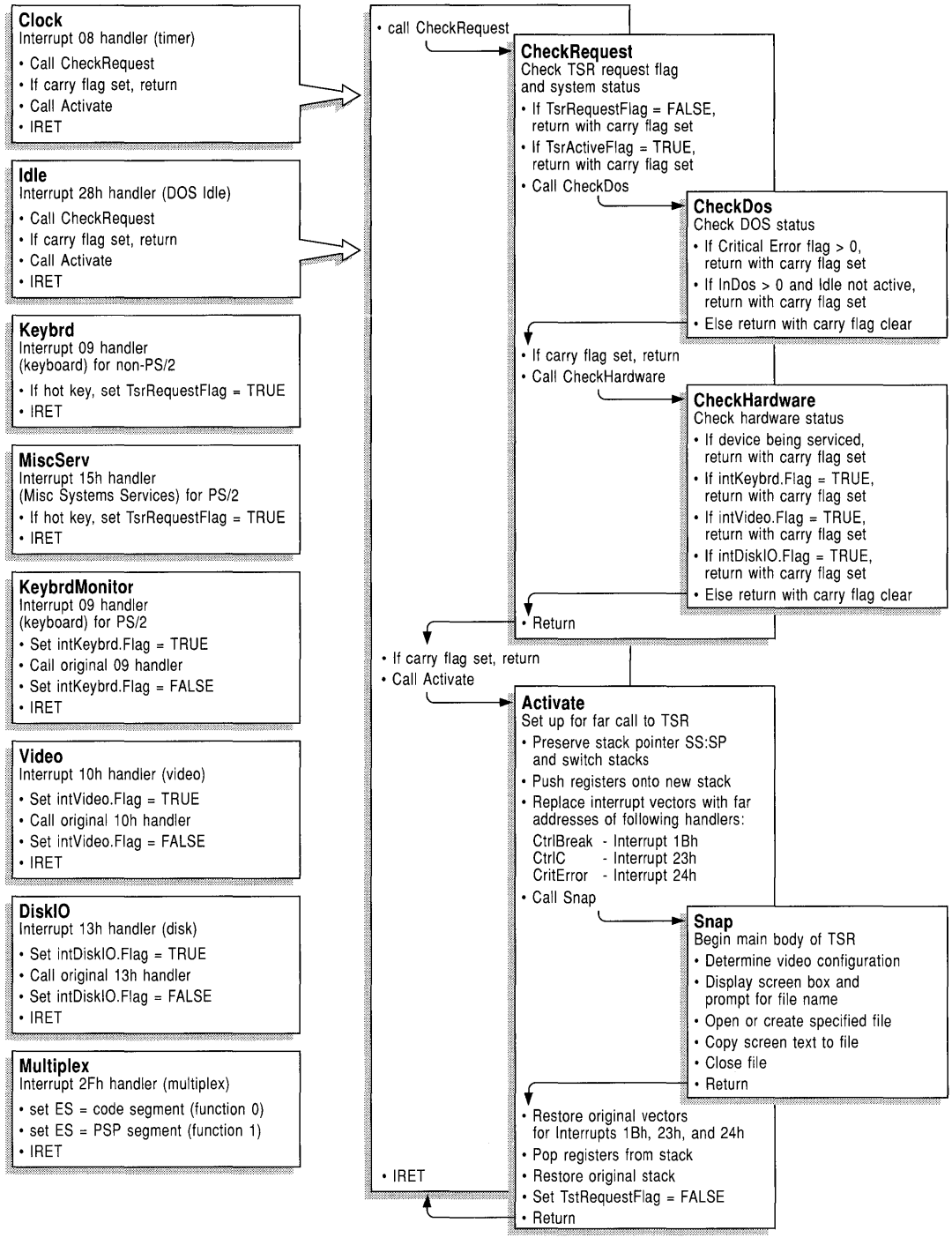


Figure 19.3 Flow Chart for SNAP.EXE: Resident Phase

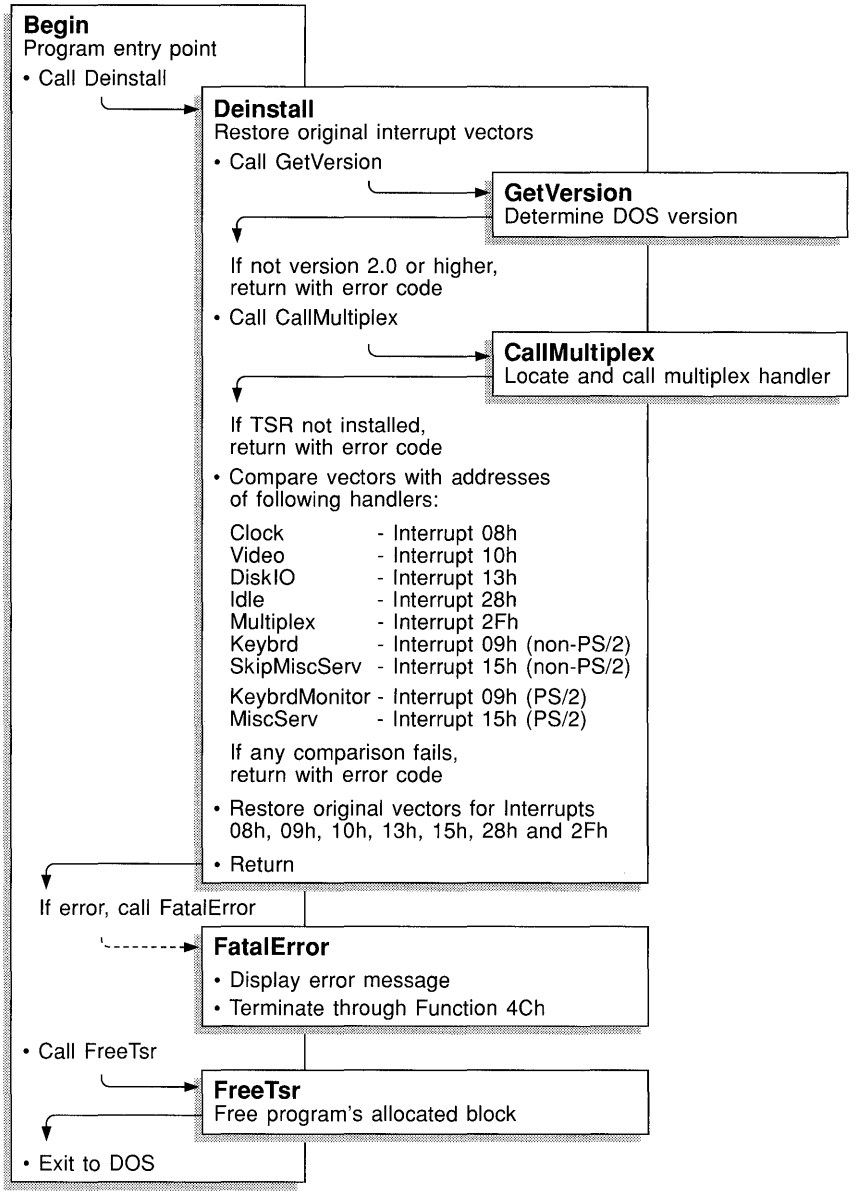


Figure 19.4 Flow Chart for SNAP.EXE: Deinstallation Phase

As you read through the following outline, you may wish to refer to the flow charts. They will help you maintain a larger perspective while exploring the details of SNAP's operation. Discussions in the outline cross-reference the charts.

Note that information in both the outline and the flow charts is generic. Except for references to the SNAP procedure, all descriptions in the outline and the flow charts apply to any TSR created with the HANDLERS and INSTALL modules.

Auditing Hardware Events for TSR Requests

To search for its start-up signal, SNAP audits the keyboard with an interrupt handler for either Interrupt 09 (keyboard) or Interrupt 15h (Miscellaneous System Services). See Section 19.2.1 for information on this topic. The `Install` procedure determines which of the two interrupts to handle based on the following code:

```
        .IF     HotScan == 0    ; If valid scan code given:
        mov     ah, HotShift    ; AH = hour to activate
        mov     al, HotMask     ; AL = minute to activate
        call    GetTimeToElapse ; Get number of 5-second intervals
        mov     CountDown, ax   ;  to elapse before activation

        .ELSE                   ; Force use of KeybrdMonitor as
                                ;  keyboard handler
        cmp     Version, 031Eh  ; DOS Version 3.3 or higher?
        jb      setup           ; No?  Skip next step

; Test for IBM PS/2 series. If not PS/2, use Keybrd and
; SkipMiscServ as handlers for Interrupts 09 and 15h
; respectively. If PS/2 system, set up KeybrdMonitor as the
; Interrupt 09 handler. Audit keystrokes with MiscServ
; handler, which searches for the hot key by handling calls
; to Interrupt 15h (Miscellaneous System Services). Refer to
; Section 19.2.1 for more information about keyboard handlers.

        mov     ax, 0C00h       ; Function 0Ch (Get System
        int     15h             ;  Configuration Parameters)
        sti                     ; Compaq ROM may leave disabled

        jc      setup           ; If carry set,
        or      ah, ah          ;  or if AH not 0,
        jnz     setup           ;  services are not supported

; Test bit 4 to see if Intercept is implemented
        test    BYTE PTR es:[bx+5], 00010000y
        jz      setup

; If so, set up MiscServ as Interrupt 15h handler
        mov     ax, OFFSET MiscServ
        mov     WORD PTR intMisc.NewHand, ax
        .ENDIF

; Set up KeybrdMonitor as Interrupt 09 handler
        mov     ax, OFFSET KeybrdMonitor
        mov     WORD PTR intKeybrd.NewHand, ax
```

This is the code's logic:

- If the program is running under DOS version 3.3 or higher and if Interrupt 15h supports Function 4Fh, set up handler `MiscServ` to search for the hot key. Handle Interrupt 09 with `KeybrdMonitor` only to maintain the keyboard active flag.
- Otherwise, set up a handler for Interrupt 09 to search for the hot key. Handle calls to Interrupt 15h with the routine `SkipMiscServ`, which contains this single instruction:

  ```
  jmp  cs:intMisc.OldHand
  ```

  The jump immediately passes control to the original Interrupt 15h routine; thus, `SkipMiscServ` has no effect. It serves only to simplify coding in other parts of the program.

At each keystroke, the keyboard interrupt handler (either `Keybrd` or `MiscServ`) calls the procedure `CheckHotKey` with the scan code of the current key. `CheckHotKey` compares the scan code and shift status with the bytes `HotScan` and `HotShift`. If the current key matches, `CheckHotKey` returns the carry flag clear to indicate that the user has pressed the hot key.

If the keyboard handler finds the carry flag clear, it sets the flag `TsrRequestFlag` and exits. Otherwise, the handler transfers control to the original interrupt routine to service the interrupt.

The timer handler `Clock` reads the request flag at every occurrence of the timer interrupt. `Clock` takes no action if it finds a zero value in `TsrRequestFlag`. Figures 19.1 and 19.3 depict the relationship between the keyboard and timer handlers.

Monitoring System Status

Because SNAP produces output to both video and disk, it avoids interrupting either video or disk operations. The program uses interrupt handlers `Video` and `DiskIO` to monitor Interrupts 10h (video) and 13h (disk). SNAP also avoids interrupting keyboard use. The instructions at the far label `KeybrdMonitor` serve as the monitor handler for Interrupt 09 (keyboard). See Section 19.2.2 for information on this topic.

The three handlers perform similar functions. Each sets an active flag and then calls the original routine to service the interrupt. When the service routine returns, the handler clears the active flag to indicate that the device is no longer in use.

The BIOS Interrupt 13h routine clears or sets the carry flag to indicate the operation's success or failure. DiskIO therefore preserves the flags register when returning, as shown here:

```
DiskIO  PROC    FAR
        mov     cs:intDiskIO.Flag, TRUE ; Set active flag
; Simulate interrupt by pushing flags and far-calling old
;  Int 13h routine
        pushf
        call    cs:intDiskIO.OldHand
; Clear active flag without disturbing flags register
        mov     cs:intDiskIO.Flag, FALSE
        sti                             ; Enable interrupts
; Simulate IRET without popping flags (since services use
;  carry flag)
        ret     2
DiskIO  ENDP
```

The terminating **RET 2** instruction discards the original flags from the stack when the handler returns.

Determining Whether to Invoke the TSR

The procedure CheckRequest determines if the TSR

- Has been requested
- Can safely interrupt the system

Each time it executes, the timer handler Clock calls CheckRequest to read the flag TsrRequestFlag. If CheckRequest finds the flag set, it scans other flags maintained by the TSR's interrupt handlers and by DOS. These flags indicate the current system status. As the flow chart in Figure 19.3 shows, CheckRequest calls CheckDos (described below) to determine the status of the operating system. CheckRequest then calls CheckHardware to check hardware status. See Section 19.2.2 for information on this topic.

CheckHardware queries the interrupt controller to determine if any device is currently being serviced. It also reads the active flags maintained by the KeybrdMonitor, Video, and DiskIO handlers. If the controller, keyboard, video, and disk are all inactive, CheckHardware clears the carry flag and returns.

CheckRequest indicates system status with the carry flag. If the procedure returns the carry flag set, the caller exits without invoking the TSR. A clear carry signals that the caller can safely execute the TSR.

Determining DOS Activity

As Figure 19.2 shows, the procedure GetDosFlags locates the InDos flag during SNAP's installation phase. GetDosFlags calls Function 34h (Get

Address of InDos Flag) and then stores the flag's address in the far pointer `InDosAddr`. See Section 19.4.2 for information on this topic.

When called from the `CheckRequest` procedure, `CheckDos` reads InDos to determine if the operating system is active. Note that `CheckDos` reads the flag directly from the address in `InDosAddr`. It does not call Function 34h to locate the flag, since it has not yet established whether DOS is active. This follows from the general rule that interrupt handlers must not call any DOS function.

The next two sections describe the procedure `CheckDos` more fully.

Interrupting DOS Functions

Figure 19.3 shows that the call to `CheckDos` can initiate either from `Clock` (timer handler) or `Idle` (Interrupt 28h handler). If `CheckDos` finds the InDos flag set, it reacts in different ways depending on the caller:

- If called from `Clock`, `CheckDos` cannot know which DOS function is active. In this case, it returns the carry flag set, indicating that `Clock` must deny the request for the TSR.
- If called from `Idle`, `CheckDos` assumes that one of the low-order polling functions is active. It therefore clears the carry flag to let the caller know the TSR can safely interrupt the function.

See Section 19.4.3 for information on this topic.

Monitoring the Critical Error Flag

The procedure `GetDosFlags` (Figure 19.2) determines the address of the Critical Error flag. The procedure stores the flag's address in the far pointer `CritErrAddr`. See Section 19.4.4 for information on this topic.

When called from either the `Clock` or `Idle` handlers, `CheckDos` reads the Critical Error flag. A nonzero value in the flag indicates that the Critical Error Handler (Interrupt 24h) is processing a critical error and the TSR must not interrupt. In this case, `CheckDos` sets the carry flag and returns, causing the caller to exit without executing the TSR.

Trapping Errors

As Figure 19.3 shows, `Clock` and `Idle` invoke the TSR by calling the procedure `Activate`. See Section 19.5.1 for information on this topic. Before calling the main body of the TSR, `Activate` sets up the following handlers:

Handler Name	For Interrupt	Receives Control When
`CtrlBreak`	1Bh (CTRL+BREAK Handler)	CTRL+BREAK sequence entered at keyboard
`CtrlC`	23h (CTRL+C Handler)	DOS detects a CTRL+C sequence from the keyboard or input stream
`CritError`	24h (Critical Error Handler)	DOS encounters a critical error

These handlers trap keyboard break signals and critical errors that would otherwise trigger the original handler routines. The `CtrlBreak` and `CtrlC` handlers contain a single **IRET** instruction, thus rendering a keyboard break ineffective. The `CritError` handler contains the following instructions:

```
CritError PROC  FAR
        sti
        sub     al, al          ; Assume DOS 2.x
                                ; Set AL = 0 for ignore error
        .IF     cs:major != 2   ; If DOS 3.x, set AL = 3
        mov     al, 3           ; DOS call fails
        .ENDIF
        iret
CritError ENDP
```

The return code in AL forces DOS to take no further action when it encounters a critical error.

As an added precaution, `Activate` also calls Function 33h (Get or Set CTRL+BREAK Flag) to determine the current setting of the checking flag. `Activate` stores the setting, then calls Function 33h again to turn off break checking.

When the TSR's main procedure finishes its work, it returns to `Activate`, which then restores the original setting for the checking flag. It also replaces the original vectors for Interrupts 1Bh, 23h, and 24h.

SNAP's error-trapping safeguards enable the TSR to retain control in the event of an error. Pressing CTRL+BREAK or CTRL+C at SNAP's prompt has no effect. If the user specifies a nonexistent drive—a critical error—SNAP merely beeps the speaker and returns normally.

Preserving an Existing Condition

`Activate` records the stack pointer SS:SP in the doubleword `OldStackAddr`. The procedure then resets the pointer to the address of a new stack before calling the TSR. Switching stacks ensures that SNAP has adequate stack depth while it executes. See Section 19.5.2 for information on this topic.

The label `NewStack` points to the top of the new stack buffer, located in the code segment of the HANDLERS.ASM module. The equate constant `STACK_SIZ` determines the size of the stack. The include file TSR.INC contains the declaration for `STACK_SIZ`.

`Activate` preserves the values in all registers by pushing them onto the new stack. It does not push DS, since that register is already preserved in the `Clock` or `Idle` handler.

SNAP does not alter the application's video configuration other than by moving the cursor. Figure 19.3 shows that `Activate` calls the procedure `Snap`, which executes Interrupt 10h to determine the current cursor position. `Snap` stores the row and column in the word `OldPos`. The procedure restores the cursor to its original location before returning to `Activate`.

Preserving Existing Data

Because SNAP does not call a DOS function that writes to the DTA, it does not need to preserve the DTA belonging to the interrupted process. However, the code for switching and restoring the DTA is included within **IFDEF** blocks in the procedure `Activate`. The equate constant `DTA_SIZ`, declared in the TSR.INC file, governs the assembly of the blocks as well as the size of the new DTA. See Section 19.5.3 for information on this topic.

SNAP can potentially overwrite existing extended error information by committing a file error. The program does not attempt to preserve the original information by calling Functions 59h and 5Dh. In certain rare instances, this may confuse the interrupted process after SNAP returns.

Communicating through the Multiplex Interrupt

The program uses the Multiplex interrupt (Interrupt 2Fh) to

- Verify that SNAP is installed
- Select a unique multiplex identity number
- Locate resident data

See Section 19.6 for information on this topic.

SNAP accesses Interrupt 2Fh through the procedure `CallMultiplex`, as shown in Figures 19.2 and 19.4. By searching for a prior installation, `CallMultiplex` ensures that SNAP is not installed more than once. During deinstallation, `CallMultiplex` locates data required to deinstall the resident TSR.

The procedure `Multiplex` serves as SNAP's multiplex handler. When it recognizes its identity number in AH, `Multiplex` determines its tasks from the function number in the AL register. The handler responds to Function 0 by returning AL equalling 0FFh and ES:DI pointing to an identifier string unique to SNAP.

`CallMultiplex` searches for the handler by invoking Interrupt 2Fh in a loop, beginning with a trial identity number of 192 in AH. At the start of each iteration of the loop, the procedure sets AL to zero to request presence verification from the multiplex handler. If the handler returns 0FFh in AL, `CallMultiplex` compares its copy of SNAP's identifier string with the text at memory location ES:DI. A failed match indicates that the multiplex handler servicing the call is not SNAP's handler. In this case, `CallMultiplex` increments AH and cycles back to the beginning of the loop.

The process repeats until the call to Interrupt 2Fh returns a matching identifier string at ES:DI or until AL returns as zero. A matching string verifies that SNAP is installed, since its multiplex handler has serviced the call. A return value of zero indicates that SNAP is not installed and that no multiplex handler claims the trial identity number in AH. In this case, SNAP assigns the number to its own handler.

Deinstalling TSRs

During deinstallation, `CallMultiplex` locates SNAP's multiplex handler as described above. The handler `Multiplex` receives the verification request and returns in ES the code segment of the resident program. See Section 19.7 for information on this topic.

`Deinstall` reads the addresses of the following interrupt handlers from the data structure in the resident code segment:

Handler Name	Description
`Clock`	Timer handler
`Keybrd`	Keyboard handler (non-PS/2)
`KeybrdMonitor`	Keyboard monitor handler (PS/2)
`Video`	Video monitor handler
`DiskIO`	Disk monitor handler
`SkipMiscServ`	Miscellaneous Systems Services handler (non-PS/2)
`MiscServ`	Miscellaneous Systems Services handler (PS/2)
`Idle`	DOS Idle handler
`Multiplex`	Multiplex handler

`Deinstall` calls DOS Function 35h (Get Interrupt Vector) to retrieve the current vectors for each of the listed interrupts. By comparing each handler address with the corresponding vector, `Deinstall` ensures that SNAP can be safely deinstalled. Failure in any of the comparisons indicates that another TSR has been installed after SNAP and has set up a handler for the same interrupt. In this case, `Deinstall` returns an error code, causing the program to abort with the following message:

```
Can't deinstall TSR
```

If all addresses match, `Deinstall` calls Interrupt 2Fh with SNAP's identity number in AH and AL set to 1. The handler `Multiplex` responds by returning in ES the address of the resident code's PSP. `Deinstall` then calls DOS Function 25h (Set Interrupt Vector) to restore the vectors for the original service routines. This is called "unhooking" or "unchaining" the interrupt handlers.

After unhooking all of SNAP's interrupt handlers, `Deinstall` returns with AX pointing to the resident code's PSP. The procedure `FreeTsr` then calls DOS Function 49h (Release Memory) to return SNAP's memory to the operating system. The program terminates with the message

```
TSR deinstalled
```

to indicate a successful deinstallation.

Deinstalling SNAP does not guarantee more available memory space for the next program. If another TSR loads after SNAP but handles interrupts other than 08, 09, 10h, 13h, 15h, 28h, or 2Fh, SNAP still deinstalls properly. The result is a harmless gap of deallocated memory formerly occupied by SNAP. DOS can use the free memory to store the next program's environment block. However, DOS loads the program itself above the still-resident TSR.

19.9 Related Topics in Online Help

In addition to information covered in this chapter, information on the following topics can be found in online help.

Topic	Access
DOS and BIOS function calls	From the "MASM 6.0 Contents" screen, choose "DOS Calls" or "BIOS Calls" from the list of "System Resources"
Processor Flags	From the "MASM 6.0 Contents" screen, choose "Language Overview" and then choose "Processor Flag Summary"
IN, OUT	From the "MASM 6.0 Contents" screen, choose "Processor Instructions" and then choose "System and I/O Access"

Chapter 20
Mixed-Language Programming

Mixed-language programming allows you to combine the unique strengths of Microsoft Basic, C, FORTRAN, and Pascal with your assembly-language routines. Any one of these languages can call MASM routines, and you can call any of these languages from within MASM routines. This makes virtually all of the routines from extensive high-level-language libraries available to a mixed-language program.

MASM 6.0 has a number of new features that make the interface in assembly-language programs similar to the interface in high-level-language programs. For example, you can now use the **INVOKE** directive to call high-level-language procedures, and the assembler handles the argument-passing details for you. You can also use H2INC to translate C header files to MASM include files (see Chapter 16).

The new mixed-language features do not make the older methods of defining mixed-language interfaces obsolete. In most cases mixed-language programs written with previous versions of MASM will assemble and link correctly under MASM 6.0. (See Appendix A for more information.)

This chapter explains how to write assembly routines that can be called from high-level-language modules and how to call high-level language routines from MASM. It assumes that you have a basic understanding of the languages you wish to combine and that you already know how to write, compile, and link multiple-module programs with these languages.

This chapter is restricted to MASM's interface with C, Basic, FORTRAN, and Pascal; it does not cover mixed-language programming between high-level languages. The focus in this chapter is the Microsoft versions of C, Basic, FORTRAN, Pascal, and QuickPascal, but the same principles apply to other languages and compilers. The material in Section 7.3 on writing procedures in MASM and in Chapter 8 on multiple-module programming explains many of the techniques used in this chapter.

Section 20.1 looks at naming and calling conventions, and Section 20.2 provides a template for writing the MASM procedure. Specific implementations of this convention in C, Basic, FORTRAN, and Pascal are described in Section 20.3. These language-specific sections also provide details on how the language manages various data structures so that your MASM programs are compatible with the data from the high-level language. This chapter also contains examples of MASM procedures called from C, FORTRAN, Basic, Pascal, and QuickPascal.

20.1 Naming and Calling Conventions

The naming convention specifies the way the compiler or assembler alters the name of the routine or identifier before placing it into an object file. Each language alters the name of the identifiers. You must be sure that the naming conventions for mixed-language programming are compatible.

A calling convention specifies the way a language implements a call to a procedure. MASM implements mixed-language calls according to the particular calling convention specified in the procedure declaration or prototype.

MASM supports three different calling conventions. The assembler uses the C calling convention when the *langtype* is **C** or **SYSCALL**; it uses the Pascal calling convention when the *langtype* is **PASCAL**, **BASIC**, or **FORTRAN**; and it uses the **STDCALL** calling convention when the *langtype* is **STDCALL**. To MASM, **BASIC**, **PASCAL**, and **FORTRAN** are synonymous when specifying the Pascal calling convention for a procedure.

There are several ways to set the calling convention. Using **.MODEL** with a *langtype* sets the default for the module. You can also use the **OPTION** directive to do the same. This is equivalent to the /Gc or /Gd option from the command line. Procedure prototypes and declarations can specify a *langtype* to override the default.

You can change the default calling convention.

When you write mixed-language routines, the easiest way to ensure calling convention compatibility is to adopt the calling conventions of the language of the called procedure. However, Microsoft languages (except QuickPascal) can change their calling conventions, so at times you may want to change the calling convention to use a particular argument-passing method instead of the defaults for a particular language. Section 20.4 explains how to change the calling convention. The fastcall calling convention is not directly supported by the assembler. This section provides more detail on the information summarized in Table 20.1:

Table 20.1 Naming and Calling Conventions

Convention	C	SYSCALL	STDCALL	BASIC	FORTRAN	PASCAL
Leading underscore	X		X			
Capitalize all				X	X	X
Arguments pushed left to right				X	X	X
Arguments pushed right to left	X	X	X			
Caller stack cleanup	X		*			
:VARARG allowed	X	X	X			

*The STDCALL language type uses caller stack cleanup if the :VARARG parameter is used. Otherwise, the called routine must clean up the stack.

20.1.1 Naming Conventions

The naming convention determines the way the compiler or assembler stores identifiers. If you set the LINK command-line option /NOI, then the names of public variables or called routines are stored differently in the object modules being linked. As a result, LINK will not be able to find a match. It will instead report unresolved external references. Therefore, you must use valid identifiers for each language and be sure the naming convention for the linked modules is the same.

The **C** naming convention is used when the *langtype* is **C** or **STDCALL**, the **SYSCALL** naming convention is used when the *langtype* is **SYSCALL**, and the Pascal naming convention is used when the *langtype* is **PASCAL**, **BASIC**, or **FORTRAN**. The list below describes each convention. For example, assume you

have a variable named `Big Time` in your source code. The list below shows the result of each convention applied to this variable.

Langtype Specified	Characteristics
C, STDCALL	The assembler and the compiler add leading underscores to the names seen by the linker. They do not translate case. The linker sees the variable as `_Big Time`.
SYSCALL	Leaves the name unmodified. The linker sees the variable as `Big Time`.
PASCAL, FORTRAN, BASIC	Converts all names to uppercase. The linker sees the variable as `BIG TIME`.

20.1.2 The C Calling Convention

C and SYSCALL are identical as calling conventions.

You must specify the C calling convention for MASM routines that link with C modules using the default calling convention. You can change the default calling convention for FORTRAN, Basic, and Pascal routines to the C calling convention, if you prefer. The characteristics of the C calling convention are summarized below.

Because the C calling convention allows for a variable number of arguments to be passed to the procedure, you may want to use this convention when you need this flexibility.

When you specify **SYSCALL** for the *langtype*, the C calling convention is used, but a leading underscore is not added to the name of the global routine (see the next section). **SYSCALL** is provided for compatibility with system calls in OS/2 version 2.0.

Argument Passing With the C calling convention, the caller pushes arguments from right to left. The assembler places arguments on the stack in the reverse of the order that they appear in the source code. The first argument is lowest in memory (because it is the last argument to be placed on the stack, and the stack grows downward). The code to remove arguments from the stack follows the procedure call, so the caller pops arguments off the stack.

Register Preservation The called routine should save BP, SI, DI, DS, and SS if they are modified.

C and SYSCALL allow a variable number of arguments.

Varying Number of Arguments Because the first argument is always the last one pushed, it is always on the top of the stack. Thus, it has the same address relative to the frame pointer, regardless of how many arguments were actually passed. Therefore, calling procedures with a variable number of arguments are possible. If the high-level-language procedure uses the C calling convention and expects a variable number of arguments, the prototype for the function must end

with **:VARARG**. See Section 7.3.3, "Declaring Parameters with the PROC Directive," for information on using **PROC** and **INVOKE** with **VARARG**.

20.1.3 The Pascal Calling Convention

By default, the **FORTRAN**, **BASIC**, and **PASCAL** *langtype* select the Pascal calling convention. This convention pushes arguments left to right so that the last argument is lowest on the stack, and it requires that the called routine remove arguments from the stack. Section 20.3.4 explains the Pascal naming convention.

Argument Passing Arguments are placed on the stack in the same order in which they appear in the source code. The first argument is highest in memory (because it is also the first argument to be placed on the stack), and the stack grows downward.

Register Preservation Routines using the Pascal calling convention must preserve SI, DI, BP, and DS and not modify SS. (This does not apply to procedures called by QuickPascal. See Section 20.3.5.) For 32-bit code, the EBX, ES, FS, and GS registers must be preserved as well as EBP, ESI, and EDI. The direction flag is also cleared upon entry and must be preserved.

Varying Number of Arguments Passing a variable number of arguments is not possible with the Pascal calling convention.

20.1.4 The Standard Calling Convention

The **STDCALL** calling convention is the same as the C calling convention, with the exception that the responsibility for removing arguments from the stack belongs to the called routine. The C calling convention is followed exactly if the **STDCALL** procedure also specifies **VARARG**, allowing a variable number of parameters. **STDCALL** is provided for compatibility with 32-bit versions of Microsoft compilers which have **STDCALL** as their default.

Argument Passing Argument passing order is the same as the C calling convention. The caller pushes the arguments from right to left. Unlike the C calling convention, however, the called routine must remove arguments from the stack unless the routine uses **VARARG** to specify a variable number of arguments, in which case the caller removes the parameters from the stack.

Register Preservation Routines using the **STDCALL** convention must preserve the same registers required by the C calling convention: BP, SI, DI, DS, and SS. The direction flag is also cleared on entry and must be preserved.

Varying Number of Arguments If the routine uses **VARARG** to specify that a variable number of arguments can be passed, the calling routine must remove arguments from the stack.

20.2 Writing the Assembly-Language Procedure

MASM 6.0 simplifies the coding required for linking MASM routines to high-level-language routines. You can use the new **PROTO** directive to write procedure prototypes, and the new **INVOKE** directive to call external routines. This list summarizes the ways MASM simplifies procedure-related tasks.

- The **PROTO** directive improves error checking on argument types.
- **INVOKE** pushes arguments onto the stack and converts argument types to types expected when possible. These arguments can be referenced by their parameter label, rather than as offsets of the stack pointer.
- The **LOCAL** directive following the **PROC** statement saves places on the stack for local variables. These variables can also be referenced by name, rather than as offsets of the stack pointer.
- **PROC** sets up the appropriate stack frame according to the processor mode.
- The **USES** keyword preserves registers given as arguments.
- The C calling conventions specified in the **PROC** syntax allow for a variable number of arguments to be passed to the procedure.
- The **RET** keyword adjusts the stack upward by the number of bytes in the argument list, removes local variables from the stack, and pops saved registers.
- The **PROC** statement lists parameter names and types. The parameters can be referenced by name inside the procedure.

The complete syntax and parameter descriptions for these procedure directives are explained in Section 7.3, "Procedures." This section summarizes information from Section 7.3 by giving a template you can use for writing a MASM routine to be called from a high-level language.

The template looks like this:

Label **PROC** [[*distance langtype visibility <prologueargs>* **USES** *reglist parmlist*]]
    LOCAL *varlist*
    .
    .
    .
    RET
Label **ENDP**

Replace the italicized words with appropriate keywords, registers, or variables as defined by the syntax in Section 7.3.3, "Declaring Parameters with the **PROC** Directive."

The *distance* (**NEAR** or **FAR**) and *visibility* (**PUBLIC**, **PRIVATE**, or **EXPORT**) that you give in the procedure declaration override the current defaults. In some languages, the model can also be specified with command-line options.

The *langtype* determines the calling convention for accessing arguments and restoring the stack. See Section 20.1 for information on calling conventions.

The types for the parameters listed in the *parmlist* must be given. Also, if any of the parameters are pointers, the assembler does not generate code to get the value of the pointer references. You must write this code yourself. An example of how to do this is in Section 7.3.3.

If you need to code your own stack-frame setup manually, or if you do not want the assembler to generate the standard stack setup and cleanup, see Section 7.3.2, "Passing Arguments on the Stack," and, in Section 7.3.8.2, "User-Defined Prologue and Epilogue Code."

20.3 The MASM/High-Level-Language Interface

Since high-level-language routines require certain program initialization code, the main program for a mixed-language program must be written in the high-level language, or you must add `EXTERN A__ACRTUSED` to your program to force the start-up code from the high-level-language run times to be loaded. Once the high-level-language code calls an assembly routine, the assembly routine can then call high-level-language routines as needed.

Use INVOKE to call high-level-language procedures.

For procedures with prototypes, **INVOKE** makes calls from MASM to high-level-language programs, much like procedure or function calls in the high-level language. **INVOKE** calls procedures and generates the code to push arguments in the order specified by the procedure's calling convention and to remove arguments from the stack at the end of the procedure.

INVOKE can also do some type checking and data conversion for the argument types so that the procedure receives compatible data. Section 7.3.6, "Declaring Procedure Prototypes," explains how to write procedure prototypes and gives several examples of procedure declarations and the corresponding prototypes.

Use H2INC to translate C prototypes to MASM.

For programs that mix assembly language and C, the H2INC utility makes it easy to write prototypes and data declarations for the C procedures you want to call from MASM. H2INC translates the C prototypes and declarations into the corresponding MASM prototypes and declarations, which **INVOKE** can use to call the procedure. Chapter 16 explains how to use H2INC. See Section 20.3.1 for examples of using H2INC to write prototypes.

Mixed-language programming also allows the main program or a routine to use external data—data defined in the other module. External data is the data that is stored in a set place in memory (unlike dynamic and local data, which is allocated on the stack and heap) and is visible to other modules.

External data is shared by all routines. One of the modules must define the static data, which causes the compiler to allocate storage for the data. The other modules that access the data must declare the data as external.

This section describes argument-passing options and the standards for preserving registers and pushing addresses that are common to all high-level languages. It also explains the two methods that compilers use to store arrays—row-major and column-major order.

Argument Passing

Each language has its own convention for how an argument is actually passed. If the argument-passing conventions of your routines do not agree, then a called routine receives bad data. Microsoft languages support three different methods for passing an argument:

- Near reference. Passes a variable's near (offset) address. This address is expressed as an offset from the default data segment.

  This method gives the called routine direct access to the variable itself. Any change the routine makes to the parameter is reflected in the calling routine.

- Far reference. Passes a variable's far (segmented) address.

  This method is similar to passing by near reference, except that an address made up of a segment and an offset is passed, and it is slower. But it is necessary when you pass data that is outside of the default data segment. (This is not an issue in Basic or Pascal unless you have specifically requested far memory.)

- Value. Passes only the variable's value, not its address.

  With this method, the called routine gets the copy of the value of the argument but has no access to the original variable. Changes to a value argument have no effect on the value of the argument in the calling routine once the routine terminates.

You can also change the default argument-passing method.

When you pass arguments between MASM and another language, you need to make sure that the called routine and the calling routine use the same method. In most cases, you should check the argument-passing defaults used by each language and make any necessary adjustments. Most languages have features that allow you to change argument-passing methods.

Register Preservation

A procedure called from any high-level language should preserve the direction flag and the values of BP, SI, DI, SS, and DS. Routines called from MASM must not alter SI, DI, SS, DS, or BP.

Pushing Addresses

Microsoft high-level languages push segment addresses before pushing offsets. This facilitates use of the **LES** and **LDS** instructions. Furthermore, when pushing arguments longer than two bytes, high-order words are always pushed before low-order words, and arguments longer than two bytes are stored on the stack from most significant to least significant.

Array Storage

Most high-level-language compilers store arrays in row-major order. This means that all elements of a row are stored consecutively. The first five elements of an array with four rows and three columns are stored in row-major order as

```
A[1, 1], A[1, 2], A[1, 3], A[2, 1], A[2, 2]
```

In column-major order, the column elements are stored consecutively. For example, the same array defined above would be stored in column-major order as

```
A[1, 1], A[2, 1], A[3, 1], A[4, 1], A[1, 2], A[2, 2]
```

20.3.1 The C/MASM Interface

This section summarizes the details unique to the C and MASM interface. The information is accurate for Microsoft C 6.0 and QuickC version 2.5.

With the default naming and calling convention, the assembler (or compiler) pushes arguments right to left and adds a leading underscore to routine names.

Compatible Data Types This list shows the C data types that are equivalent to the MASM 6.0 data types.

C Type	Equivalent MASM Type
unsigned char	**BYTE**
char	**SBYTE**
unsigned short, unsigned int	**WORD**
int, short	**SWORD**
unsigned long	**DWORD**
float	**REAL4**

C Type	Equivalent MASM Type
long	SDWORD
double	REAL8
long double	REAL10

Naming Restrictions C is case sensitive and does not convert names to uppercase. Since C normally links with the /NOI command-line option, assemble MASM modules with the /Cx or /Cp option to prevent the assembler from converting names to uppercase.

Argument-Passing Defaults When the C module is compiled in small or medium model and when a distance is not specified, the C compiler passes arrays by near reference. In compact, large, or huge model, C arrays are passed by far reference (if a distance is not explicitly specified). All other types defined in the C module are passed by value. You can pass by reference if you specifically pass pointers or addresses.

Changing the Calling Convention Put **_pascal** or **_fortran** in the C function declaration to specify the Pascal calling convention.

Equivalent Arrays Array declarations give the number of elements. `A1[a][b]` declares a two-dimensional array in C with `a` rows and `b` columns. By default, the array's lower bound is zero. Arrays are stored by the compiler in row-major order. By default, passing arrays from C passes a pointer to the first element of the array.

String Format C stores strings as arrays of bytes and uses a null character as the end-of-string delimiter. For example, consider the string declared as follows:

```
char msg[] = "string of text"
```

The string occupies 15 bytes of memory as:

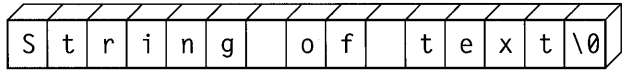


Figure 20.1 C String Format

Since `msg` is an array of characters, it is passed by reference. To pass by value, declare the string to be a member of a structure and pass the structure.

External data can be accessed directly by other modules.

External Data In C, the **extern** keyword tells the compiler that the data or function is external. You can define a static data object in a C module by defining a data object outside all functions and subroutines. Do not use the **static** keyword in C with a data object that you wish to be public.

C structures are word-aligned by default.

Structure Alignment By default, C uses word alignment (unpacked storage) for all data objects longer than one byte. This storage method specifies that occasional bytes may be added as padding, so that word and doubleword objects start on an even boundary. In addition, all nested structures and records start on a word boundary. MASM is byte-aligned by default.

When transferring .H files with H2INC, you can use the /Zp command-line option to specify structure alignment. If the /Zp option is not specified, H2INC uses word-alignment. Without H2INC, set the alignment to 2 when declaring the MASM structure, or compile the C module with /Zp1 or the MASM module with /Zp2.

Compiling and Linking Use the same memory model for both C and MASM.

Returning Values The assembler returns simple data types in registers. Table 20.2 shows the register conventions for returning simple data types to a C program.

Table 20.2 Register Conventions for Simple Return Values

Data Type	Registers
char	AL
int, short, near	AX
long, far	High-order portion (or segment address) in DX; low-order portion (or offset address) in AX

Procedures using the C calling convention and returning type **float** or type **double** store their return values into static variables. In multi-threaded programs, this could mean that the return value may be overwritten. You can avoid this by using the Pascal calling convention for multi-threaded programs so **float** or **double** values are passed on the stack.

Your procedures can also return structures.

Structures less than four bytes long are returned in DX:AX. To return a longer structure from a procedure that uses the C calling convention, you must copy the structure to a global variable and then return a pointer to that variable in the AX register (DX:AX, if you compiled in compact, large, or huge model or if the variable is declared as a far pointer).

Structures, Records, and User-Defined Data Types You can pass structures, records, and user-defined types as arguments by value or by reference.

Writing Procedure Prototypes The H2INC utility simplifies the task of writing prototypes for the C functions you want to call from MASM. The C prototype converted by H2INC into a MASM prototype allows **INVOKE** to correctly call the C function. Here are some examples of C functions and the MASM prototypes created with H2INC.

```
/* Function Prototype Declarations to Convert with H2INC */

long checktypes (
        char *name,
        unsigned char a,
        int b,
        float d,
        unsigned int *num );

my_func (float fNum, unsigned int x);

extern my_func1 (char *argv[]);

struct videoconfig _far * _far pascal my_func2 (int, scri );
```

For the C prototypes above, H2INC generates this code:

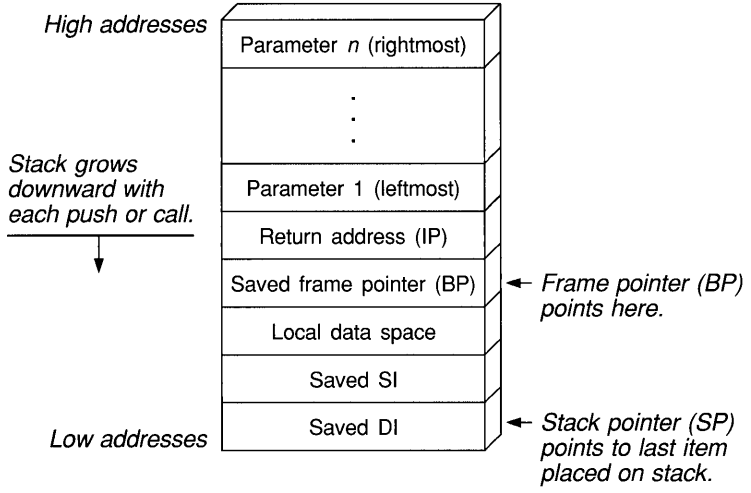
```
TYPEDEF         PROTO C :PTR SBYTE, :BYTE,
                                        :SWORD, :REAL4, :PTR WORD
checktypes      PROTO           @proto_0

@proto_1        TYPEDEF         PROTO C :REAL4, :WORD
my_func         PROTO           @proto_1

@proto_2        TYPEDEF         PROTO C :PTR PTR SBYTE
my_func1        PROTO           @proto_2

@proto_3        TYPEDEF         PROTO FAR PASCAL :SWORD, :scri
my_func2        PROTO           @proto_3
```

Example As shown in the short example below, the main module (written in C) calls an assembly routine, `Power2`.

```
#include <stdio.h>

extern int Power2( int factor, int power );

void main()
{
    printf( "3 times 2 to the power of 5 is %d\n", Power2( 3, 5 ) );
}
```

Figure 20.2 shows how functions that observe the C calling convention use the stack frame.

Near Function Call



Far Function Call

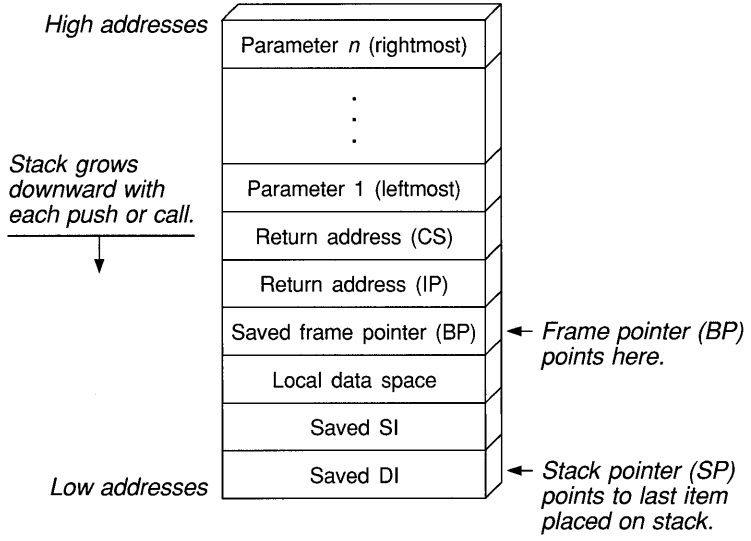


Figure 20.2 C Stack Frame

The MASM module that contains the Power2 routine looks like this:

```
        .MODEL  small, c

Power2  PROTO C factor:SWORD, power:SWORD
        .CODE

Power2  PROC  C factor:SWORD, power:SWORD
        mov     ax, factor      ; Load Arg1 into AX
        mov     cx, power       ; Load Arg2 into CX
        shl     ax, cl          ; AX = AX * (2 to power of CX)
                                ; Leave return value in AX
        ret
Power2  ENDP
        END
```

The MASM procedure declaration for the Power2 routine specifies the C *langtype* and the parameters expected by the procedure. The *langtype* specifies the calling and naming conventions for the interface between MASM and C. The routine is public by default. When the C module calls Power2, it passes two arguments, 3 and 5 by value.

The C module first defines a prototype for the MASM routine. MASM 6.0 also supports prototyping of procedures and functions. See Section 7.3.6, "Declaring Procedure Prototypes," and the examples in this section.

20.3.2 The FORTRAN/MASM Interface

This section summarizes the specific details important to calling FORTRAN procedures or receiving arguments from FORTRAN routines that call MASM routines. It includes a sample MASM and FORTRAN module. A FORTRAN procedure follows the Pascal calling convention by default. This convention passes arguments in the order listed, and the calling procedure removes the arguments from the stack. The naming convention determines that exported names are uppercase.

Compatible Data Types This list shows the FORTRAN data types that are equivalent to the MASM 6.0 data types.

FORTRAN Type	Equivalent MASM Type
CHARACTER*1	BYTE
INTEGER*1	SBYTE
INTEGER*2	SWORD
REAL*4	REAL4
INTEGER*4	SDWORD
REAL*8, DOUBLE PRECISION	REAL4

Naming Restrictions FORTRAN allows 31 characters for identifier names. A digit or an underscore cannot be the first character in an identifier name.

Argument-Passing Defaults By default, FORTRAN passes arguments by reference as far addresses if the FORTRAN module is compiled in large or huge memory model. It passes them as near addresses if the FORTRAN module is compiled in medium model. Versions of FORTRAN prior to Version 4.0 always requires large model.

The FORTRAN compiler passes an argument by value when declared with the **VALUE** attribute. This declaration can occur either in a FORTRAN **INTERFACE** block (which determines how to pass an argument) or in a function or subroutine declaration (which determines how to receive an argument).

In FORTRAN you can apply the **NEAR** (or **FAR**) attribute to reference parameters. These keywords override the default. They have no effect when they specify the same method as the default.

Changing the Calling Convention A call to a FORTRAN function or subroutine declared with the **PASCAL** or **C** attribute passes all arguments by value in the parameter list (except for parameters declared with the **REFERENCE** attribute). This change in default passing method applies to function and subroutine definitions as well as to the functions and subroutines described by **INTERFACE** blocks.

Equivalent Arrays When you declare FORTRAN arrays, you can specify any integer for the lower bound (the default is 1). The FORTRAN compiler

stores all arrays in column-major order—that is, the leftmost subscript increments most rapidly. For example, the first seven elements of an array defined as `A[3,4]` are stored as

```
A[1,1], A[2,1], A[3,1], A[1,2], A[2,2], A[3,2], A[1,3]
```

FORTRAN strings do not have an end-of-string delimiter.

String Format FORTRAN stores strings as a series of bytes at a fixed location in memory, with no delimiter at the end of the string. When passing a variable-length FORTRAN string to another language, you need to devise a method by which the target routine can find the end of the string.

Consider the string declared as

```
CHARACTER*14 MSG

MSG = 'String of text'
```

The string is stored in 14 bytes of memory like this:

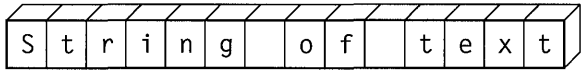


Figure 20.3 FORTRAN String Format

Strings are passed by reference. Although FORTRAN has a method for passing length, the variable-length FORTRAN strings cannot be used in a mixed-language interface because other languages cannot access the temporary variable that FORTRAN uses to communicate string length. However, fixed-length strings can be passed if the FORTRAN **INTERFACE** statement declares the length of the string in advance.

External Data FORTRAN routines can directly access external data. In FORTRAN you can declare data to be external by adding the **EXTERN** attribute to the data declaration. You can also access a FORTRAN variable from MASM if it is declared in a **COMMON** block.

A FORTRAN program can call an external assembly procedure with the use of the **INTERFACE** statement. However, the **INTERFACE** statement is not strictly necessary unless you intend to change one of the FORTRAN defaults.

Structure Alignment By default, FORTRAN uses word alignment (packed storage) for all data objects larger than one byte. This storage method specifies that occasional bytes may be added as padding, so that word and doubleword objects start on an even boundary. In addition, all nested structures and records start

on a word boundary. MASM's default is byte-alignment, so specify an *alignment* of 2 for MASM structures or use the /Zp1 option when compiling in FORTRAN.

Compiling and Linking Use the same memory model for the MASM and FORTRAN modules.

Returning Values You must use a special convention to return floating-point values, records, user-defined types, arrays, and values larger than four bytes to a FORTRAN module from an assembly procedure. The FORTRAN module creates space in the stack segment to hold the actual return value. When the call to the assembly procedure is made, an extra parameter is passed. This parameter is the last one pushed. The segment address of the return value is contained in SS.

In the assembly procedure, put the data for the return value at the location pointed to by the return value offset. Then copy the return-value offset (located at BP + 6) to AX, and copy SS to DX. This is necessary because the calling module expects DX:AX to point to the return value.

Structures, Records, and User-Defined Data Types The FORTRAN structure variable, defined with the **STRUCTURE** keyword and declared with the **RECORD** statement, is equivalent to the Pascal **RECORD** and the C **struct**. You can pass structures as arguments by value or by reference (the default).

MASM structures can be compatible with FORTRAN COMPLEX types.

The FORTRAN types **COMPLEX*8** and **COMPLEX*16** are not directly implemented in MASM. However, you can write structures that are equivalent. The type **COMPLEX*8** has two fields, both of which are four-byte floating-point numbers; the first contains the real component, and the second contains the imaginary component. The type **COMPLEX** is equivalent to the type **COMPLEX*8**.

The type **COMPLEX*16** is similar to **COMPLEX*8**. The only difference is that each field of the former contains an eight-byte floating-point number.

A FORTRAN **LOGICAL*2** is stored as a one-byte indicator value (1=true, 0=false) followed by an unused byte. A FORTRAN **LOGICAL*4** is stored as a one-byte indicator value followed by three unused bytes. The type **LOGICAL** is equivalent to **LOGICAL*4**, unless **$STORAGE:2** is in effect.

To pass or receive a FORTRAN **LOGICAL** type, declare a MASM structure with the appropriate fields.

Varying Number of Arguments In FORTRAN, you can call routines with a variable number of arguments by including the **VARYING** attribute in your interface to the routine, along with the **C** attribute. You must use the **C** attribute because a variable number of arguments is possible only with the **C** calling convention. The **VARYING** attribute prevents FORTRAN from enforcing a matching number of parameters.

LOCNEAR and LOCFAR determine addresses.

Pointers and Addresses FORTRAN programs can determine near and far addresses with the **LOCNEAR** and **LOCFAR** functions. Store the result as **INTEGER*2** (with the **LOCNEAR** function) or as **INTEGER*4** (with the **LOCFAR** function). If you pass the result of **LOCNEAR** or **LOCFAR** to another language, be sure to pass by value.

Example In the following example, the FORTRAN module calls an assembly procedure that calculates `A*2^B`, where `A` and `B` are the first and second parameters, respectively. This is done by shifting the bits in `A` to the left `B` times.

```
INTERFACE TO INTEGER*2 FUNCTION POWER2(A, B)
INTEGER*2 A, B
END

PROGRAM MAIN
INTEGER*2 POWER2
INTEGER*2 A, B
A = 3
B = 5
WRITE (*, *) '3 TIMES 2 TO THE B OR 5 IS ',POWER2(A, B)
END
```

To understand how to write the assembly procedure, consider how the parameters are placed on the stack, as illustrated in Figure 20.4.

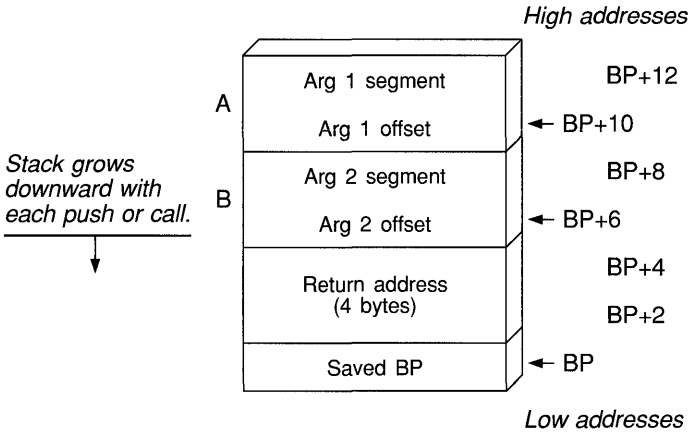


Figure 20.4 FORTRAN Stack Frame

Figure 20.4 assumes that the FORTRAN module is compiled in large model. If you compile the FORTRAN module in medium model, then each argument is passed as a two-byte, not four-byte, address. The return address is four bytes long because procedures called from FORTRAN must always be **FAR**.

The assembler code looks like this:

```
        .MODEL LARGE, FORTRAN

Power2  PROTO  FORTRAN, factor:FAR PTR SWORD, power:FAR PTR SWORD

        .CODE

Power2  PROC   FORTRAN, factor:FAR PTR SWORD, power:FAR PTR SWORD

        les     bx, factor
        mov     ax, ES:[bx]
        les     bx, power
        mov     cx, ES:[bx]
        shl     ax, cl
        ret
Power2  ENDP        END
```

20.3.3 The Basic/MASM Interface

This section explains how to call MASM procedures or functions from Basic and how to receive Basic arguments for the MASM procedure. Pascal is the default naming and calling convention, so all lowercase letters are converted to uppercase. Routines defined with the **FUNCTION** keyword return values, but routines defined with **SUB** do not. Basic **DEF FN** functions and **GOSUB** routines cannot be called from another language.

The information provided pertains to Microsoft's Basic and QuickBasic compilers. Differences between the two compilers are noted when necessary.

Compatible Data Types The list shows the Basic data types that are equivalent to the MASM 6.0 data types.

Basic Type	Equivalent MASM Type
STRING*1	WORD
INTEGER (X%)	SWORD
SINGLE (X!)	REAL4
LONG (X&), CURRENCY	SDWORD
DOUBLE (X#)	REAL8

Naming Conventions Basic recognizes up to 40 characters of a name. In the object code, Basic also drops any of its reserved characters: %, &, !, #, @, &.

Argument-Passing Defaults Basic can pass data in several ways and can receive it by value or by near reference.

By default, Basic arguments are passed by near reference as two-byte addresses. To pass a near address, pass only the offset; if you need to pass a far address, pass the segment and offset separately as integer arguments. Pass the segment address first, unless you have specified C compatibility with the **CDECL** keyword.

Basic passes each argument in a call by far reference when **CALLS** is used to invoke a routine. You can also use **SEG** to modify a parameter in a preceding **DECLARE** statement so that Basic passes that argument by far reference.

To pass a Basic argument by value, apply the **BYVAL** keyword to the argument in the **DECLARE** statement. Arrays and user-defined types cannot be passed by value.

```
DECLARE SUB Test(BYVAL a%, b%, SEG c%)

CALL Test(x%, y%, z%)

CALLS Test(x%, y%, z%)
```

The **CALL** statement above passes the first argument (`a%`) by value, the second argument (`b%`) by near reference, and the third argument (`c%`) by far reference. The statement `CALLS Test2(x%, y%, z%)` passes each argument by far reference.

Changing the Calling Convention Including the **CDECL** keyword in the Basic **DECLARE** statement enables the C calling and naming convention. This also allows a call to a MASM procedure with a varying number of arguments.

Equivalent Arrays The **DIM** statement sets the number of dimensions for a Basic array and also sets the array's maximum subscript value. In the array declaration `DIM x(a,b)`, the upper bounds (the maximum number of values possible) of the array are `a` and `b`. The default lower bound is 0. The default upper bound for an array subscript is 10.

Basic stores arrays in column-major order.

The default for column storage in Basic is column-major order, as in FORTRAN. For an array defined as `DIM Arr%(3,3)`, reference the last element as `Arr%(3,3)`. The first five elements of `Arr (3,3)` are

```
Arr(0,0), Arr(1,0), Arr(2,0), Arr(0,1), Arr(1,1)
```

When you pass an array from Basic to a language that expects arrays to be stored in row-major order, use the command-line option /R when compiling the Basic module.

Most Microsoft languages permit you to reference arrays directly. Basic uses an array descriptor, however, which is similar in some respects to a Basic string

descriptor. The array descriptor is necessary because Basic may shift the location of array data in memory; Basic handles memory allocation for arrays dynamically.

To pass arrays to MASM, you need to follow several rules.

A reference to an array in Basic is really a near reference to an array descriptor. Array descriptors are always in DGROUP, even though the data may be in far memory. Array descriptors contain information about type, dimensions, and memory locations of data. You can safely pass arrays to MASM routines only if you follow three rules:

- Pass the array's address by applying the **VARPTR** function to the first element of the Basic array and passing the result by value. To pass the far address of the array, apply both the **VARPTR** and **VARSEG** functions and pass each result by value. The receiving language gets the address of the first element and considers it to be the address of the entire array. It can then access the array with its normal array-indexing syntax.
- If the MASM routine that receives the array makes a call back to Basic, then the location of the array data may change, and the address that was passed to the routine will be meaningless.
- Basic can pass any member of an array by value. When passing individual array elements, the above restrictions do not apply.

You can apply **LBOUND** and **UBOUND** to a Basic array to determine lower and upper bounds, and then pass the results to another routine. This way, the size of the array does not need to be determined in advance.

String Format Strings are stored in Basic as four-byte string descriptors, as shown below. The first field of the string descriptor contains a two-byte integer indicating the length of the actual string text. The second field contains the address of this text.

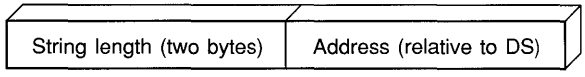


Figure 20.5 Basic String Descriptor Format

Basic's string descriptors are not compatible with the string formats of other languages.

This address is an offset into the default data area and is assigned by Basic's string-space management routines. These management routines need to be available to reassign this address whenever the length of the string changes, yet these management routines are available only to Basic. Therefore, your MASM procedure should not alter the length of a Basic string.

Prior to version 7.0 of the Microsoft Basic Compiler, there are two ways to pass strings:

1. Pass the address of the Basic string data to the other language
2. Mimic the form of the Basic string descriptor in the other language, then use that to access the string as Basic would access one of its own strings

NOTE Version 7.0 of the Microsoft Basic Compiler provides new functions that access the string descriptors and allow simplified string passing between Basic and other languages. Follow the instructions in the Basic documentation.

The routine that receives the string must not call any Basic routine. If it does, Basic's string-space management routines may change the location of the string data without warning.

The **SADD** function returns the address of a specified string variable. Basic should pass the result of the **SADD** function by value. Bear in mind that the string's address, not the string itself, is passed by value. This amounts to passing the string itself by reference. The Basic module passes the string address, and the other module receives the string address. The address returned by **SADD** is declared as type **INTEGER** but is actually equivalent to a C near pointer or Pascal **ADR** variable.

To return the far address of a string variable, version 7.0 (or later) of Basic provides the **SSEGADD** function. See your Basic documentation.

MASM can access data declared with a COMMON statement.

External Data Variables can be global to modules in a Basic program by declaring them with the **COMMON** statement. Global variables do not require any additional declarations to be used by MASM procedures.

Structure Alignment Basic packs user-defined types. For MASM structures to be compatible, select byte-alignment.

Use medium memory model with Basic.

Compiling and Linking Always assemble the MASM module with medium model when you are linking to Basic. If you are listing other libraries on the LINK command line, specify Basic libraries first. (There are differences between the QBX and command-line compilation. See your Basic documentation.)

Returning Values Basic follows the usual convention of returning values in AX or DX:AX. If the value is not floating point, an array, or a structured type, or if it is less than 4 bytes long, then the two-byte integers should be returned from the MASM procedure in AX and four-byte integers should be returned in DX:AX. For all other types, return the near offset in AX.

User-Defined Data Types The Basic **TYPE** statement defines structures composed of individual fields. These types are equivalent to the C **struct**,

FORTRAN record (declared with the **STRUCTURE** keyword), and Pascal **Record** types.

You can use any of the Basic data types except variable-length strings or dynamic arrays in a user-defined type. Once defined, Basic types can be passed only by reference.

Varying Number of Arguments You can vary the number of arguments in a Basic routine only when you use **CDECL** to change the calling convention. To call a function with a varying number of arguments, you also need to suppress the type-checking that normally forces a call to be made with a fixed number of arguments. In Basic, you can remove this type checking by omitting a parameter list from the **DECLARE** statement.

Pointers and Addresses **VARSEG** accesses a variable's segment address, and **VARPTR** accesses a variable's offset address. The values returned by these intrinsic Basic functions should then be passed or stored as ordinary integer variables. Pass segment addresses first unless your procedure specifies the **cdecl** calling convention. If you pass them to MASM procedures, pass by value. Otherwise you are attempting to pass the address of the address, rather than the address itself.

Example This example calls the `Power2` procedure in the MASM 6.0 module.

```
DEFINT A-Z

DECLARE FUNCTION Power2 (A AS INTEGER, B AS INTEGER)
PRINT "3 times 2 to the power of 5 is ";
PRINT Power2(3, 5)

END
```

The first argument, `A`, is higher in memory than `B` because Basic pushes arguments in the same order in which they appear.

Figure 20. 6 shows how the arguments are placed on the stack:

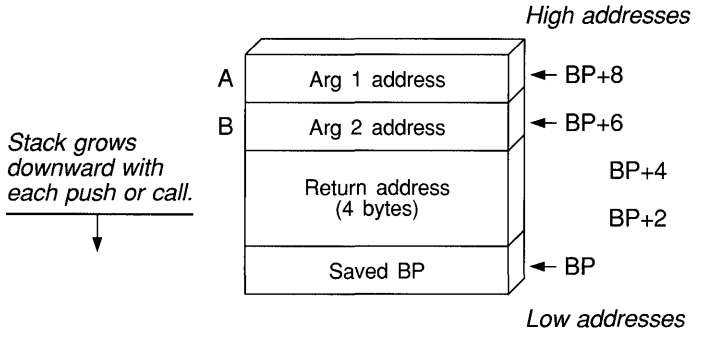


Figure 20.6 Basic Stack Frame

The assembly procedure can be written as follows:

```
        .MODEL  medium

Power2  PROTO   PASCAL, Factor:PTR WORD, Power:PTR WORD
        .CODE
Power2  PROC    PASCAL, Factor:PTR WORD, Power:PTR WORD

        mov     bx, WORD PTR Factor ; Load Factor into
        mov     ax, [bx]            ;  AX
        mov     bx, WORD PTR Power  ; Load Power into
        mov     cx, [bx]            ;  CX
        shl     ax, cl              ; AX = AX * (2 to power
                                    ;  of CX)
        ret
Power2  ENDP
        END
```

Note that each parameter must be loaded in a two-step process because the address of each is passed rather than the value. The return address is four bytes long because procedures called from Basic must be **FAR**.

20.3.4 The Pascal/MASM Interface

This section summarizes details important to calling Microsoft Professional Pascal, Version 4.0, routines from MASM and MASM routines from Pascal. It includes information on parameters and data types specific to Pascal source modules. The information in this section does not apply to QuickPascal (see Section 20.3.5 for that).

The Pascal calling convention—the default—places arguments on the stack in the same order in which they appear in the Pascal source code. The first

argument is highest in memory because it is also the first argument to be placed on the stack, and the stack grows downward. The default naming convention exports names in uppercase.

Compatible Data Types This list shows the Pascal types that are equivalent to the MASM 6.0 data types.

Pascal Type	Equivalent MASM Type
BYTE, CHAR, BOOLEAN	BYTE
WORD	WORD
INTEGER2	SWORD
REAL, REAL4	REAL4
INTEGER4	SDWORD
REAL8	REAL8

Naming Restrictions Microsoft Pascal Version 4.0 recognizes only the first 8 characters of any name, while the assembler recognizes the first 256. Names used publicly with Pascal should not be longer than 8 characters.

The default for Pascal is passing by value.

Argument-Passing Defaults By default, Pascal arguments are passed by value, but they can be passed by near reference when declared as **VAR** or **CONST** and as far reference when declared as **VARS** or **CONSTS**. A **VARS** or **CONSTS** argument includes both a two-byte segment address and a two-byte offset with the segment pushed first.

Pascal arguments are also passed by near (or far) reference when the **ADR** (or **ADS**) of a variable, or a pointer to a variable, is passed by value. In other words, the address of the variable is first determined. Then this address is passed by value.

Pascal routines can use the C calling convention.

Changing the Calling Convention To use the C calling convention from Pascal, type `[C]` at the end of the declarations before the semicolon, as shown:

```
Procedure MyProc ( x : integer ) [C]; EXTERN;
```

Equivalent Arrays The lower bound for Pascal arrays can be any integer. Subscripts vary in row-major order.

String Format Pascal has two types of strings, each of which uses a different format: a fixed-length type **STRING** and the variable-length type **LSTRING**.

The format used for **STRING** is identical to that of the FORTRAN string.The format of an **LSTRING** stores the length in the first byte. For example, consider an **LSTRING** declared as

```
VAR Msg:LSTRING(14);
Msg := 'String of text'
```

Pascal strings store the string length in the first byte.

The string is stored in 15 bytes of memory. The first byte indicates the length of the string text. The remaining bytes contain the string text itself:

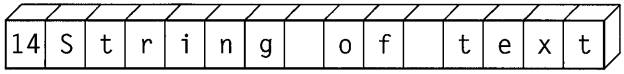


Figure 20.7 Pascal String Format

The Pascal data type **LSTRING** is not compatible with the formats used by the other languages. You can pass an **LSTRING** indirectly, however, by first assigning it to a **STRING** variable. Pascal supports such assignments by performing a conversion of the data.

Pascal passes an additional two-byte argument that indicates string length whenever you pass an argument of type **STRING** or **LSTRING**. To suppress the passing of this additional argument, declare a fixed-length type.

External Data Pascal routines can directly access external data. You can declare data as external by adding the **EXTERN** attribute to the data declaration.

Structure Alignment Pascal uses word alignment (unpacked storage) for all data objects larger than one byte. In addition, all nested structures and records start on a word boundary. You can turn on packing for Pascal modules, or you can define structures in MASM to have 2 for their alignment value.

Compiling and Linking Always use large model for the MASM module when linking with Pascal.

Returning Values Functions that return **REAL**, **REAL4**, or **REAL8** values use the long return method; that is, the caller passes an additional, hidden offset of a temporary stack variable that will receive the result.

INVOKE cannot handle long return values directly, but you can add an additional parameter to the prototype for the Pascal procedure. For example, a prototype for a Pascal procedure that expects an **SWORD** argument looks like this:

```
PascalProc PROTO Pascal arg1:SWORD, PtrRetVal:NEAR PTR
```

Before calling the Pascal procedure with **INVOKE**, allocate space on the stack with

```
add     sp, space
mov     cx, sp
INVOKE  PascalProc, ax, cx
.
.
.
sub     sp, space
```

These statements place the address of the allocated space in CX.

Since calls to Pascal procedures must be made from within a MASM procedure previously called from the Pascal module, an alternative way to handle a long return value is to create a local variable to receive the return value. This example illustrates this technique:

```
Proc1   PROC    arg1:SWORD
        LOCAL   RetVal:REAL8
        INVOKE  PascalProc, ax, ADDR RetVal
```

To return structures from MASM using the Pascal calling convention, the calling program allocates space for the return value on the stack and passes a pointer (as a hidden argument) to the location where the return value is to be placed. Copy the MASM structure into the location pointed to by the hidden argument and return the pointer to that location in the AX register (or DX:AX for far data models).

Use the C and VARYING attributes for routines that will receive a variable number of arguments.

Varying Number of Arguments In Pascal, you can call routines with a variable number of arguments by including the **VARYING** attribute in your interface to the routine, along with the **C** attribute. You must use the **C** attribute for the Pascal routine, because a variable number of arguments is possible only with the C calling convention.

Each time you call the routine, you will not be required to pass the same number of arguments as are declared in the interface to the routine. However, each actual argument that you pass will be type-checked against whatever formal parameters you may have declared.

Structures, Records, and User-Defined Types You can pass Pascal structures, records, and user-defined types as arguments by value or by reference depending on the size of the data.

Pointers and Addresses The Pascal **ADR** and **ADS** types are equivalent to the C near and far pointers. You can pass **ADR** and **ADS** variables as **ADRMEM** or **ADSMEM**.

Example This example shows the `Power2` procedure as it is called by Pascal.

```
Program Asmtest( input, output );
function Power2( a:integer; b:integer ): integer; extern;
begin
    writeln( '3 times 2 to the power of 5 is ', Power2( 3, 5 ) );
end.
```

To understand how to write the assembly procedure, consider how the arguments are placed on the stack, as illustrated in Figure 20.8.

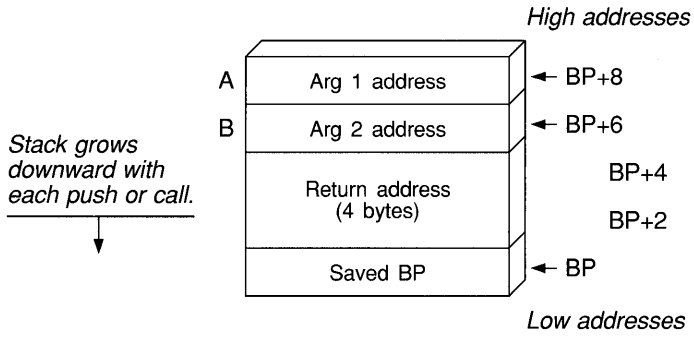


Figure 20.8 Pascal Stack Frame

The first argument, 3, is higher in memory than 5 because Pascal pushes arguments in the same order they appear. Both arguments are passed by value.

The MASM 6.0 module can be written as follows:

```
        .MODEL  medium, PASCAL
        .386
Power2  PROTO PASCAL  factor:WORD, power:WORD
        .CODE

Power2  PROC    factor:WORD, power:WORD

        mov     ax, factor      ; Load Factor into AX
        mov     cx, power       ; Load Power into CX
        shl     ax, cl          ; AX = AX * (2 to power of CX)
        ret                     ; Leave return value in AX

Power2  ENDP
        END
```

The AX and CX registers can be loaded directly because the arguments are passed by value.

20.3.5 The QuickPascal/MASM Interface

The QuickPascal implementation of Pascal uses several data types and defaults that are different from version 4.0 of the Microsoft Pascal compiler. This section summarizes the techniques for calling MASM procedures from QuickPascal and for accessing QuickPascal data and routines from MASM. The following information also applies to other compilers that are compatible with QuickPascal.

The Pascal calling convention pushes arguments in the order listed and exports identifiers in uppercase.

Compatible Data Types This list gives the QuickPascal data types that are equivalent to the MASM 6.0 data types.

QuickPascal Type	Equivalent MASM Type
Char, Boolean	**BYTE**
Byte, ShortInt	**SBYTE**
Word	**WORD**
Integer	**SWORD**
Single	**REAL4**
LongInt	**SDWORD**
Real	**FWORD**
Comp, Double	**REAL8**
Extended	**REAL10**

Naming Restrictions The first 63 characters of QuickPascal identifiers are significant. Identifiers are not case sensitive and the first character must be a letter or an underscore character (_). Digits can be used in the indentifier's name.

By default, Pascal passes arguments by value.

Register Preservation Procedures called by QuickPascal must preserve the values of the BP, SP, SS, and DS registers. BP and SP are preserved by standard entry and exit code. If you need to alter DS or SS, you must preserve the current values.

Argument-Passing Defaults When an argument is passed by value, QuickPascal takes different actions depending on the data type and size. This

convention for value parameters is not shared by other Microsoft high-level languages, which always push arguments passed by value directly onto the stack.

- Enumerated type arguments are passed as unsigned bytes if the enumeration has 256 or fewer values; otherwise they are passed as an unsigned word.
- Types **Single** (4 bytes), **Real** (6 bytes), **Double** (8 bytes), **Comp** (8 bytes), and **Extended** (10 bytes) are passed on the stack.
- Pointer types are passed as doublewords. The segment is pushed before the offset so the offset is lowest in memory.
- If the value parameter is **Char**, **Boolean**, any pointer, any **Integer**, or any floating-point type, QuickPascal pushes the argument onto the stack.
- If the argument is a string or set type, QuickPascal passes a pointer to the data. This action is really the same as passing by reference. If you want to avoid any possibility of altering the data, make a temporary copy of the data and then work with the temporary data.
- If the argument is an array or record type and if it is not more than four bytes long, QuickPascal pushes the variable directly onto the stack. Otherwise, it pushes a pointer to the data.

When an argument is passed by reference, QuickPascal pushes a four-byte pointer to the data. The offset portion of the pointer is always pushed first and is therefore higher in memory.

Changing the Calling Convention QuickPascal supports only the Pascal calling convention.

Equivalent Arrays Arrays are stored in row-major order. Arrays (and records with one, two, or four bytes) are passed directly on the stack.

String Format In the **STRING** format, the first byte of the string stores the string length. In the **CSTRING** format, there is no length indicator and there is a terminating null byte.

External Data You cannot declare public data in a data segment of the MASM module for the QuickPascal module to reference. QuickPascal can use private, static data in MASM modules; however, the data declared in the MASM module must be initialized with **?**. MASM can reference data in a QuickPascal unit.

Structure Alignment QuickPascal records are byte-aligned.

Compiling and Linking For QuickPascal to access an assembled MASM module (named MASMMOD.OBJ), include this line at the beginning of your QuickPascal program:

```
{$L QPEX.OBJ}
```

You do not need to use LINK or any other utility to produce executable files.

QuickPascal sets up the link to the MASM module by copying the MASM object file into the program and changing the file into its own internal object-code format. The disk-based object file is left unchanged.

Returning Values To return a value to a QuickPascal module, follow these conventions:

- For a **String**, **CSTRING**, **Comp**, or floating-point type other than **Real**, QuickPascal passes an additional argument. This argument is pushed first and is a pointer to a temporary storage location. The function must place the result of the function in this location and not remove this pointer.
- For ordinal types, (including **Char**, **Boolean**, and any integer), place the result in AL if one byte, in AX if two bytes, and in DX:AX if a doubleword (in which DX holds the most-significant byte).
- QuickPascal does not support functions that return array or record types. However, you can set the value of an array or record if QuickPascal passes it as a **VAR** parameter.

Example This example includes a Pascal program to call the assembly module `Power2`.

```
{$L QPEX.OBJ}

program  Asmtest( input, output );
function POWER2( factor:integer; power:integer ): integer; external;
begin
    writeln( '3 times 2 to the power of 5 is ', POWER2( 3, 5 ) );

end.
```

This is the assembly module to be called from the Pascal program.

```
Power2  PROTO PASCAL  factor:WORD, power:WORD

CODE    SEGMENT WORD PUBLIC
        ASSUME  CS:CODE
```

```
Power2  PROC PASCAL   factor:WORD, power:WORD

        mov     ax, factor      ; Load factor into AX
        mov     cx, power       ; Load power into CX
        shl     ax, cl          ; AX = AX * (2 to power of CX)
                                ; Leave return value in AX
        ret
Power2  ENDP

CODE    ENDS
        END
```

You cannot use the /Zi command-line option when assembling a module to be called from QuickPascal.

20.4 Related Topics in Online Help

Other information available online which relates to topics in this chapter is listed below:

Topic	Access
/NOI Linker option	From the list of Utilities on the "Microsoft Advisor Contents" screen, choose "LINK"; then choose "LINK Options"
PROC, LOCAL, INVOKE, LABEL	From the "MASM 6.0 Contents" screen, choose "Directives"; then choose "Procedures and Code Labels"
H2INC	From the "ML Contents" screen, choose "H2INC Utility"
STRUCT	From the "MASM 6.0 Contents" screen, choose "Directives"; then choose "Complex Data Types"
EXTERN, EXTERNDEF, PUBLIC	From the "MASM 6.0 Contents" screen, choose "Directives"; then choose "Scope and Visibility"
USES, RET, VARARG	From the MASM Index, select **PROC**

Appendixes

Appendix A

Differences between MASM 6.0 and 5.1

Version 6.0 of the Microsoft Macro Assembler contains significant changes over previous versions. Some of these changes include:

- An environment called Programmer's WorkBench (PWB) from which you can write, edit, debug, and execute code
- Expanded functionality for structures, unions, and type definitions
- New directives for generating loops and decision statements, and for declaring and calling procedures
- Simplified methods for applying public attributes to variables and routines in multiple-module programs
- Enhancements for writing and using macros
- Flat-model support for OS/2 version 2.0 and new instructions for the 80486 processor

Section A.1 describes the new features of MASM 6.0. The appendix does not go into great detail about the new features, but it does provide references to the information presented elsewhere in the MASM 6.0 documentation. For full explanations and coding examples, see the documentation listed in the cross-references.

Section A.2 discusses compatibility with MASM 5.1. To get your MASM 5.1 code running under MASM 6.0 using **OPTION M510** (or the /Zm command-line option), see Section A.2.1, "Rewriting Code for Compatibility." To remove **OPTION M510** (or /Zm) from your code, see Section A.2.2, "Using the OPTION Directive."

A.1 New Features of Version 6.0

MASM 6.0 contains many new features. This section briefly describes each one. Some new features, such as the new behavior of structures, also allow you to select compatibility options. These features are also discussed in Section A.2, "Compatibility between MASM 5.1 and 6.0."

A.1.1 The Assembler, Environment, and Utilities

Most of the executable files provided with MASM 6.0 are new or revised. For a complete list of these files, read the PACKING.LST file on the distribution disk. The book *Installing and Using the Professional Development System* also provides more information about setting up the environment, assembler, and online help system.

The Assembler The macro assembler, now named ML.EXE, is capable of assembling and linking in one step. The command-line options are completely new. For example, the new /EP option produces a listing file during the assembler's first pass. Command-line options now are case-sensitive and must be separated by spaces.

For backward compatibility with version 5.1 makefiles, a MASM.EXE utility is included. When you run MASM.EXE, it translates version 5.1 command-line options to the new version 6.0 command-line options and calls ML.EXE. See the *Microsoft Macro Assembler Reference* for details.

H2INC H2INC converts C include files to MASM include files. It translates data structures and declarations but does not translate executable code. For more information, see Chapter 16, "Converting C Header Files to MASM Include Files."

NMAKE NMAKE is the new version of the MAKE utility. NMAKE provides new functionality in evaluating target files and more flexibility with macros and command-line options. For more information, see Chapter 10, "Managing Projects with NMAKE."

Integrated Environment PWB is an integrated environment for writing, developing, and debugging programs. See *Installing and Using* for information on using PWB, and the *Reference* for information on command-line options. See also Chapter 14, "Customizing the Microsoft Programmer's WorkBench," and Chapter 15, "Debugging Assembly-Language Programs with CodeView."

Online Help The Microsoft Advisor online help system has been added to MASM 6.0. It provides a vast database of online help about all aspects of MASM, including the syntax and timings for processor and coprocessor instructions, MASM directives, command-line options, and support programs such as LINK and PWB.

See *Installing and Using*, Chapter 4, for information on how to set up the help system. You can invoke the help system from within PWB or from the QuickHelp program (QH).

HELPMAKE You can use the HELPMAKE utility to create additional help files from ASCII text files, allowing you to customize the online help system. For more information, see Chapter 11, "Creating Help Files with HELPMAKE."

Other Programs MASM 6.0 contains the most recent versions of LINK, LIB, BIND, CodeView, and the mouse driver. The CREF program is not included in MASM 6.0. The Source Browser provides the information that CREF provided under MASM 5.1. For more information on the source browser, see Chapter 3 of *Installing and Using the Professional Development System* or online help.

A.1.2 Segment Management

This section lists the changes and additions to memory-model and operating-system support as well as to directives that relate to these topics.

New Predefined Symbols The following new predefined symbols (also called predefined equates) provide information about simplified segments:

Predefined Symbol	Value
@stack	**DGROUP** for near stacks, **STACK** for far stacks
@Interface	Information about language parameters
@Model	Information about the current memory model
@Line	The source line in the current file
@Date	The current date
@FileCur	The current file
@Time	The current time
@Environ	The current environment variables

For more information, see Section 1.2.3, "Predefined Symbols," or online help.

Enhancements to the ASSUME Directive MASM automatically generates **ASSUME** values for the code segment register (CS) when a segment is opened. It is no longer necessary to include lines such as

```
ASSUME CS:MyCodeSegment
```

in your programs. In addition, the **ASSUME** directive can now include **ERROR**, **FLAT**, or *register*:*type*. Generating **ASSUME** values for the code segment register CS to be other than the current segment or group is no longer valid.

For more information, see Sections 2.3.3, "Setting the ASSUME Directive for Segment Registers," and 3.3.2, "Defining Register Types with ASSUME."

Relocatable Offsets For compatibility with Windows programs, the new **LROFFSET** operator can calculate a relocatable offset, which is resolved by the loader at run time. See online help for details.

Flat Model In the flat memory model (available only in version 2.0 of OS/2), segments may be as large as four gigabytes because offsets contain 32 bits instead of 16. Segments are limited to 64K in all other memory models supported by DOS and earlier versions of OS/2. Version 2.0 of OS/2 runs only on 80386/486 processors. For more information about memory models, see Section 2.2.1, "Defining Basic Attributes with .MODEL."

Operating Systems Support Specifying the new **OS_OS2** or **OS_DOS** keywords in the **.MODEL** statement allows the new **.STARTUP** directive to generate start-up code appropriate for the language and operating system. The new **.EXIT** directive generates the appropriate exit code.

Section 2.2.1, "Defining Basic Attributes with .MODEL," provides more information on specifying an operating system. Also see Section 2.2.6, "Starting and Ending Code with .STARTUP and .EXIT."

A.1.3 Data Types

MASM 6.0 introduces an entirely new concept of data typing for assembly language. This section summarizes new and changed features relating to data declarations in MASM 6.0.

Defining Typed Variables You can now use the type names as directives to define variables. Initializers are unsigned by default. The following are equivalent:

```
var1    DB      25
var1    BYTE    25
```

Signed Types You can use the new **SBYTE**, **SWORD**, and **SDWORD** directives to declare signed data. See Section 4.1.1, "Allocating Memory for Integer Variables."

Floating-Point Types MASM 6.0 also introduces new directives for declaring floating-point variables, **REAL4**, **REAL8**, and **REAL10**. See Section 6.1.1, "Declaring Floating-Point Variables and Constants," for information on these new type directives.

Qualified Types MASM 6.0 allows type definitions to include distance and language type attributes. Procedures, procedure prototypes, and external declarations allow the type to be specified as a qualified type. Section 1.2.6, "Data Types," gives a complete description of qualified types.

Structures Structures have changed in several ways:

- Structures can be nested.
- The names of structure fields need not be unique. As a result, references to field names must be qualified.
- Initialization of structure variables can continue over multiple lines as long as the final noncomment character in the line is a comma.
- Curly braces and angle brackets are equivalent.

For example, this code works in MASM 6.0:

```
SCORE           STRUCT
  team1         BYTE    10 DUP (?)
  score1        BYTE    ?
  team2         BYTE    10 DUP (?)
  score2        BYTE    ?
 SCORE          ENDS

 first   SCORE  {"BEARS", 20,        ; This comment is allowed.
                 "CUBS",  10 }

        mov     al, [bx].score.team1 ; Field name must be qualified
                                     ;  with structure name.
```

You can use **OPTION OLDSTRUCTS** or **OPTION M510** to enable MASM 5.1 behavior for structures. See Section A.2, "Compatibility between MASM 5.1 and 6.0." For more information on structures and unions, see Section 5.2.

Unions MASM 6.0 allows the definition of unions. Unions differ from structures in that all field initializers occupy the same data space. The new **UNION** directive defines these variables. For more information, see Section 5.2, "Structures and Unions."

Types Defined with TYPEDEF The new **TYPEDEF** directive defines a type for use later in the program. It is most useful for defining pointer types. For more information, see Sections 1.2.6, "Data Types," and 3.3.1, "Defining Pointer Types with TYPEDEF."

Names of Identifiers The names of identifiers in MASM 6.0 can be up to 247 characters long, and all the characters are significant. In previous versions of MASM (or if **OPTION M510** is enabled), names are significant to 31 characters only. For more information on identifiers, see Section 1.2.2, "Identifiers." For more information on the **OPTION** directive, see Section 1.3.2, "Using the OPTION Directive."

Multiple-Line Initializers In MASM 6.0, a comma at the end of a line implies that the line continues. For example, the following code is legal in MASM 6.0:

```
longstring      BYTE    "This string ",
                         "continues over two lines."
bitmasks        BYTE    80h, 40h, 20h, 10h,
                        08h, 04h, 02h, 01h
```

For more information, see Section 1.2.8, "Statements."

Comments in Extended Lines Earlier versions of MASM allow a backslash (\ ) as the line-continuation character if it is the last nonspace character in the line. MASM 6.0 permits a comment to follow the backslash.

Determining Size and Length of Data Labels The new **LENGTHOF** operator returns the number of data items allocated for a data label. MASM 6.0 also has a new **SIZEOF** operator. When applied to a type, the **SIZEOF** operator returns the size attribute of the type expression. When applied to a data label, **SIZEOF** returns the number of bytes used by the initializer in the label's definition. In this case, **SIZEOF** for a variable equals the number of bytes in the type multiplied by **LENGTHOF** for the variable.

The **LENGTH** and **SIZE** operators have been retained for backward compatibility. See "Length and Size of Labels with OPTION M510" in Section A.2.2 for the behavior of **SIZE** under **OPTION M510**, and see "LENGTH Operator Applied to Record Types" in Section A.2.1.2 for obsolete behavior with the **LENGTH** operator.

For information on **LENGTHOF** and **SIZEOF**, see Section 5.1.1, "Declaring and Referencing Arrays," Section 5.1.2, "Declaring and Initializing Strings," Section 5.2.1, "Declaring Structure and Union Variables," and Section 5.3.2, "Defining Record Variables."

HIGHWORD and LOWWORD Operators These new operators return the high and low words for the 32-bit operand given. They are similar to the **HIGH** and **LOW** operators of MASM 5.1 except that **HIGHWORD** and **LOWWORD** can take only constants as operands, not relocatables (labels).

PTR and CodeView In MASM 5.1, the **PTR** operator, when applied to a data initializer, specifies what information should be generated by CodeView.

Semantically using **PTR** in this manner is still valid, but this does not affect CodeView typing. Defining pointers with the **TYPEDEF** directive allows CodeView to generate correct information. See Section 3.3.1, "Defining Pointer Types with TYPEDEF."

A.1.4 Procedures, Loops, and Jumps

Significant changes have been made for procedure and jump handling in MASM 6.0. The new functionality closely resembles high-level-language implementations of the procedure calls. MASM now generates the code to correctly handle argument passing, to check type compatibility between parameters and arguments, and to process a variable number of arguments. MASM 6.0 can also handle jumps intelligently and optimize the coding according to the distance from the target.

Function Prototypes and Calls The **PROTO** directive prototypes a function, which enables type-checking and type conversion of arguments if the function is called with **INVOKE**. For more information, see Section 7.3.6, "Declaring Procedure Prototypes."

The new **INVOKE** directive calls a procedure and correctly passes the arguments according to the prototype. For more information, see Section 7.3.7, "Calling Procedures with INVOKE."

You can also use the new **VARARG** keyword to pass a variable number of arguments to a procedure with **INVOKE**. See Section 7.3.3, "Declaring Parameters with the PROC Directive."

The **ADDR** keyword is also new. When used with **INVOKE**, it changes an expression to an address expression (for passing by reference instead of by value). See Section 7.3.7, "Calling Procedures with INVOKE."

High-Level Flow-Control Constructions MASM 6.0 contains several new directives that generate code for loops and decisions depending on the status of a conditional statement. The conditions are tested at run time rather than at assembly time.

The new directives are **.IF**, **.ELSE**, **.ELSEIF**, **.REPEAT**, **.UNTIL**, **.UNTILCXZ**, **.WHILE**, and **.ENDW**. MASM 6.0 also implements the associated **.BREAK** and **.CONTINUE** directives to use in loops and **if** statements and the binary operators used in the C language to form binary expressions.

For more information, see Section 7.2, "Loops," and Section 7.1.2.6, "Decision Directives."

Automatic Optimization for Unconditional Jumps MASM 6.0 automatically determines the smallest encoding for direct unconditional jumps. See Section 7.1.1, "Unconditional Jumps."

Automatic Lengthening for Conditional Jumps If a conditional jump requires a distance other than **SHORT**, MASM automatically generates the necessary comparison and unconditional jump to the destination. See Section 7.1.2, "Conditional Jumps."

User-Defined Stack Frame Setup and Cleanup The code generated following a **PROC** statement—a prologue—sets up the stack for parameters and local variables. The epilogue code handles stack cleanup. MASM 6.0 allows the implementation of user-defined prologues and epilogues with macros and the **OPTION** directive. See Section 7.3.8, "Generating Prologue and Epilogue Code."

A.1.5 Simplifying Multiple-Module Projects

Previous versions of MASM require that you declare data and routines used in more than one module both public and external by using the **PUBLIC** and **EXTRN** directives in the appropriate modules. With MASM 6.0, you can now use a single directive to accomplish the same task. This makes include files much more convenient for collecting all the common data and procedure declarations for your projects.

EXTERNDEF in Include Files The **EXTERNDEF** directive allows you to put global data declarations within an include file. The data is then visible to all source files that include the file. For more information, see Section 8.2.2.1, "Using EXTERNDEF."

Search Order for Include Files MASM 6.0 searches for include files in the directory of the main source file rather than in the current directory. Similarly, it searches for nested include files in the directory of the include file. You can specify additional paths to search with the /I command-line option. For more information, see Section 8.2.1, "Organizing Modules."

Enforcing Case Sensitivity In MASM 6.0, *langtype* takes precedence over the command-line options that specify case sensitivity. In MASM 5.1, only the command-line options influence case, not *langtype*.

Alternate Names for Externals The syntax for **EXTERN** allows you to specify an alternate symbol name, which the linker can use to resolve an external if the symbol is not otherwise referenced. See Section 8.4.2, "Using EXTERN with Library Routines."

A.1.6 Expanded State Control

Several new directives enable or disable various aspects of the assembler control, such as the new 80486 coprocessor instructions and use of compatibility options.

The OPTION Directive The new **OPTION** directive allows you to selectively define the assembler's behavior, including the enabling of compatibility with MASM 5.1. See Sections 1.3.2, "Using the OPTION Directive," and A.2, "Compatibility between MASM 5.1 and 6.0."

The .NO87 Directive The new **.NO87** directive disables all coprocessor instructions. See online help for more information.

The .486 and .486P Directives To enable the 80486 instructions, use the new **.486** directive. The **.486P** directive enables 80486 instructions at the highest privilege level (recommended for systems-level programs only). See online help for more information.

The PUSHCONTEXT and POPCONTEXT Directives The directive **PUSHCONTEXT** saves the assembly environment, and **POPCONTEXT** restores it. The environment includes the segment register assumes, the radix, the listing and CREF flags, and the current processor and coprocessor. Note that **.NOCREF** (the MASM equivalent to **.XCREF**) still determines whether information for a given symbol will be added to Browser information and to the symbol table in the listing file. See Appendix C or online help for more information on listing files.

A.1.7 New Processor Instructions

MASM 6.0 supports these new instructions for the 80486 processor:

80486 Instruction	Description
BSWAP	Byte swap
CMPXCHG	Compare and exchange
INVD	Invalidate data cache
INVLPG	Invalidate Translation Lookaside Buffer entry
WBINVD	Write back and invalidate data cache
XADD	Exchange and add

See the *Reference* or online help for full descriptions of these new instructions.

A.1.8 Renamed Directives

To make the language more consistent, the following directives have been renamed. The new, preferred, name is in the left column. MASM 6.0 still supports the old, obsolete names in the right column.

MASM 6.0	MASM 5.1
.DOSSEG	DOSSEG
.LISTIF	.LFCOND
.LISTMACRO	.XALL
.LISTMACROALL	.LALL
.NOCREF	.XCREF
.NOLIST	.XLIST
.NOLISTIF	.SFCOND
.NOLISTMACRO	.SALL
ECHO	%OUT
EXTERN	EXTRN
FOR	IRP
FORC	IRPC
REPEAT	REPT
STRUCT	STRUC
SUBTITLE	SUBTTL

Specifying 16-Bit and 32-Bit Instructions MASM 6.0 supports all instructions that work with the extended (32-bit) registers of the 80386/486. On certain instructions, you can override the default operand size with the **W** (word) and the **D** (doubleword) suffixes. See online help or the *Reference* for details.

A.1.9 Macro Enhancements

The changes to macro functionality in MASM 6.0 are also significant. New directives provide for a variable number of arguments, loop constructions, definitions of text equates, and macro functions.

Variable Arguments In MASM 5.1, extra arguments passed to macros are ignored. In MASM 6.0, you can pass a variable number of arguments to a macro by appending the **VARARG** keyword to the last macro parameter in the macro definition. Additional arguments passed to this macro can then be referenced

relative to the last declared parameter. Section 9.6, "Returning Values with Macro Functions," explains how to do this.

Required and Default Macro Arguments With MASM 6.0, you can use **REQ** or the := operator to specify required or default arguments. See Section 9.2.3.

New Directives for Macro Loops Within a macro definition, **WHILE** repeats assembly as long as a condition remains true. Other macro loop directives, **IRP**, **IRPC**, and **REPT**, have been renamed **FOR**, **FORC**, and **REPEAT**. For more information, see Section 9.4, "Defining Repeat Blocks with Loop Directives."

Text Macros You should use the **EQU** directive to define numeric constants, but MASM 6.0 also has a new **TEXTEQU** directive for defining text macros. **TEXTEQU** allows greater functionality than **EQU**. For example, it can assign the value calculated by a macro function to a label. For more information, see Section 9.1, "Text Macros."

The GOTO Directive for Macros Within a macro definition, **GOTO** transfers assembly to a labeled line. Lines in macros can be labeled using a leading colon(:). The **GOTO** directive can then be used to change the flow of control within that macro. See online help.

Macro Functions At assembly time, macro functions can determine and return a text value using **EXITM**. Predefined macro string functions concatenate strings, return the size of a string, find a substring in a string, and return the position of a substring within a string. For information on writing your own macro functions, see Section 9.6, "Returning Values with Macro Functions."

Predefined Macro Functions The following predefined text macro functions are new:

Symbol	Value Returned
@CatStr	A concatenated string
@InStr	The position of one string within another
@SizeStr	The size of a string
@SubStr	A substring

For more information, see Section 9.5, "String Directives and Predefined Functions."

A.1.10 MASM 6.0 Programming Practices

As you can see, MASM 6.0 provides many new features that can make MASM 6.0 code simpler to write. If you are familiar with MASM 5.1 programming, you may find it helpful to adopt this list of new programming practices for programming with the new assembler. This list summarizes many of the changes discussed in the next section, "Compatibility between MASM 5.1 and 6.0."

- Select identifier names that do not begin with the dot operator (.).
- Use the dot operator (.) only to reference structure fields, and the plus operator (+) when not referencing structures.
- Different structures can have the same field names if you like, but the names of structure fields must always be qualified with the structure's type.
- Separate macro arguments with commas, not spaces.
- Avoid adding extra ampersands in macros. (Section A.2.2.3, "OPTION OLDMACROS," and Section 9.3.3, "Substitution Operator," give the new rules for using ampersands in macros.)
- By default, code labels defined with a colon are local. Place two colons after code labels if you want to reference the label outside of the procedure.

A.2 Compatibility between MASM 5.1 and 6.0

This section discusses the differences between MASM 5.1 and MASM 6.0. Section A.2.1 provides information in addition to that found on the MASM 6.0 Quick Start card. The information in this section explains what changes you may need to make in order to get your MASM 5.1 code to run under MASM 6.0 in compatibility mode.

Note If you have not already done so, please read the *Quick Start for MASM 5.0 and 5.1 Users* card provided in your MASM 6.0 package.

Once your code runs in compatibility mode using **OPTION M510** or the /Zm command-line option, you may want to modify your code so it runs under MASM 6.0 without the compatibility options. To learn how to do this, see Section A.2.2, "Using the OPTION Directive."

You may notice that the .OBJ and .EXE files differ between MASM 5.1 and MASM 6.0. These differences do not necessarily indicate compatibility problems, since MASM 6.0 generates optimal encoding.

A.2.1 Rewriting Code for Compatibility

In some cases, MASM 6.0 with **OPTION M510** does not support MASM 5.1 behavior. Several of these changes result from correcting bugs reported against MASM 5.1. To update your code to MASM 6.0, use the instructions in this section. This usually requires only minor changes.

Many of the items listed in this section will not exist in your code. The items most likely to occur are listed first, followed by those that are less likely to occur.

In addition, you may have conflicts between identifier names and new reserved words. You can use **OPTION NOKEYWORD** to resolve errors generated due to use of reserved words as identifiers. See Section A.2.2.9 for more information.

A.2.1.1 Bug Fixes from MASM 5.1

This section lists the differences between MASM 5.1 and MASM 6.0 due to bug corrections from MASM 5.1.

Invalid Use of LOCK, REPNE, and REPNZ MASM 6.0 flags illegal uses of the instruction prefixes **LOCK**, **REPNE**, and **REPNZ**. The error generated for invalid uses of the **LOCK**, **REPNE**, and **REPNZ** prefixes is error A2068:

```
instruction prefix not allowed
```

Table A.1 summarizes the correct use of the instruction prefixes. It lists each string instruction with the type of repeat prefix it uses and indicates whether the instruction works on a source, a destination, or both.

Table A.1 Requirements for String Instructions

Instruction	Repeat Prefix	Source/Destination	Register Pair
MOVS	**REP**	Both	DS:SI, ES:DI
SCAS	**REPE/REPNE**	Destination	ES:DI
CMPS	**REPE/REPNE**	Both	DS:SI, ES:DI
LODS	None	Source	DS:SI
STOS	**REP**	Destination	ES:DI
INS	**REP**	Destination	ES:DI
OUTS	**REP**	Source	DS:SI

No Closing Quotation Marks in Macro Arguments In MASM 5.1, both single and double quotation marks (' and ") can be used to begin strings in macro arguments, and the assembler does not generate an error or warning if the

string does not end with quotation marks on a macro call. Instead, the assembler considers the remainder of the line to be part of the macro argument containing the opening quote (as if there were a closing quotation mark at the end of the line).

By default, MASM 6.0 now generates error A2046:

```
missing single or double quotation mark in string
```

so all single and double quotation marks in macro arguments must be matched. (Angle brackets not enclosed by brackets must also be matched.)

To correct errors the assembler finds, either end the string with a closing quotation mark as shown in this example, or use the macro escape character (!) to treat the quotation mark literally.

```
; MASM 5.1 code
MyMacro     "all this in one argument

; Default MASM 6.0 code
MyMacro     "all this in one argument"
```

Making a Scoped Label Public MASM 5.1 considers code labels defined with a single colon inside a procedure to be local to that procedure if the module contains a **.MODEL** directive with a language type. Although the label is local, MASM 5.1 does not generate an error if it is also declared **PUBLIC**. MASM 6.0 generates error A2203:

```
cannot declare scoped code label as PUBLIC."
```

If you want to make the label **PUBLIC**, it must not be local. You can use the double colon operator to define a non-scoped label, as shown in this example:

```
        PUBLIC  publicLabel
publicLabel::                   ; Non-scoped label MASM 6.0
```

Byte Form of BT, BTS, BTC, and BTR Instructions MASM 5.1 allows a byte argument for the 80386 bit-test instructions, but encodes it as a word argument. The byte form is not supported by the processor.

MASM 6.0 does not support this behavior and generates error A2024:

```
invalid operand size for instruction
```

Rewrite your code to use a word-sized argument.

Default Values for Record Fields In MASM 5.1, default values for record fields can range down to -2^n (where n is the number of bits in the field), resulting in the loss of the sign bit.

The allowed range for default values in MASM 6.0 is -2^{n-1} to 2^{n-1}. Illegal initializers generate error A2071:

```
initializer too large for specified size
```

A.2.1.2 Design Change Issues

MASM 6.0 makes some changes in MASM 5.1 behavior to make the language more consistent. These design changes are not affected by the **OPTION** directive. Therefore, they require revisions in your code. In most cases, the necessary revisions are minor and the circumstances requiring changes are rare.

Conflicting Structure Definitions MASM 5.1 allows two structures to be defined with the same name. The second definition replaces the first definition. However, the fields from the first are still defined. MASM 6.0 does not allow conflicting definitions of a structure. Errors A2160 through A2165 are generated when the assembler finds a conflicting definition. Each error notes a specific conflict, such as conflicting number of fields, conflicting names of fields, or conflicting initializers.

Forward References to Text Macros Outside of Expressions MASM 5.1 allows forward references to text macros in specialized cases. MASM 6.0 with **OPTION M510** also permits forward references, but only when the text macro is referenced in an expression. To revise your code, place all macro definitions at the beginning of the file.

HIGH and LOW Applied to Relocatable Operands In MASM 5.1, applying **HIGH** and **LOW** to relocatable memory expressions is acceptable in some cases. For example, MASM 5.1 allows this code sequence:

```
; MASM 5.1 code
EXTRN   var1:WORD
var2    DW      0
        mov     al, LOW var1  ; These two instructions yield the
        mov     ah, HIGH var1 ;  same as mov ax, OFFSET var1
```

However, `mov ax, LOW var2` is not legal. MASM 6.0 generates error A2105:

```
HIGH and LOW require immediate operands
```

The **OFFSET** operator is required on these operands in MASM 6.0, as shown below. Rewrite your code if necessary.

```
; MASM 6.0 code
        mov     al, LOW OFFSET var1
        mov     ah, HIGH OFFSET var2
```

OFFSET Applied to Group Names and Indirect Memory Operands

In MASM 6.0, you cannot apply **OFFSET** to a group name, indirect argument, or procedure argument. Doing so generates error A2098:

```
invalid operand for OFFSET
```

LENGTH Operator Applied to Record Types

In MASM 5.1, the **LENGTH** operator, when applied to a record type, returns the total number of bits in a record definition.

In MASM 6.0, the statement `LENGTH recordName` returns error A2143:

```
expected data label
```

Rewrite your code if necessary. The new **SIZEOF** operator returns information about records in MASM 6.0. See Section 5.3.2, "Defining Record Variables," for more information.

Signed Comparison of Hexadecimal Values Using GT, GE, LE, or LT

The rules for two's-complement comparisons have changed. In MASM 5.1, the statement

```
0FFFFh GT -1
```

is false because the two's-complement values are equal. However, because hexadecimal numbers are now treated as unsigned, the expression is true in MASM 6.0. To update, rewrite the affected code.

RET Used with a Constant in Procedures with Epilogues

By default in MASM 6.0, the **RET** instruction followed by a constant suppresses automatic generation of epilogue (stack cleanup) code. MASM 5.1 ignores the operand and generates the epilogue. Remove the argument if necessary. See Section 7.3.8, "Generating Prologue and Epilogue Code."

Code Labels at Top of Procedures with Prologues

By default in MASM 5.1, a code label defined on the same line as the first procedure instruction refers to the first byte of the prologue (the stack frame setup).

In MASM 6.0, a code label defined at the beginning of a procedure refers to the first byte of the procedure after the prologue. If a label is needed before the prologue, then the label must be placed before the **PROC** statement. See Section 7.3.8, "Generating Prologue and Epilogue Code," for more information.

Use of % as an Identifier Character

MASM 5.1 allows % as an identifier character. This undocumented behavior leads to ambiguities when % is used as the expansion operator in macros. Since % is not allowed as a character in MASM 6.0 identifiers, you must change the names of any identifiers containing the % character. See Section 1.2.2 for a list of legal identifier characters.

ASSUME CS Set to Wrong Value MASM 6.0 does not require the use of the **ASSUME** statement for the CS register. Instead, MASM 6.0 generates an automatic **ASSUME** statement for the code segment register to the current segment or group (see Section 2.3.3). Additionally, MASM 6.0 does not allow explicit **ASSUME** statements for CS that contradict the automatically set **ASSUME** statement.

MASM 5.1 allows CS to be assumed to the current segment, even if that segment is a member of a group. With MASM 6.0, this results in warning A4004:

```
cannot ASSUME CS
```

To avoid this warning with MASM 6.0, delete the **ASSUME** statement for CS.

A.2.1.3 Code Requiring Two-Pass Assembly

MASM 6.0 does not perform the standard two source passes that previous versions do. Therefore pass-dependent constructs are no longer meaningful.

Obsolete Two-Pass Directives Because MASM 6.0 assembles in one pass, the directives referring to two passes are no longer supported. These include **.ERR1**, **.ERR2**, **IF1**, **IF2**, **ELSEIF1**, and **ELSEIF2**. If you use **IF2** or **.ERR2**, the assembler generates error A2061:

```
[ELSE]IF2/.ERR2 not allowed : single-pass assembler
```

The **.ERR1** directive is treated as though it were **.ERR**, and the **IF1** directive is treated as though it were **IF**.

MASM 5.1 directives that refer to the first pass are always true. Directives that refer to the second pass are flagged as errors. This change requires you to rewrite the affected code, since **OPTION M510** does not enable this behavior.

You typically use pass-sensitive directive when doing the following: (Each example shows a MASM 6.0 rewrite.)

- Declaring `var` external only if it is not defined in this module:

```
; PREVIOUS VERSIONS OF MASM:
    IF2
        IFNDEF  var
            EXTRN var:far
        ENDIF
    ENDIF

; MASM 6.0:
    EXTERNDEF   var:far
```

- Including a file of definitions only once to speed assembly:

```
; PREVIOUS VERSIONS OF MASM:
    IF1
        INCLUDE file1.inc
    ENDIF

; MASM 6.0:
    INCLUDE FILE1.INC
```

- Generating a %OUT or .ERR message only once:

```
; PREVIOUS VERSIONS OF MASM:
    IF2
        %OUT This is my message
    ENDIF

    IF2
        .ERRNZ A NE B
    ENDIF

; MASM 6.0:
    ECHO This is my message

    .ERRNZ A NE B    <ASSERTION FAILURE: A NE B>
```

- Generating an error if a symbol is not defined but may be forward referenced:

```
; PREVIOUS VERSIONS OF MASM:
    IF2
        .ERRNDEF    var
    ENDIF

; MASM 6.0:
        .ERRNDEF    var
```

See Section 1.3.3 for information on conditional directives.

Note In the following three cases, MASM 6.0 generates warnings if **OPTION M510** is used.

IFDEF and IFNDEF with Forward-Referenced Identifiers If you use a symbol name that has not yet been defined in an **IFDEF** or **IFNDEF** expression, MASM 6.0 returns FALSE for the **IFDEF** expression and TRUE for the **IFNDEF** expression. The assembler generates warning A5005:

```
IF condition may be pass-dependent
```

when **OPTION M510** is enabled. To resolve the error, move the symbol definition to the beginning of the file.

Address Spans as Constants The value of offsets calculated on the first assembly pass may not be the same as those calculated on later passes. Therefore, comparisons with a constant, such as the following, should be avoided:

```
IF OFFSET var1 - OFFSET var2 EQ 10
```

Note that expressions containing span distances can be used with the **.ERR** directives, since these directives are evaluated after all offsets are determined:

```
.ERRE OFFSET var1 - OFFSET var2 - 10, <span incorrect>
```

.TYPE with Forward References In MASM 5.1, **.TYPE** is evaluated on both assembly passes. This means it yields zero on the first pass and non-zero on the second pass if applied to an expression that forward references a symbol.

In MASM 6.0, **.TYPE** is evaluated on the first assembly pass. As a result, if the operand references a symbol that has not yet been defined, **.TYPE** will yield 0. This means that **.TYPE**, if used in a conditional-assembly construction, may yield different results with MASM 6.0 than with MASM 5.1.

A.2.1.4 Obsolete Features No Longer Supported

This section lists features no longer supported by MASM 6.0. Because both of these items are obscure features provided by early versions of the assembler, they probably do not affect your MASM 5.1 code.

The ESC Instruction The **ESC** instruction, typically used to send hand-coded commands to the coprocessor, is no longer supported. Because MASM 6.0 recognizes and assembles the full set of coprocessor mnemonics, the **ESC** instruction is not necessary . Using the **ESC** instruction generates error A2205:

```
ESC instruction is obsolete: ignored
```

To update MASM 5.1 code, use the coprocessor instructions instead of **ESC**.

The MSFLOAT Binary Format MASM 6.0 does not support the **.MSFLOAT** directive, which provided the Microsoft Binary Format (MSB) for floating-point numbers in variable initializers. Using the **.MSFLOAT** directive generates error A2204:

```
.MSFLOAT directive is obsolete: ignored
```

Use IEEE format or, if MSB format is necessary, initialize variables with hexadecimal values. See Section 6.1.2, "Storing Numbers in Floating-Point Format."

A.2.2 Using the OPTION Directive

The **OPTION** directive can be used with various arguments to control compatibility with MASM 5.1 code. This section explains the differences in MASM 5.1 and MASM 6.0 behavior that can be influenced with the **OPTION** directive.

Section A.2.2.1 discusses the **M510** argument to the **OPTION** directive, which selects the MASM 5.1 compatibility mode. In this mode, MASM 6.0 implements MASM 5.1 behavior relating to macros, offsets, scope of code labels, structures, identifier names, identifier case, and other behaviors.

Note Wherever this appendix suggests using **OPTION M510** in your code, you can set the /Zm command-line option instead.

If you prefer to choose specific MASM 5.1 behaviors, rather than all those implemented by the **OPTION M510** directive, use the **OPTION** arguments discussed in Sections A.2.2.2 through A.2.2.9. Each section also explains how to revise your code if you want to remove **OPTION** directives from your MASM 5.1 code.

If you have used any processor or coprocessor instruction names as label names in your code, you can use the **OPTION NOKEYWORD** directive to remove them from the reserved word list. See Section A.2.2.9.

A.2.2.1 OPTION M510

Using **OPTION M510** is equivalent to adding /Zm to the command line. The **OPTION M510** directive automatically sets the following:

```
OPTION OLDSTRUCTS       ; MASM 5.1 structures
                        ;  See Section A.2.2.2
OPTION OLDMACROS        ; MASM 5.1 macros
                        ;  See Section A.2.2.3
OPTION DOTNAME          ; Identifiers may begin with a dot (.)
                        ;  See Section A.2.2.4
```

If you do not have a **.386**, **386P** **.486**, or **486P** directive in your module, then **OPTION M510** adds:

```
OPTION EXPR16           ; 16-bit expression precision
                        ;  See Section A.2.2.5
```

If you do not have a **.MODEL** directive in your module, **OPTION M510** adds:

```
OPTION OFFSET:SEGMENT   ; OFFSET operator defaults to
                        ;  segment-relative
                        ;  See Section A.2.2.6
```

If you do not have a **.MODEL** directive with a language specifier in your module, **OPTION M510** also adds:

```
OPTION NOSCOPED         ; Code labels are not local inside
                        ;  procedures
                        ;  See Section A.2.2.7
OPTION PROC:PRIVATE     ; Labels defined with PROC are not
                        ;  public by default
                        ;  See Section A.2.2.8
```

If you want to remove **OPTION M510** from your code (or /Zm from the command line), add the **OPTION** directive arguments to your module according to the conditions stated above.

There may be compatibility issues affecting your code that are supported under **OPTION M510**, but are not covered by the other **OPTION** directive arguments. Once your source code has been modified so it no longer requires behavior supported by **OPTION M510**, you can replace **OPTION M510** with other **OPTION** directive arguments. These compatibility issues are discussed in Sections A.2.2.2 through A.2.2.9.

Once you have replaced **OPTION M510** with other forms of the **OPTION** directive and your code works correctly, try removing the **OPTION** directives, one at a time. Make appropriate source modifications as necessary (see Sections A.2.2.2 through A.2.2.9), until your code uses only MASM 6.0 defaults.

Note **OPTION M510** enables the behaviors discussed below in addition to the behaviors corrected by the **OPTION** directive arguments described in Sections A.2.2.2 through A.2.2.9.

Reserved Keywords Dependent on CPU Mode with OPTION M510

With **OPTION M510**, keywords and instructions that are not available in the current CPU mode (such as **ENTER** under **.8086**) are not treated as keywords. This also means the **USE32**, **FLAT**, **FAR32**, and **NEAR32** segment types and the 80386/486 registers are not keywords with a processor selection less than **.386**.

If you remove **OPTION M510**, then any reserved word that you use as an identifier generates a syntax error. You can either rename the identifiers or use **OPTION NOKEYWORD**. See Section A.2.2.9 for more information on **OPTION NOKEYWORD**.

Invalid Use of Instruction Prefixes with OPTION M510 Code without **OPTION M510** generates errors for all invalid uses of the instruction prefixes. Using **OPTION M510** suppresses some of these errors in order to match MASM 5.1 behavior. MASM 5.1 does not check for illegal uses of the instruction prefixes **LOCK**, **REP**, **REPE**, **REPZ**, **REPNE**, and **REPNZ**.

Illegal uses of these prefixes result in error A2068:

```
instruction prefix not allowed
```

See Section 5.1.3.1, "Overview of String Operations", and Section A.2.1.1, "Bug Fixes from MASM 5.1" for more information on these instruction prefixes.

Sizes of Constant Operands with OPTION M510 In MASM 5.1, a constant whose value is so large it can fit only in the CPU's default word (four bytes for **.386** and **.486**, two bytes otherwise) is assigned a size attribute of the default word size. The value of the constant affects the number of bytes changed by the instruction. For example,

```
; Legal only with OPTION M510
        mov     [bx], 0100h
```

is legal in **OPTION M510** mode. Since `0100h` cannot fit in a byte, it is interpreted as a word.

Without **OPTION M510**, the assembler never assigns a size automatically. You must state it explicitly. Use **OPTION M510** to enable the MASM 5.1 behavior if you do not want to change your MASM 5.1 code.

For code without **OPTION M510**, the example above could be rewritten as:

```
; Without OPTION M510
        mov     ax, WORD PTR 0100h
```

Code Labels in Data Definition with OPTION M510 MASM 5.1 allows a code label definition in a data definition statement if that statement does not also define a data label. This is also allowed by MASM 6.0 if **OPTION M510** is enabled; otherwise it is illegal.

```
; Legal only with OPTION M510
MyCodeLabel:    DW      0
```

SEG Operator with OPTION M510 In MASM 5.1, the **SEG** operator returns a label's segment unless the frame is explicitly specified, in which case the frame is returned. A statement such as `SEG DGROUP:var` always returns `DGROUP`, whereas `SEG var` always returns the segment of `var`. **OPTION M510** provides this behavior.

If you do not use **OPTION M510**, the behavior of the **SEG** operator is determined by the **OPTION OFFSET** directive. See Section A.2.2.6.

When you use the **SEG** operator with a variable that is not external, code without **OPTION M510** returns the address of the frame (the segment, group, or the value assumed to the segment register) if one has been explicitly set. Otherwise, it returns the group if one has been specified. In the absence of a defined group, **SEG** returns the segment where the variable is defined.

Expression Evaluation with OPTION M510 By default, MASM 6.0 changes the way that expressions are evaluated. In MASM 5.1,

```
var-2[bx]
```

is parsed as

```
(var-2)[bx]
```

Without **OPTION M510**, you need to rewrite this statement, since it is parsed as

```
var-(2[bx])
```

which generates an error. **OPTION M510** provides the MASM 5.1 behavior.

Length and Size of Labels with OPTION M510 With **OPTION M510**, the **LENGTH** and **SIZE** operators can be applied to any label. For a code label, **SIZE** returns `0FFFFh` for **NEAR** and `0FFFEh` for **FAR**, and **LENGTH** always returns 1. For strings, **SIZE** and **LENGTH** return 1.

Without **OPTION M510**, **LENGTH** returns 1 except when used with **DUP**. In this case, the **LENGTH** operator returns the outermost **DUP** count. **SIZE** returns the length multiplied by the size of the type. However, the new **LENGTHOF** and **SIZEOF** operators return the number of data items and the number of bytes used by the initializer.

If you specify **OPTION M510** and the current word size is 2, **NEAR16** and **FAR16** correspond to the constants `0FFFFh` and `0FFFEh`, respectively. When the current word size is 4, **NEAR** and **FAR** (mapped to **NEAR32** and **FAR32**, respectively) correspond to `0FFFFh` and `0FFFEh`.

Without **OPTION M510**, the distance attributes **SHORT**, **NEAR16**, **NEAR32**, **FAR16**, and **FAR32** correspond to `0FF01h`, `0FF02h`, `0FF04h`, `0FF05h`, and `0FF06h`, respectively.

The behavior of the new **SIZEOF** and **LENGTHOF** operators for labels and strings is discussed in Section 5.1.1, "Declaring and Referencing Arrays"; Section 5.1.2, "Declaring and Initializing Strings"; Section 5.2.2, "Defining Structure and Union Variables"; and Section 5.3.2, "Defining Record Variables."

Comparing Types Using EQ and NE with OPTION M510 With **OPTION M510**, types are converted to a constant value equal to the size of the data type before comparisons with **EQ** and **NE**. Code types are converted to `0FFFFh` (near) and `0FFFEh` (far). If **OPTION M510** is not enabled, types are converted to constants only when comparing them with constants; two types are equal only if they are equivalent qualified types.

For existing MASM 5.1 code, these distinctions affect only the use of the **TYPE** operator in conjunction with **EQ** and **NE**. The following example illustrates this situation:

```
MYSTRUCT        STRUC
f1              DB              0
f2              DB              0
MYSTRUCT        ENDS

; With OPTION M510

val     =       (TYPE MYSTRUCT) EQ WORD     ; True: 2 EQ 2
val     =       2 EQ WORD                   ; True: 2 EQ 2
val     =       WORD EQ WORD                ; True: 2 EQ 2
val     =       SWORD EQ SWORD              ; True: 2 EQ 2

; Without OPTION M510

val     =       (TYPE MYSTRUCT) EQ WORD     ; False: MyStruct NE WORD
val     =       2 EQ WORD                   ; True:  2 EQ 2
val     =       WORD EQ WORD                ; True:  WORD EQ WORD
val     =       SWORD EQ SWORD              ; False: SWORD NE WORD
```

Use of Constant and PTR as a Type with OPTION M510 A constant can be used as the left operand to **PTR** when **OPTION M510** is enabled. Otherwise a type expression must be used. With **OPTION M510**, a constant must have a value of 1 (byte), 2 (word), 4 (dword), 6 (fword), 8 (qword) or 10 (tbyte), and it is treated as if the parenthesized type had been specified instead. Note that the **TYPE** operator yields a type expression, but the **SIZE** operator yields a constant.

```
; With OPTION M510

MyData  DW      0

        mov     WORD PTR [bx], 10           ; Legal
        mov     (TYPE MyData) PTR [bx], 10  ; Legal
        mov     (SIZE MyData) PTR [bx], 10  ; Legal
        mov     2 ptr [bx], 10              ; Legal

; Without OPTION M510

        mov     WORD PTR [bx], 10           ; Legal
        mov     (TYPE MyData) PTR [bx], 10  ; Legal
;       mov     (SIZE MyData) PTR [bx], 10  ; Illegal
;       mov     2 PTR [bx], 10              ; Illegal
```

Structure Type Cast on Expressions with OPTION M510 As with MASM 5.1, a constant can be type cast with the **PTR** operator to a structure type. This is most often used in data initializers to affect the CodeView information of the data label being defined. Without **OPTION M510**, the assembler generates an error.

```
MYSTRC  STRUC
f1      DB      0
MYSTRC  ENDS

MyPtr   DW      MYSTRC PTR 0    ; Illegal without OPTION M510
```

The type of initializers does not influence CodeView's type information with MASM 6.0.

Hidden Coercion of OFFSET Expression Size with OPTION M510

When programming for the 80386 or 80486, the size of an **OFFSET** expression may be two bytes (for a symbol in a **USE16** segment) or 4 bytes (for a symbol in a **USE32** or **FLAT** segment). However, with **OPTION M510**, a 32-bit **OFFSET** expression may be used in a 16-bit context. Without **OPTION M510**, the **LOWWORD** operator must be used to convert the offset size.

```
; With OPTION M510

        .386

seg32   SEGMENT USE32
MyLabel WORD    0
seg32   ENDS

seg16   SEGMENT USE16 'code'                ; With OPTIONS M510:
        mov     ax,  OFFSET MyLabel         ; Legal
        mov     ax,  LOWWORD OFFSET MyLabel ; Legal
        mov     eax, OFFSET MyLabel         ; Legal
seg16   ENDS

; Without OPTION M510

        .386

seg32   SEGMENT USE32
MyLabel WORD    0
seg32   ENDS

seg16   SEGMENT USE16 'code'                ; Without OPTION M510:
;       mov     ax,  OFFSET MyLabel         ; Illegal
        mov     ax,  LOWWORD offset MyLabel ; Legal
        mov     eax, OFFSET MyLabel         ; Legal
seg16   ENDS
```

Specifying Radixes with OPTION M510 If the current radix in your code (without **OPTION M510**) is greater than 10, then the radix specifiers **B** (binary) and **D** (decimal) are not supported. You will need to change **B** to **Y** for binary, and **D** to **T** for decimal, since both **B** and **D** are legitimate hexadecimal values, making numbers such as `12D` ambiguous. See Section 1.2.4, "Integer Constants and Constant Expressions," for more information.

If you don't want to change radix specifiers when the current radix is greater than 10, you need to specify **OPTION M510** in your code.

Naming Conventions with OPTION M510 By default in MASM 5.1, specifying a language type of **PASCAL**, **FORTRAN**, or **BASIC** does not cause names to be mapped to uppercase when publicly declared variables are written into the object file.

Unless you use **OPTION M510** in your code, these language types map identifier names to uppercase by default in MASM 6.0, even if you assemble with the /Cp or /Cx command-line options. See Section 20.1, "Naming and Calling Conventions."

When you link with /NOI[[GNORECASE]], case must be matched in the object files to resolve externals.

Length Significance of Symbol Names with OPTION M510 With MASM 5.1, only the first 31 characters of a symbol name are considered significant, and only the first 31 characters of a public or external symbol name are placed in the object file.

Without **OPTION M510**, the entire name is considered significant. The maximum number of characters placed in the object file is controlled with the /H*number* command-line option, with a default of 247 (the maximum length of an identifier in MASM 6.0).

String Defaults in Structure Variables with OPTION M510 With **OPTION M510**, a structure field initialized to a string value can be overridden with a constant. Without **OPTION M510**, a string can be overridden only with another string or with a list. To update your code, surround the constant override value with angle brackets or curly braces to indicate a list with one element.

```
MTSTRUCT        STRUCT
MyString        BTYE        "This is a string"
MTSTRUCT        ENDS

; With OPTION M510

MyInst          MTSTRUCT    <0>

; Without OPTION M510, either of these statement is correct

MyInst          MTSTRUCT    <<0>>

MyInst          MTSTRUCT    {<0>}
```

Effects of the ? Initializer in Data Definitions with OPTION M510 When **?** is used as a data initializer, it is sometimes treated as a zero and sometimes causes a byte to be left unspecified in the object file. The conditional behavior for MASM 6.0 without **OPTION M510** is explained in Section 5.1.2. With

OPTION M510, however, the **?** initializer is always treated as a zero unless it is used with the **DUP** operator. This rarely affects program execution.

Current Address Operator with OPTION M510 When **OPTION M510** is enabled, the value of the current address operator ($) for a structure instance is the offset of the first byte of the instance. When **OPTION M510** is not enabled, the value of **$** is the offset of the current field in the instance.

Segment Association for FAR Externals with OPTION M510 With MASM 5.1, a **FAR** external symbol defined inside a segment is considered to be inside that segment unless a **.MODEL** directive is used. With MASM 6.0, such a symbol is never considered to be inside that segment unless **OPTION M510** is used, in which case the MASM 5.1 behavior is emulated. Segment association for externals affects the frame of fixups generated on references to the symbols.

Defining Aliases Using EQU with OPTION M510 In MASM 5.1, a symbol can be equated to another symbol. These equates are called "aliases" in MASM 5.1. This behavior is simulated with **OPTION M510**.

If you don't use **OPTION M510**, aliases cannot be defined using **EQU**. The right operand of an **EQU** directive must be an immediate expression or text. Change aliases to use the **TEXTEQU** directive, which is described in Section 9.1. This change should have no effect on your code but may cause an expression to evaluate differently.

These examples illustrate MASM 5.1 code, MASM 6.0 code with **OPTION M510**, and MASM 6.0 code without **OPTION M510**:

```
; MASM 5.1 code
var1    EQU     3
var2    EQU     var1 ; var2 taken as an alias
                     ; var2 references var1 anywhere var2 is
                     ;  used as a symbol

; MASM 6.0 with OPTION M510
var1    EQU     3
var2    EQU     var1 ; var2 taken as a var2 EQU <var1>
                     ; var2 substituted for var1 whenever
                     ;  text macros substituted

; MASM 6.0 without OPTION M510
var1    EQU     3
var2    EQU     var1 ; Treated as var2 EQU 3
```

Difference in Text Macro Expansions with OPTION M510 When the name of a text macro is supplied as a text item, MASM 5.1 replaces the text macro name with its text value. However, if that text value contains other text macro names, no recursive expansion occurs. With MASM 6.0, recursive expansion occurs unless **OPTION M510** is enabled, as shown in the following example:

```
; With OPTION M510

tm1     EQU     <contains tm2>
tm2     EQU     <value>

tm3     CATSTR  tm1         ; == <contains tm2>

; Without OPTION M510

tm3     CATSTR  tm1         ; == <contains value>
```

Conditional Directives and Missing Operands with OPTION M510

MASM 5.1 considers a missing argument to be a zero. MASM 6.0 requires an argument unless **OPTION M510** is enabled.

A.2.2.2 OPTION OLDSTRUCTS

Changes made in MASM 6.0 that apply to structures are discussed in this section. With **OPTION OLDSTRUCTS** or **OPTION M510**:

- The plus operator can be used in structure field references in MASM 6.0. (The dot operator is required with **OPTION NOOLDSTRUCTS**, the default.)
- Labels and structure field names cannot have the same name with **OPTION OLDSTRUCTS** (but they can with **OPTION NOOLDSTRUCTS**).

Plus Operator Not Allowed with MASM 6.0 Structures

By default, each reference to structure member names must use the dot (.) operator to separate the structure variable name from the field name. Note that the dot (.) operator cannot be used as the plus (+) operator, nor can the plus operator be used as the dot operator.

To convert your code so that it does not need **OPTION OLDSTRUCTS**:

- Qualify all structure field references
- Change all uses of the dot operator (.) that occur outside of structure references to use the plus operator (+)

If you remove **OPTION OLDSTRUCTS** from your code, the assembler generates errors on all lines needing to be changed. Non-structure uses of the dot operator result in error A2166:

```
structure field expected
```

Unqualified structure references result in error A2006:

```
undefined symbol : identifier
```

This example shows code that doesn't work under the default, **OPTION NOOLDSTRUCTS**, and how to change it:

```
; OPTION OLDSTRUCTS (Does not work with OPTION NOOLDSTRUCTS)
structname      STRUC
a               BYTE  ?
b               WORD  ?
structname      ENDS

structinstance  structname <>

        mov     ax, [bx].b
        mov     al, structinstance.a
        mov     ax, [bx].4

; OPTION NOOLDSTRUCTS (the MASM 6.0 default)
structname      STRUCT
a               BYTE  ?
b               WORD  ?
structname      ENDS

structinstance  structname <>

        mov     ax, [bx].structname.b  ; Add qualifying type
        mov     al, structinstance.a   ; No change needed
        mov     ax, [bx]+4             ; Change dot to plus
; Alternative methods in MASM 6.0
        ASSUME  bx:PTR structname
        mov     ax, [bx] ; OR:

        mov     ax, (structname PTR[bx]).b
```

Non-Unique Structure Field Names Allowed in MASM 6.0 With the default, **OPTION NOOLDSTRUCTS**, label and structure field names may have the same name. With **OPTION OLDSTRUCTS** (the MASM 5.1 default), labels and structure fields cannot have the same name. For more information, see Section 5.2, "Structures and Unions."

A.2.2.3 OPTION OLDMACROS

If you use MASM 6.0 without **OPTION OLDMACROS** or **OPTION M510**, the behavior of macros is changed in several ways. If you want the MASM 5.1 macro behavior, add **OPTION OLDMACROS** or **OPTION M510** to your MASM 5.1 code.

Depending on the complexity of your MASM 5.1 macros and your programming style, it may be easy to make the necessary changes to remove **OPTION OLDMACROS**. This section describes the differences.

Commas Separating Macro Arguments MASM 5.1 allows white spaces or commas to separate arguments to macros. MASM 6.0 with **OPTION NOOLDMACROS** (the default), requires commas between arguments. For example, in the macro call

```
MyMacro  var1 var2 var3, var4
```

OPTION OLDMACROS passes four arguments (separated by spaces), but **OPTION NOOLDMACROS** passes only two arguments (separated by a comma). To convert your macro code, replace any space delimiters between macro arguments with commas.

New Behavior with Ampersands in Macros Using the MASM 6.0 assembler default, **OPTION NOOLDMACROS**, causes ampersands (**&**) to be interpreted within a macro differently than in MASM 5.1. The number of ampersands and their positions in a statement determine the result of the macro expansion in MASM 5.1. Parameters for use in nested MASM 5.1 macros must be prefixed with several ampersands, since the assembler removes one ampersand for each level of macro expansion. Using **OPTION OLDMACROS** enables this behavior.

Without **OPTION OLDMACROS**, ampersands are removed only once no matter how deeply nested the macro. To update your MASM 5.1 macros, a simple rule can be followed: Replace every sequence of ampersands with a single ampersand. The only exception to this is when macro parameters immediately precede and follow the ampersand, and both are to be substituted. In this case, two ampersands are needed. See Section 9.3.3, "Substitution Operator," for a description of the new rules.

This example shows how to update a MASM 5.1 macro:

```
; OPTION OLDMACROS (the MASM 5.1 behavior)

createNames     macro   arg
    irp         tail, <Next, Last>
        irp     num,  <1, 2>
        ; Define more names of the form: abcNext1?
arg&&tail&&&num&&&?   label   BYTE
        ENDM
    ENDM
ENDM

; OPTION NOOLDMACROS (the MASM 6.0 default)

createNames     macro   arg
    for         tail, <Next, Last>  ; FOR is the MASM 6.0
        for     num,  <1, 2>        ;  synonym for irp
        ; Define more names of the form: abcNext1?
arg&&tail&&num&?   label   BYTE
        ENDM
    ENDM
ENDM
```

A.2.2.4 OPTION DOTNAME

MASM 5.1 allows names of identifiers to begin with a period. The MASM 6.0 default is **OPTION NODOTNAME**. Adding **OPTION DOTNAME** to your code provides the MASM 5.1 behavior.

If you don't want to use this directive in your source code, rename the identifiers whose names begin with a period.

A.2.2.5 OPTION EXPR16

The **OPTION EXPR16** statement sets the expression word size to 16 bits. If you do not have **.386**, **.386P**, **.486**, or **.486P** directives in your MASM 5.1 code, **OPTION EXPR16** is the default. For MASM 6.0, **OPTION EXPR32** (an expression word size of 32 bits) is the default.

It may not be easy to determine the effect of changing from 16-bit internal expression size to 32-bit size. In many cases, the 32-bit word size results in no change to MASM 5.1. code. However, problems may arise due to differences in intermediate values during evaluation of expressions. If you generate a listing file with the /Fl and /Sa command-line options with and without **OPTION EXPR16**, you can compare the files for differences.

It is illegal to change the expression size once it has been set with the **OPTION** directive. Changing the CPU type to **.386** or **.486** also sets **OPTION EXPR32**.

A.2.2.6 OPTION OFFSET

In MASM 5.1 code, offsets are computed with respect to the segment when the **.MODEL** is not used. This is equivalent to **OPTION OFFSET:SEGMENT**. **OPTION M510** adds **OPTION OFFSET:SEGMENT** to your code if there is no **.MODEL** directive.

When the **.MODEL** directive is used, offsets are computed with respect to the group. This is equivalent to MASM 6.0's **OPTION OFFSET:GROUP** (the MASM 6.0 default).

Changing from **OPTION OFFSET:SEGMENT** to **OPTION OFFSET:GROUP** usually causes no problems. However, it is not easy to determine if changes are needed.

The behavior of the **OFFSET** operator depends on the arguments used with **OPTION OFFSET**. If no **GROUP** directives are used, no changes are needed. Otherwise, use of the **OFFSET** operator must be examined to see if the operand is in a grouped segment with no group override. If so, a segment name override must be used. The following example shows equivalent statements for **OPTION OFFSET:SEGMENT** and **OPTION OFFSET:GROUP**:

```
; OPTION OFFSET:SEGMENT
MyGroup GROUP   MySeg

MySeg   SEGMENT 'data'
MyLabel LABEL   BYTE
        DW      OFFSET MyLabel
        DW      OFFSET MyGroup:MyLabel
        DW      OFFSET MySeg:MyLabel
MySeg   ENDS
```

In this example, the first use of **OFFSET** must be changed to `OFFSET MySeg:MyLabel`. The second and third uses do not need to be changed:

```
; OPTION OFFSET:GROUP
MyGroup GROUP   MySeg

MySeg   SEGMENT 'data'
MyLabel LABEL   BYTE
        DW      OFFSET MySeg:MyLabel
        DW      OFFSET MyGroup:MyLabel
        DW      OFFSET MySeg:MyLabel
MySeg   ENDS
```

Without **OPTION M510**, the **OPTION OFFSET** directive determines whether **SEG** is group- or segment-relative. When you don't use **OPTION M510**, the **SEG** operator behaves the same as the **OFFSET** operator does relative to **OPTION OFFSET**. With **OPTION M510**, **SEG** is always segment-relative by default, regardless of the current value of **OPTION OFFSET** (including the effect on **OPTION OFFSET** of a **.MODEL** directive).

To remove **OPTION M510** from your code, add **OPTION OFFSET:SEGMENT** if there is no **.MODEL** directive in your code.

A.2.2.7 OPTION NOSCOPED

Under MASM 5.1, code labels are scoped (local to the current procedure) if the **.MODEL** directive specifies a language type.They are not scoped (not local to the current procedure) if a language is not specified. Without **OPTION M510** or **OPTION NOSCOPED**, code labels are always scoped.

If your MASM 5.1 code does not specify a language type and you want to assemble without **OPTION M510**, add **OPTION NOSCOPED** to your code.

To determine which labels need to be changed, remove the **OPTION NOSCOPED** directive and assemble the module. The assembler generates error A2006:

```
undefined symbol : identifier
```

for each reference to a non-local symbol.

A.2.2.8 OPTION PROC

By default, MASM 6.0 procedures are public (**OPTION PROC:PUBLIC**), but you can explicitly specify the default for procedure visibility with **OPTION PROC:PRIVATE** or **OPTION PROC:EXPORT**.

If your module does not have a language specifier with the **MODEL** directive, using **OPTION M510** adds **OPTION PROC:PRIVATE** to the module. If you do not want to use **OPTION PROC:PRIVATE**, you can add the **PRIVATE** keyword to each procedure you want to make private. The following example shows how to change MASM 5.1 code to make a procedure private:

```
; MASM 5.1 (OPTION PROC:PRIVATE)
MyProc PROC NEAR

; MASM 6.0 (OPTION PROC:PUBLIC)
MyProc PROC NEAR PRIVATE
```

This is necessary only to avoid naming conflicts between public names in multiple modules or libraries. The symbol table in a listing file shows the visibility (public, private, or export) of each procedure.

A.2.2.9 OPTION NOKEYWORD

MASM 5.1 allows you to use reserved words for names of identifiers, macro parameters, and text macros. Several new reserved words have been added to MASM 6.0. If your existing code uses a reserved word as a symbol name, your code generates a syntax error on assembly.

Identifiers and text macros can be keywords if you disable individual keywords with the **OPTION NOKEYWORD** directive. For example,

```
OPTION NOKEYWORD:<INVOKE STRUCT>
```

removes two keywords, **INVOKE** and **STRUCT** from the reserved word list.

As an alternative to using **OPTION NOKEYWORD**, you can rename the offending label. For example, a label named `Str` could be renamed `Str1`.

The following list names all the new reserved words in MASM 6.0:

.BREAK
.CONTINUE
.DOSSEG
.ELSE
.ELSEIF
.ENDIF
.ENDW
.EXIT
.IF
.LISTALL
.LISTIF
.LISTMACRO
.LISTMACROALL
.NO87
.NOCREF
.NOLIST
.NOLISTIF
.NOLISTMACRO
.REPEAT
.STARTUP
.UNTIL
.UNTILCXZ
.WHILE
ADDR
ALIAS
BSWAP
CARRY?
CMPXCHG
ECHO
EXTERN
EXTERNDEF
FAR16
FAR32
FLAT
FLDENVD
FLDENVW
FNSAVED
FNSAVEW
FNSTENVD
FNSTENVW
FOR
FORC
FRSTORD
FRSTORW
FSAVED
FSAVEW
FSTENVD
FSTENVW
GOTO
HIGHWORD
INVD
INVLPG
INVOKE
IRETDF
IRETF
LENGTHOF
LOOPD
LOOPED
LOOPEW
LOOPNED
LOOPNEW
LOOPNZD
LOOPNZW
LOOPW
LOOPZW
LOWWORD
LROFFSET
NEAR16
NEAR32
OPATTR
OPTION
OVERFLOW?
PARITY?
POPAW
POPCONTEXT
PROTO
PUSHAW
PUSHCONTEXT
PUSHD
PUSHW
REAL10
REAL4
REAL8
REPEAT
SBYTE
SDWORD
SIGN?
SIZEOF
STDCALL
STRUCT
SUBTITLE
SWORD
SYSCALL
TEXTEQU
TR3
TR4
TR5
TYPEDEF
UNION
VARARG
WBINVD
WHILE
XADD
ZERO?

A.2.3 Changes to Instruction Encodings

MASM 6.0 contains changes to the encodings for several instructions. In some cases, the changes help optimize code size.

Coprocessor Instructions MASM 5.1 adds an extra **NOP** instruction before the no-wait versions of coprocessor instructions. MASM 6.0 does not. In the rare case that the missing **NOP** affects the timing, insert **NOP**.

Also, in **.286** mode, MASM 6.0 does not prefix any 8087, 80287, 80387, or 80486 coprocessor instruction with **FWAIT** (unless the instruction is the **WAIT** form of an instruction that has a **NOWAIT** form). MASM 5.1 prefixes some of these instructions with **FWAIT**.

RET Instruction If the operand to **RET**, **RETN**, or **RETF** is 0, MASM 6.0 uses the one-byte encoding. MASM 5.1 generates the three-byte encoding in this case. Thus, it is possible to suppress the epilogue generation but still specify the default size for the **RET** (**NEAR** or **FAR**), by coding the return as

```
RET 0
```

If the operand for **RET**, **RETN**, or **RETF** is an external absolute, MASM 6.0 generates the three-byte encoding. In this case, MASM 5.1 ignores the parameter and generates the one-byte encoding.

LEA Instruction with Direct Memory Operands When the second operand to the **LEA** instruction is a direct memory operand (that is, the second operand does not contain registers), MASM 6.0 encodes the instruction as

```
mov     reg, OFFSET directmem
```

This is smaller and faster than the equivalent **LEA** encoding that MASM 5.1 generates. This should not affect your MASM 5.1 code.

Arithmetic Instructions If your program uses the arithmetic instructions **ADC**, **ADD**, **AND**, **CMP**, **OR**, **SUB**, **SBB**, and **XOR**, and the following conditions are also true:

- Either AX or EAX is the first operand
- A sign-extendable byte constant is the second operand

then the instructions are encoded in MASM 5.1 as *ax/eax, imm16/32*.

MASM 6.0 uses this encoding instead: *rm16/32. imm8*.

With the AX register, there is no size or speed difference between the two encodings. In the EAX case, MASM 6.0's encoding is two bytes smaller. The **OPTION NOSIGNEXTEND** directive provides the MASM 5.1 behavior.

Appendix B

BNF Grammar

The BNF grammar gives the full description of the MASM language. The MASM BNF follows the Backus-Naur Form (BNF) for grammar notation.

You can use the BNF to determine the exact syntax for any language component. The BNF format clearly defines recursive definitions and shows all the available options for any placeholder.

Definitions

Terminals are endpoints in a BNF definition. No other resolution of their definition is possible. Terminals include the set of reserved words and user-defined objects.

Nonterminals are placeholders in the BNF definition. All nonterminals are defined elsewhere in the BNF.

The BNF references two types of expressions before they are formally defined: *constExpr* and *immExpr*. A *constExpr* is an expression whose value is not relocatable and not completely known at assembly time. An *immExpr* is similar to a *constExpr*, except that it may also be relocatable.

Conventions

The conventions use different font attributes for different items in the BNF. The symbols and formats are as follows:

Attribute	Description
nonterminal	Italic type indicates nonterminals.
RESERVED	Terminals in boldface type are literal reserved words and symbols that must be entered as shown. Characters in this context are always case insensitive.
[[ ]]	Objects enclosed in double brackets ([[ ]]) are optional. The brackets do not actually appear in the source code.
|	A vertical bar indicates a choice between the items on each side of the bar.

Attribute	Description
.8086	Underlined items indicate the default option if one is given.
default typeface	Characters in the set described or listed can be used as terminals in MASM statements.

How to Use

To illustrate the use of the BNF, Figure B.1 explores the definition of the **TYPEDEF** directive by starting with the nonterminal *typedefDir*.

Of course *typedefDir* is also an option given in the definition for a nonterminal "higher" than *typedefDir*. Look at the BNF definition for *generalDir* as an example.

The entries under each horizontal brace in Figure B.1 are terminals (such as **NEAR16**, **NEAR32**, **FAR16**, and **FAR32**) or nonterminals (such as *qualifier*, *qualifiedType*, *distance*, *and protoSpec*) that can be further defined. Each nonterminal (italicized word) in the *typedefDir* definition is also an entry in the BNF. Three vertical dots mean that the BNF description for that nonternminal is not illustrated in this figure (but is in the BNF).

Definitions can be recursive. As an example, note that *qualifiedType* is used in one of the two possible definitions for *qualifiedType* and is also a component of the definition for *qualifier*.

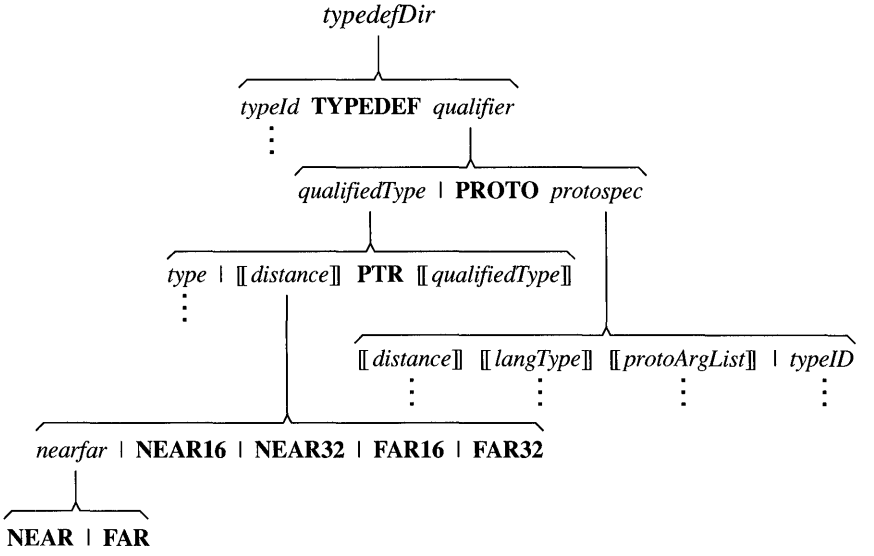


Figure B.1 BNF Definition of the TYPEDEF Directive

Nonterminal	Definition
;;	endOfLine | *comment*
=Dir	*id* = *immExpr ;;*
addOp	+ | -
aExpr	*term* | *aExpr* **&&** *term*
alpha	a thru z | A thru Z | **?** | **@** | **_** | **$**
altId	*id*
arbitraryText	*charList*
asmInstruction	*mnemonic* [[*exprList*]]
assumeDir	**ASSUME** *assumeList ;;* | **ASSUME NOTHING** *;;*
assumeList	*assumeRegister* | *assumeList* , *assumeRegister*
assumeReg	*register* : *assumeVal*
assumeRegister	*assumeSegReg* | *assumeReg*
assumeSegReg	*segmentRegister* : *assumeSegVal*
assumeSegVal	*frameExpr* | **NOTHING** | **ERROR**
assumeVal	*qualifiedType* | **NOTHING** | **ERROR**
bcdConst	[[*sign*]] *decNumber*
binaryOp	== | **!=** | >= | <= | > | < | **&**
bitDef	*bitFieldId* : *bitFieldSize* [[= *constExpr*]]
bitDefList	*bitDef* | *bitDefList* , [[*;;*]] *bitDef*
bitFieldId	*id*
bitFieldSize	*constExpr*
blockStatements	*directiveList* | **.CONTINUE** [[**.IF** *cExpr*]] | **.BREAK** [[**.IF** *cExpr*]]
byteRegister	AL | AH | BL | BH | CL | CH | DL | DH

Nonterminal	Definition
cExpr	*aExpr* | *cExpr* || *aExpr*
character	Any character value (ordinal in the range 0–255) except linefeed (10)
charList	*character* | *charList character*
className	*string*
commDecl	[[*nearfar*]] [[*langType*]] *id* **:** *commType* [[**:** *constExpr*]]
commDir	**COMM** *commList ;;*
comment	**;** *text ;;*
commentDir	**COMMENT** *delimiter* *text* *text delimiter text ;;*
commList	*commDecl* | *commList* **,** *commDecl*
commType	*type* | *constExpr*
constant	*digits* [[*radixOverride*]]
constExpr	*expr*
contextDir	**PUSHCONTEXT** *contextItemList ;;* | **POPCONTEXT** *contextItemList ;;*
contextItem	**ASSUMES** | **RADIX** | **LISTING** | **CPU** | **ALL**
contextItemList	*contextItem* | *contextItemList* **,** *contextItem*
controlBlock	*whileBlock* | *repeatBlock*
controlDir	*controlIf* | *controlBlock*
controlElseif	**.ELSEIF** *cExpr ;;* *directiveList* [[*controlElseif*]]

Nonterminal	Definition
controlIf	**.IF** *cExpr ;;* *directiveList* [[*controlElseif*]] [[**.ELSE** *;;* *directiveList*]] **.ENDIF** *;;*
coprocessor	**.8087** | **.287** | **.387** | **.NO87**
crefDir	*crefOption ;;*
crefOption	**.CREF** | **.XCREF** [[*idList*]] | **.NOCREF** [[*idList*]]
cxzExpr	*expr* | **!** *expr* | *expr* == *expr* | *expr* **!=** *expr*
dataDecl	DB | DW | DD | DF | DQ | DT | *dataType* | *typeId*
dataDir	[[*id*]] *dataItem ;;*
dataItem	*dataDecl scalarInstList* | *structTag structInstList* | *typeId structInstList* | *unionTag structInstList* | *recordTag recordInstList*
dataType	**BYTE** | **SBYTE** | **WORD** | **SWORD** | **DWORD** | **SDWORD** | **FWORD** | **QWORD** | **TBYTE** | **REAL4** | **REAL8** | **REAL10**
decdigit	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
decNumber	*decdigit* | *decNumber decdigit*
delimiter	Any character other than *whiteSpaceCharacter*
digits	*decdigit* | *digits decdigit* | *digits hexdigit*
directive	*generalDir* | *segmentDef*
directiveList	*directive* | *directiveList directive*
distance	*nearfar* | **NEAR16** | **NEAR32** | **FAR16** | **FAR32**

Nonterminal	Definition
e01	*e01 orOp e02* | *e02*
e02	*e02* **AND** *e03* | *e03*
e03	**NOT** *e04* | *e04*
e04	*e04 relOp e05* | *e05*
e05	*e05 addOp e06* | *e06*
e06	*e06 mulOp e07* | *e06 shiftOp e07* | *e07*
e07	*e07 addOp e08* | *e08*
e08	**HIGH** *e09* | **LOW** *e09* | **HIGHWORD** *e09* | **LOWWORD** *e09* | *e09*
e09	**OFFSET** *e10* | **LROFFSET** *e10* | **TYPE** *e10* | **THIS** *e10* | *e09* **PTR** *e10* | *e09* **:** *e10* | *e10*
e10	*e10* **.** *e11* | *e10* [[*expr*]] | *e11*

Nonterminal	Definition
e11	(*expr*) | [[*expr*]] | **WIDTH** *id* | **MASK** *id* | **SIZE** *sizeArg* | **SIZEOF** *sizeArg* | **LENGTH** *id* | **LENGTHOF** *id* | *recordConst* | *string* | *constant* | *type* | *id* | **$** | *segmentRegister* | *register* | **ST** | **ST** (*expr*)
echoDir	**ECHO** *arbitraryText ;;*
elseifBlock	*elseifStatement ;;* *directiveList* [[*elseifBlock*]]
elseifStatement	**ELSEIF** *constExpr* | **ELSEIFE** *constExpr* | **ELSEIFB** *textItem* | **ELSEIFNB** *textItem* | **ELSEIFDEF** *id* | **ELSEIFNDEF** *id* | **ELSEIFDIF** *textItem* , *textItem* | **ELSEIFDIFI** *textItem* , *textItem* | **ELSEIFIDN** *textItem* , *textItem* | **ELSEIFIDNI** *textItem* , *textItem* | **ELSEIF1** | **ELSEIF2**
endDir	**END** [[*immExpr*]] *;;*
endpDir	*procId* **ENDP** *;;*
endsDir	*id* **ENDS** *;;*
equDir	*textMacroId* **EQU** *equType ;;*
equType	*immExpr* | *textLiteral*
errorDir	*errorOpt ;;*

Nonterminal	Definition
errorOpt	**.ERR** [[*textItem*]] | **.ERRE** *constExpr* [[*optText*]] | **.ERRNZ** *constExpr* [[*optText*]] | **.ERRB** *textItem* [[*optText*]] | **.ERRNB** *textItem* [[*optText*]] | **.ERRDEF** *id* [[*optText*]] | **.ERRNDEF** *id* [[*optText*]] | **.ERRDIF** *textItem* , *textItem* [[*optText*]] | **.ERRDIFI** *textItem* , *textItem* [[*optText*]] | **.ERRIDN** *textItem* , *textItem* [[*optText*]] | **.ERRIDNI** *textItem* , *textItem* [[*optText*]] | **.ERR1** [[*textItem*]] | **.ERR2** [[*textItem*]]
exitDir	**.EXIT** [[*expr*]] *;;*
exitmDir:	**EXITM** | **EXITM** *textItem*
exponent	E [[*sign*]] *decNumber*
expr	**SHORT** *e05* | **.TYPE** *e01* | **OPATTR** *e01* | *e01*
exprList	*expr* | *exprList* , *expr*
externDef	[[*langType*]] *id* [[(*altId*)]] **:** *externType*
externDir	*externKey externList ;;*
externKey	**EXTRN** | **EXTERN** | **EXTERNDEF**
externList	*externDef* | *externList* , [[*;;*]] *externDef*
externType	**ABS** | *qualifiedType*
fieldAlign	*constExpr*
fieldInit	[[*initValue*]] | *structInstance*
fieldInitList	*fieldInit* | *fieldInitList* , [[*;;*]] *fieldInit*

Nonterminal	Definition
fileChar	Any character value (ordinal in the range 0–255) except backspace (8), tab (9), linefeed (10), vertical tab (11), form feed (12), carriage return (13), ^Z (26), or space (32)
fileCharList	*fileChar* | *fileCharList fileChar*
fileSpec	*fileCharList* | *textLiteral*
flagName	**ZERO?** | **CARRY?** | **OVERFLOW?** | **SIGN?** | **PARITY?**
floatNumber	[[*sign*]] *decNumber* . [[*decNumber*]] [[*exponent*]] | *digits* R | *digits* r
forcDir	**FORC** | **IRPC**
forDir	**FOR** | **IRP**
forParm	*id* [[**:** *forParmType*]]
forParmType	**REQ** | = *textLiteral*
frameExpr	*expr*
generalDir	*modelDir* | *segOrderDir* | *nameDir* | *includeLibDir* | *commentDir* | *groupDir* | *assumeDir* | *structDir* | *recordDir* | *typedefDir* | *externDir* | *publicDir* | *commDir* | *protoTypeDir* | *equDir* | =*Dir* | *textDir* | *contextDir* | *optionDir* | *processorDir* | *radixDir* | *titleDir* | *pageDir* | *listDir* | *crefDir* | *echoDir* | *ifDir* | *errorDir* | *includeDir* | *macroDir* | *macroCall* | *macroRepeat* | *purgeDir* | *macroWhile* | *macroFor* | *macroForc* | *aliasDir*
gpRegister	AX | EAX | BX | EBX | CX | ECX | DX | EDX | BP | EBP | SP | ESP | DI | EDI | SI | ESI
groupDir	*groupId* **GROUP** *segIdList*
groupId	*id*

Nonterminal	Definition
hexdigit	a | b | c | d | e | f | A | B | C | D | E | F
id	*alpha* | *id alpha* | *id decdigit*
idList	*id* | *idList* , *id*
ifDir	*ifStatement ;;* *directiveList* [[*elseifBlock*]] [[**ELSE** *;;* *directiveList*]] **ENDIF** *;;*
ifStatement	**IF** *constExpr* | **IFE** *constExpr* | **IFB** *textItem* | **IFNB** *textItem* | **IFDEF** *id* | **IFNDEF** *id* | **IFDIF** *textItem* , *textItem* | **IFDIFI** *textItem* , *textItem* | **IFIDN** *textItem* , *textItem* | **IFIDNI** *textItem* , *textItem* | **IF1** | **IF2**
immExpr	*expr*
includeDir	**INCLUDE** *fileSpec ;;*
includeLibDir	**INCLUDELIB** *fileSpec ;;*
initValue	*immExpr* | *string* | **?** | *constExpr* **DUP** (*scalarInstList*) | *floatNumber* | *bcdConst*
inSegDir	[[*labelDef*]] *inSegmentDir*
inSegDirList	*inSegDir* | *inSegDirList inSegDir*

Nonterminal	Definition
inSegmentDir	*instruction* | *dataDir* | *controlDir* | *startupDir* | *exitDir* | *offsetDir* | *labelDir* | *procDir* [[*localDirList*]] [[*inSegDirList*]] *endpDir* | *invokeDir* | *generalDir*
instrPrefix	**REP** | **REPE** | **REPZ** | **REPNE** | **REPNZ** | **LOCK**
instruction	[[*instrPrefix*]] *asmInstruction*
invokeArg	*register* **::** *register* | *expr* | **ADDR** *expr*
invokeDir	**INVOKE** *expr* [[**,** [[*;;*]] *invokeList*]] *;;*
invokeList	*invokeArg* | *invokeList* **,** [[*;;*]] *invokeArg*
keyword	Any reserved word
keywordList	*keyword* | *keyword keywordList*
labelDef	*id* **:** | *id* **::** | **@@:**
labelDir	*id* **LABEL** *qualifiedType ;;*
langType	**C** | **PASCAL** | **FORTRAN** | **BASIC** | **SYSCALL** | **STDCALL**
listDir	*listOption ;;*
listOption	**.LIST** | **.NOLIST** | **.XLIST** | **.LISTALL** | **.LISTIF** | **.LFCOND** | **.NOLISTIF** | **.SFCOND** | **.TFCOND** | **.LISTMACROALL** | **.LALL** | **.NOLISTMACRO** | **.SALL** | **.LISTMACRO** | **.XALL**
localDef	**LOCAL** *idList ;;*

Nonterminal	Definition
localDir	**LOCAL** *parmList ;;*
localDirList	*localDir* | *localDirList localDir*
localList	*localDef* | *localList localDef*
macroArg	**%** *constExpr* | **%** *textMacroId* | **%** *macroFuncId* (*macroArgList*) | *string* | *arbitraryText* | < *arbitraryText* >
macroArgList	*macroArg* | *macroArgList* , *macroArg*
macroBody	[[*localList*]] *macroStmtList*
macroCall	*id macroArgList ;;* | *id* (*macroArgList*)
macroDir	*id* **MACRO** [[*macroParmList*]] *;;* *macroBody* **ENDM** *;;*
macroFor	*forDir forParm* , < *macroArgList* > *;;* *macroBody* **ENDM** *;;*
macroForc	*forcDir id* , *textLiteral ;;* *macroBody* **ENDM** *;;*
macroFuncId	*id*
macroId	*macroProcId* | *macroFuncId*
macroIdList	*macroId* | *macroIdList* , *macroId*
macroLabel	*id*
macroParm	*id* [[: *parmType*]]
macroParmList	*macroParm* | *macroParmList* , [[*;;*]] *macroParm*
macroProcId	*id*

Nonterminal	Definition
macroRepeat	*repeatDir constExpr ;;* *macroBody* **ENDM** *;;*
macroStmt	*directive* | *exitmDir* | **:** *macroLabel* | **GOTO** *macroLabel*
macroStmtList	*macroStmt ;;* | *macroStmtList macroStmt ;;*
macroWhile	**WHILE** *constExpr ;;* *macroBody* **ENDM** *;;*
mapType	**ALL** | **NONE** | **NOTPUBLIC**
memOption	**TINY** | **SMALL** | **MEDIUM** | **COMPACT** | **LARGE** | **HUGE** | **FLAT**
mnemonic	Instruction name
modelDir	**.MODEL** *memOption* [[**,** *modelOptlist*]] *;;*
modelOpt	*langType* | *osType* | *stackOption*
modelOptlist	*modelOpt* | *modelOptlist* **,** *modelOpt*
module	[[*directiveList*]] *endDir*
mulOp	***** | **/** | **MOD**
nameDir	**NAME** *id ;;*
nearfar	**NEAR** | **FAR**
nestedStruct	*structHdr* [[*id*]] *;;* *structBody* **ENDS** *;;*
offsetDir	*offsetDirType ;;*
offsetDirType	**EVEN** | **ORG** *immExpr* | **ALIGN** [[*constExpr*]]
offsetType	**GROUP** | **SEGMENT** | **FLAT**
oldRecordFieldList	[[*constExpr*]] | *oldRecordFieldList* **,** [[*constExpr*]]

Nonterminal	Definition
optionDir	**OPTION** *optionList ;;*
optionItem	**CASEMAP :** *mapType* | **DOTNAME** | **NODOTNAME** | **EMULATOR** | **NOEMULATOR** | **EPILOGUE :** *macroId* | **LANGUAGE :** *langType* | **LJMP** | **NOLJMP** | **M510** | **NOM510** | **NOKEYWORD :** < *keywordList* > | **NOSIGNEXTEND** | **OFFSET :** *offsetType* | **OLDMACROS** | **NOOLDMACROS** | **OLDSTRUCTS** | **NOOLDSTRUCTS** | **PROC :** *oVisibility* | **PROLOGUE :** *macroId* | **READONLY** | **NOREADONLY** | **SCOPED** | **NOSCOPED** | **SEGMENT :** *segSize*
optionList	*optionItem* | *optionList* **,** [[*;;*]] *optionItem*
optText	**,** *textItem*
orOp	**OR** | **XOR**
osType	**OS_DOS** | **OS_OS2**
oVisibility	**PUBLIC** | **PRIVATE** | **EXPORT**
pageDir	**PAGE** [[*pageExpr*]] *;;*
pageExpr	**+** | [[*pageLength*]] [[**,** *pageWidth*]]
pageLength	*constExpr*
pageWidth	*constExpr*
parm	*parmId* [[**:** *qualifiedType*]] | *parmId* **[** *constExpr* **]** [[**:** *qualifiedType*]]
parmId	*id*
parmList	*parm* | *parmList* **,** [[*;;*]] *parm*
parmType	**REQ** | **=** *textLiteral* | **VARARG**
pOptions	[[*distance*]] [[*langType*]] [[*oVisibility*]]

Nonterminal	Definition
primary	*expr binaryOp expr* | *flagName* | *expr*
procDir	*procId* **PROC** [[*pOptions*]] [[< *macroArgList* >]] [[*usesRegs*]] [[*procParmList*]]
processor	**.8086** | **.186** | **.286** | **.286C** | **.286P** | **.386** | **.386C** | **.386P** | **.486** | **.486P**
processorDir	*processor ;;* | *coprocessor ;;*
procId	*id*
procParmList	[[[[, [[*;;*]]]] *parmList*]] [[[[, [[*;;*]]]] *parmId* **:VARARG**]]
protoArg	[[*id*]] **:** *qualifiedType*
protoArgList	[[[[, [[*;;*]]]] *protoList*]] [[[[, [[*;;*]]]] [[*id*]] **:VARARG**]]
protoList	*protoArg* | *protoList* , [[*;;*]] *protoArg*
protoSpec	[[*distance*]] [[*langType*]] [[*protoArgList*]] | *typeId*
protoTypeDir	*id* **PROTO** *protoSpec*
pubDef	[[*langType*]] *id*
publicDir	**PUBLIC** *pubList ;;*
pubList	*pubDef* | *pubList* , [[*;;*]] *pubDef*
purgeDir	**PURGE** *macroIdList*
qualifiedType	*type* | [[*distance*]] **PTR** [[*qualifiedType*]]
qualifier	*qualifiedType* | **PROTO** *protoSpec*
quote	" | '
radixDir	**.RADIX** *constExpr ;;*

Nonterminal	Definition
radixOverride	h | o | q | t | y | H | O | Q | T | Y
recordConst	*recordTag* { *oldRecordFieldList* } | *recordTag* < *oldRecordFieldList* >
recordDir	*recordTag* **RECORD** *bitDefList ;;*
recordFieldList	[[*constExpr*]] | *recordFieldList* , [[*;;*]] [[*constExpr*]]
recordInstance	{ [[*;;*]] *recordFieldList* [[*;;*]] } | < *oldRecordFieldList* > | *constExpr* **DUP** (*recordInstance*)
recordInstList	*recordInstance* | *recordInstList* , [[*;;*]] *recordInstance*
recordTag	*id*
register	*specialRegister* | *gpRegister* | *byteRegister*
regList	*register* | *regList register*
relOp	**EQ** | **NE** | **LT** | **LE** | **GT** | **GE**
repeatBlock	**.REPEAT** *;;* *blockStatements ;;* *untilDir ;;*
repeatDir	**REPEAT** | **REPT**
scalarInstList	*initValue* | *scalarInstList* , [[*;;*]] *initValue*
segAlign	**BYTE** | **WORD** | **DWORD** | **PARA** | **PAGE**
segAttrib	**PUBLIC** | **STACK** | **COMMON** | **MEMORY** | **AT** *constExpr* | **PRIVATE**

Nonterminal	Definition
segDir	**.CODE** [[*segId*]] | **.DATA** | **.DATA?** | **.CONST** | **.FARDATA** [[*segId*]] | **.FARDATA?** [[*segId*]] | **.STACK** [[*constExpr*]]
segId	*id*
segIdList	*segId* | *segIdList* , *segId*
segmentDef	*segmentDir* [[*inSegDirList*]] *endsDir* | *simpleSegDir* [[*inSegDirList*]] [[*endsDir*]]
segmentDir	*segId* **SEGMENT** [[*segOptionList*]] *;;*
segmentRegister	CS | DS | ES | FS | GS | SS
segOption	*segAlign* | *segRO* | *segAttrib* | *segSize* | *className*
segOptionList	*segOption* | *segOptionList segOption*
segOrderDir	**.ALPHA** | **.SEQ** | **.DOSSEG** | **DOSSEG**
segRO	**READONLY**
segSize	**USE16** | **USE32** | **FLAT**
shiftOp	**SHR** | **SHL**
sign	- | +
simpleExpr	(*cExpr*) | *primary*
simpleSegDir	*segDir ;;*
sizeArg	*id* | *type* | *e10*
specialChars	: | . | [[|]] | (|) | < | > | { | } | + | - | / | * | & | % | ! | ' | \ | = | ; | , | " | white space (8, 9, 11–13, 26, 32) | endOfLine

Nonterminal	Definition
specialRegister	CR0 | CR2 | CR3 | DR0 | DR1 | DR2 | DR3 | DR6 | DR7 | TR3 | TR4 | TR5 | TR6 | TR7
stackOption	**NEARSTACK** | **FARSTACK**
startupDir	**.STARTUP** *;;*
stext	*stringChar* | *stext stringChar*
string	*quote* [[*stext*]] *quote*
stringChar	*quote quote* | Any character value (ordinal in the range 0–255) except linefeed (10) and elements of *quote*
structBody	*structItem ;;* | *structBody structItem ;;*
structDir	*structTag structHdr* [[*fieldAlign*]] [[, **NONUNIQUE**]] *;;* *structBody* *structTag* **ENDS** *;;*
structHdr	**STRUC** | **STRUCT** | **UNION**
structInstance	< [[*fieldInitList*]] > | { [[*;;*]] [[*fieldInitList*]] [[*;;*]] } | *constExpr* **DUP** (*structInstList*)
structInstList	*structInstance* | *structInstList* , [[*;;*]] *structInstance*
structItem	*dataDir* | *generalDir* | *offsetDir* | *nestedStruct*
structTag	*id*
term	*simpleExpr* | **!** *simpleExpr*
text	*textLiteral* | *text character* | **!** *character text* | *character* | **!** *character*
textDir	*id textMacroDir ;;*

Nonterminal	Definition
textItem	*textLiteral* | *textMacroId* | % *constExpr*
textLen	*constExpr*
textList	*textItem* | *textList* , [[;;]] *textItem*
textLiteral	< *text* >;;
textMacroDir	**CATSTR** [[*textList*]] | **TEXTEQU** [[*textList*]] | **SIZESTR** *textItem* | **SUBSTR** *textItem* , *textStart* [[, *textLen*]] | **INSTR** [[*textStart* ,]] *textItem* , *textItem*
textMacroId	*id*
textStart	*constExpr*
titleDir	*titleType arbitraryText ;;*
titleType	**TITLE** | **SUBTITLE** | **SUBTTL**
type	*structTag* | *unionTag* | *recordTag* | *distance* | *dataType* | *typeId*
typedefDir	*typeId* **TYPEDEF** *qualifier*
typeId	*id*
unionTag	*id*
untilDir	**.UNTIL** *cExpr ;;* **.UNTILCXZ** [[*cxzExpr*]] *;;*
usesRegs	**USES** *regList*
whileBlock	**.WHILE** *cExpr ;;* *blockStatements ;;* **.ENDW**
whiteSpaceCharacter	ASCII 8, 9, 11–13, 32

Appendix C

Generating and Reading Assembly Listings

MASM creates an assembly listing of your source file whenever you select the appropriate option in PWB, use one of the related source code directives, or specify the /Fl option on the MASM command line. The assembly listing contains both the statements in the source file and the binary code (if any) generated for each statement. The listing also shows the names and values of all labels, variables, and symbols in your file.

The assembler creates tables for macros, structures, unions, records, segments, groups, and other symbols. These tables are placed at the end of the assembly listing. MASM lists only the types of symbols encountered in the program. For example, if your program has no macros, the symbol table does not have a macros section.

C.1 Generating Listing Files

MASM 6.0 provides several ways to generate a listing file. From within PWB, follow these steps:

1. From the "Options" menu, choose MASM Options.
2. In the MASM Options dialog box, choose Set Debug or Release Options.

The resulting dialog box for Set Debug or Release Options lists the choices summarized in Table C.1. This table also shows the equivalent directives you can use in your source code or the equivalent command-line options.

Table C.1 Options for Generating or Modifying Listing Files

To generate this information:	In PWB[1], select:	In source code, enter:	From command line, enter:
Default listing—includes all assembled lines	Generate Listing File	**.LIST** (default)	/Fl
Turn off all source listings (overrides all listing directives)	Generate Listing File (turn off)	**.NOLIST** (synonym = **.SFCOND**)	—
List all source lines, including false conditionals and generated code	Include All Source Lines	**.LISTALL**	/Fl /Sa
Show assembler-generated code	List Generated Instructions	—	/Fl /Sg
Include false conditionals[2]	List False Conditionals	**.LISTIF** (synonym = **.LFCOND**)	/Fl /Sx
Suppress listing of any subsequent conditional blocks whose condition is false	List False Conditionals (turn off)	**.NOLISTIF** (synonym = **.SFCOND**)	—
Toggle between **.LISTIF** and **.NOLISTIF**	—	**.TFCOND**	—
Suppress symbol table generation	Generate Symbol Table (turn off the default)	—	/Fl /Sn
List all processed macro statements	—	**.LISTMACROALL** (synonym = **.LALL**)	—
List only instructions, data, and segment directives in macros	—	**.LISTMACRO** (default) (synonym = **.XALL**)	—
Turn off all listing during macro expansion	—	**.NOLISTMACRO** (synonym = **.SALL**)	—
Specify title for each page (use only once per file)	—	**TITLE** *name*	/St
Specify subtitle for page	—	**SUBTITLE** *name*	/Ss
Designate page length and line width, increment section number, or generate page breaks	—	**PAGE** [[*length*,*width*]][[+]]	/Sp *length* /Sl *width*

[1] Select MASM Options from the "Options" menu. Then choose Set Dialog Options from the MASM Options dialog box.

[2] See Section 1.3.2.2, "Conditional Directives."

C.1.1 Generating a First Pass Listing

The /EP command-line option may be used to produce a listing during the assembler's first pass. This listing is printed to standard output and is suitable for processing by the assembler. A first pass listing can be helpful for locating problems when there are many errors, or when unmatched nesting errors occur.

C.1.2 Controlling the Contents of the Listing File

With source code directives you can vary the contents of the listing file for different sections of the source file, whereas (in the absence of source code directives) the PWB or command-line options affect the entire listing.

The /Fl command-line option enables a listing. Without /Fl, no listing is produced. The /S options are legal without /Fl, but they have no effect.

A file generated with /Fl shows all assembled source lines and provides a header at the beginning of the listing. It also adds a header before the symbol table and each section of the symbol table but does not add any page breaks between sections.

C.1.3 Controlling Listing Information on Macros

The only way to control the listing of macro expansions is with the source directives. The assembler always lists the full macro definition. The directives affect only expansion of macro calls. Macro comments are never listed in macro expansions. The default, **.LISTMACRO**, ignores comments and equates. The **.NOLISTMACRO** directive shows the initial macro call but not the source lines generated by the initial call or by recursive calls.

The assembler lists normal comments in macros only when you specify the **.LISTMACROALL** directive. This directive produces all statements processed during a macro expansion, including normal comments (preceded by a single semicolon) but not macro comments (preceded by a double semicolon).

C.1.4 Controlling the Page Format

With source code directives or command-line options, you can specify the line length, page length, title, and subtitle of the pages in a listing file. In PWB, you can enter listing file options in the "Additional Options" section of the MASM Options dialog box. Table C.1 gives the command-line options and source code listings for control of page format.

C.1.5 Precedence of Command-Line Options and Listing Directives

Since command-line options and source code directives can specify opposite behavior for the same listing file option, the assembler interprets the commands according to the precedence levels below. Selecting PWB options is equivalent to specifying /Fl /S*letter* on the command line:

- /Sa overrides any source code directives that suppress listing.
- Source code directives override all command-line options except /Sa.
- **.NOLIST** overrides other listing directives such as **.NOLISTIF** and **.LISTMACROALL**.
- The /Sx, /Ss, /Sp, and /Sl options set initial values for their respective features. Directives in the source file override these command-line options.

C.2 Reading the Listing File

The first column of the listing file gives the offset and binary code generated by the assembler. The next column gives the source statement exactly as it appears in the source file or as expanded by a macro. Various symbols and abbreviations in this column provide information about the code, as explained below.

C.2.1 Code Generated

The assembler lists the code generated from the statements of a source file. Each line has this syntax:

offset [[*code*]]

The *offset* is the offset from the beginning of the current segment to the code. If the statement generates code or data, *code* shows the numeric value in hexadecimal notation if the value is known at assembly time. If the value is calculated at run time, the assembler indicates what action is necessary to compute the value.

C.2.2 Error Messages

If any errors occur during assembly, each error message and error number appears directly below the statement where the error occurred. An example of an error line and message is shown below:

```
                                mov     ax, [dx][di]
listtst.asm(66): error A2031: must be index or base register
```

C.2.3 Symbols and Abbreviations

The assembler uses the symbols and abbreviations shown in Table C.2 to indicate addresses that need to be resolved by the linker or values that were generated in a special way. The example in this section illustrates many of these symbols. The numbers in column one correspond to the location of this symbol in the sample listing file.

The listing file was produced using "List-Generated Instructions" from PWB (or using /Fl /Sg from the command line).

Table C.2 Symbols and Abbreviations in Listings

Label	Character	Meaning
①	C	Line from include file
②	=	**EQU** or equal-sign (=) directive
③	*nn*[*xx*]	**DUP** expression: *nn* copies of the value *xx*
④	----	Segment/group address (linker must resolve)
④	R	Relocatable address (linker must resolve)
④	*	Assembler-generated code
⑤	E	External address (linker must resolve)
⑥	*n*	Macro-expansion nesting level (+ if more than 9)
⑦	!	Operator size override
⑧	&	Address size override
⑨	*nn*:	Segment override in statement
⑩	*nn*/	**REP** or **LOCK** prefix instruction

The sample listing file also shows the size of structures and unions in the first column.

```
Microsoft (R) Macro Assembler Version 6.00 Nov 13 01:27:05 1990
listtst.asm                                     Page 1 - 1

                                .MODEL  small, c
                                .386
                                .DOSSEG
                                .STACK  256
                                INCLUDE dos.mac
①                       C StrDef  MACRO   name1, text
                        C name1   BYTE    &text
                        C         BYTE    13d, 10d
                        C l&name1 EQU     LENGTHOF name1
                        C         ENDM
                        C
                        C Display MACRO   string
                        C         mov     ah, 09h
                        C         mov     dx, OFFSET string
                        C         int     21h
                        C         ENDM
                        C
②       = 0020            num     EQU     20h
                          COLOR   RECORD  b:1, r:3=1, i:1=1, f:3=7
        = 35              value   TEXTEQU %3 + num
        = 32              tnum    TEXTEQU %num
        = 04              strpos  TEXTEQU @InStr( , <person>, son> )

                          PutStr  PROTO   pMsg:PTR BYTE

        0004              DATE    STRUCT
        0000  05          month   BYTE    5
        0001  07          day     BYTE    7
        0002  07C3        year    WORD    1987
                          DATE    ENDS

        0002              U1      UNION
        0000  0028        fsize   WORD    40
                          bsize   BYTE    60
                          U1      ENDS

        0000                      .DATA

        0000 00000000     ddData  DWORD   ?
        0004 1F           text    COLOR   <>
        0005 09 16 07C3   today   DATE    <9,22,1987>
        0009 00           flag    BYTE    0
③       000A  001E [      buffer  WORD    30 DUP (0)
               0000
              ]
```

```
                                        StrDef   ending, "Finished."
0046 46 69 6E 69 73 68          1       ending   BYTE    "Finished."
     65 64 2E
004F 0D 0A                      1                BYTE    13d, 10d
= 0009                          1       lending  EQU     LENGTHOF ending
0051 54 68 69 73 20 69                  Msg      BYTE    "This is a string","0"
     73 20 61 20 73 74
     72 69 6E 67 30

                        float   TYPEDEF      REAL4
                        FPBYTE  TYPEDEF FAR  PTR BYTE
0062 ---- 0051 R       FPMSG   FPBYTE       Msg
                        PBYTE   TYPEDEF      PTR BYTE
                        NPWORD  TYPEDEF NEAR PTR WORD
                        PVOID   TYPEDEF      PTR
                        PPBYTE  TYPEDEF      PTR PBYTE

0000                                    .CODE
                                        .STARTUP
④  0000  B8 ---- R      *       mov     ax, DGROUP
   0003  8E D8          *       mov     ds, ax
   0005  8C D3          *       mov     bx, ss
   0007  2B D8          *       sub     bx, ax
   0009  C1 E3 04       *       shl     bx, 004h
   000C  8E D0          *       mov     ss, ax
   000E  03 E3          *       add     sp, bx

                        EXTERNDEF       work:NEAR
⑤  0010  E8 0000 E                      call    work

                                        Display ending
⑥  0013  B4 09           1              mov     ah, 09h
   0015  BA 0046 R       1              mov     dx, OFFSET ending
   0018  CD 21           1              int     21h

⑦  001A  66| A1 0000 R                  mov     eax, ddData
⑧  001E  67& FE 03                      inc     BYTE PTR [ebx]

                                        INVOKE  PutStr, ADDR msg
   0021  B8 0051 R      *       lea     ax, DGROUP:Msg
   0024  50             *       push    ax
   0025  E8 0042 R      *       call    PutStr
   0028  83 C4 02       *       add     sp, 00002h

   002B  B8 ---- R                  mov     ax, @data
   002E  8E C0                      mov     es, ax
   0030  B8 0063                    mov     ax, 'c'
⑨  0033  26: 8B 0E 0020             mov     cx, es:num
   0038  BF 0052                    mov     di, 82
⑩  003B  F2/ AE                     repne   scasb
   003D  57                         push    di
```

```
                                .EXIT
003E  B4 4C             *       mov     ah, 04Ch
0040  CD 21             *       int     021h

0042                    PutStr  PROC    pMsg:PTR BYTE
0042  55                *       push    bp
0043  8B EC              *       mov     bp, sp
0045  B4 02                      mov     ah, 02H
0047  8B 7E 04                   mov     di, pMsg
004A  8A 15                      mov     dl, [di]
                                 mov     ax, [dx][di]
isttst.asm(71): error A2031: must be index or base register
                                 .WHILE  (dl)
004C  EB 10             *       jmp     @C0001
004E                    *@C0002:
004E  CD 21                      int     21h
0050  47                         inc     di
0051  8A 15                      mov     dl, [di]
                                 .ENDW
0053                    *@C0001:
0053  0A D2             *       or      dl, dl
0055  75 02             *       jne     @C0002
                                 ret
0057  5D                *       pop     bp
0058  C3                *       ret     00000h
0059                    PutStr  ENDP

                                 END
```

C.2.4 Reading Tables in a Listing File

The tables at the end of a listing file list the macros, structures, unions, records, segments, groups, and symbols that appear in a source file. These tables are not printed in the sample listing, but this section summarizes the information.

Macro Table Lists all macros in the main file or the include files. Differentiates between macro functions and macro procedures.

Structures and Unions Table Provides the size in bytes of the structure or union and the offset of each field. The type of each field is also given.

Record Table "Width" gives the number of bits of the entire record. "Shift" provides the offset in bits from the low-order bit of the record to the low-order bit of the field. "Width" for fields gives the number of bits in the field. "Mask" gives the maximum value of the field, expressed in hexadecimal notation. "Initial" gives the initial value supplied for the field.

Type Table The "Size" column in this table gives the size of the **TYPEDEF** type in bytes, and the "Attr" column gives the base type for the **TYPEDEF** definition.

Segment and Group Table "Size" specifies whether the segment is 16 bit or 32 bit. "Length" gives the size of the segment in bytes. "Align" gives the segment alignment (**WORD**, **PARA**, and so on). "Combine" gives the combine type (Public, Stack, etc.). "Class" gives the segment's class (DATA, STACK, CODE, etc.).

Procedures, Parameters, and Locals Gives the types and offsets from BP of all parameters and locals defined in each procedure, as well as the size and memory location of each procedure.

Symbol Table All symbols (except names for macros, structures, unions, records, and segments) are listed in a symbol table at the end of the listing. The "Name" column lists the names in alphabetical order. The "Type" column lists each symbol's type.

The length of a multiple-element variable, such as an array or string, is the length of a single element, not the length of the entire variable.

If the symbol represents an absolute value defined with an **EQU** or equal-sign (=) directive, the "Value" column shows the symbol's value. The value may be another symbol, a string, or a constant numeric value (in hexadecimal), depending on the type. If the symbol represents a variable or label, the "Value" column shows the symbol's hexadecimal offset from the beginning of the segment in which it is defined.

The "Attr" column shows the attributes of the symbol. The attributes include the name of the segment (if any) in which the symbol is defined, the scope of the symbol, and the code length. A symbol's scope is given only if the symbol is defined using the **EXTERN** and **PUBLIC** directives. The scope can be external, global, or communal. The "Attr" column is blank if the symbol has no attribute.

Appendix D

MASM Reserved Words

This appendix lists the reserved words recognized by MASM. They are divided primarily by their use in the language. The primary categories are

- Operands and symbols
- Registers
- Operators and directives
- Processor instructions
- Coprocessor instructions

Reserved words in MASM 6.0 are reserved under all CPU modes. Words enabled in **.8086** mode, the default, can be used in all higher CPU modes. To use words from subcategories such as "Special Operands for the 80386" (Section D.1.1) requires **.386** mode or higher.

You can disable the recognition of any reserved word specified in this appendix by setting the **NOKEYWORD** option for the **OPTION** directive. Once disabled, the word can be used in any way as a user-defined symbol (provided the word is a valid identifier). If you want to remove the **STR** instruction, the **MASK** operator, and the **NAME** directive, for instance, from the set of words MASM recognizes as reserved, add this statement to your program:

```
OPTION NOKEYWORD:<STR MASK NAME>
```

* Words in this appendix identified with an asterisk (*) are new to MASM 6.0.

D.1 Operands and Symbols

The words on the two lists in this section are the operands to certain directives. They have special meaning to the assembler. The words on the first list are not reserved words. They can be used in every way as normal identifiers, without affecting their use as operands to directives. The assembler interprets their use from context.

Even though the words on the first list are not reserved, they should not be defined to be text macros or text macro functions. If they are, they will not be recognized in their special contexts. The assembler does not give a warning if such a redefinition occurs.

ABS	LARGE	NOTHING
ALL	LISTING*	NOTPUBLIC*
ASSUMES	LJMP*	OLDMACROS*
AT	LOADDS*	OLDSTRUCTS*
CASEMAP*	M510*	OS_DOS*
COMMON	MEDIUM	OS_OS2*
COMPACT	MEMORY	PARA
CPU*	NEARSTACK*	PRIVATE*
DOTNAME*	NODOTNAME*	PROLOGUE*
EMULATOR*	NOEMULATOR*	RADIX*
EPILOGUE*	NOKEYWORD*	READONLY*
ERROR*	NOLJMP*	REQ*
EXPORT*	NOM510*	SCOPED*
EXPR16*	NONE	SMALL
EXPR32*	NONUNIQUE*	STACK
FARSTACK*	NOOLDMACROS*	TINY
FLAT	NOOLDSTRUCTS*	USE16
FORCEFRAME	NOREADONLY*	USE32
HUGE	NOSCOPED*	USES
LANGUAGE*	NOSIGNEXTEND*	

These operands are reserved words. Reserved words are never case sensitive.

$	FAR16*	REAL10*
?	FORTRAN	SBYTE*
@B	FWORD	SDWORD*
@F	NEAR	SIGN?*
ADDR*	NEAR16*	STDCALL*
BASIC	OVERFLOW?*	SWORD*
BYTE	PARITY?*	SYSCALL*
C	PASCAL	TBYTE
CARRY?*	QWORD	VARARG*
DWORD	REAL4*	WORD
FAR	REAL8*	ZERO?*

* Words in this appendix identified with an asterisk (*) are new to MASM 6.0.

D.1.1 Special Operands for the 80386/486

FLAT*
NEAR32*
FAR32*

D.1.2 Predefined Symbols

Unlike most MASM reserved words, the predefined symbols are case sensitive.

@CatStr*
@code
@CodeSize
@Cpu
@CurSeg
@data
@DataSize
@Date*
@Environ*
@fardata
@fardata?
@FileCur*
@FileName
@InStr*
@Interface*
@Line*
@Model*
@SizeStr*
@stack*
@SubStr*
@Time*
@Version
@WordSize

D.2 Registers

AH
AL
AX
BH
BL
BP
BX
CH
CL
CR0
CR2
CR3
CS
CX
DH
DI
DL
DR0
DR1
DR2
DR3
DR6
DR7
DS
DX
EAX
EBP
EBX
ECX
EDI
EDX
ES
ESI
ESP
FS
GS
SI
SP
SS
ST
TR3*
TR4*
TR5*
TR6
TR7

* Words in this appendix identified with an asterisk (*) are new to MASM 6.0.

D.3 Operators and Directives

.186
.286
.286C
.286P
.287
.386
.386C
.386P
.387
.486*
.486P*
.8086
.8087
.ALPHA
.BREAK*
.CODE
.CONST
.CONTINUE*
.CREF
.DATA
.DATA?
.DOSSEG*
.ELSE*
.ELSEIF*
.ENDIF*
.ENDW*
.ERR
.ERR1
.ERR2
.ERRB
.ERRDEF
.ERRDIF
.ERRDIFI
.ERRE
.ERRIDN
.ERRIDNI
.ERRNB
.ERRNDEF
.ERRNZ
.EXIT*
.FARDATA
.FARDATA?
.IF*
.LALL
.LFCOND
.LIST
.LISTALL*
.LISTIF*
.LISTMACRO*
.LISTMACROALL*
.MODEL
.NO87*
.NOCREF*
.NOLIST*
.NOLISTIF*
.NOLISTMACRO*
.RADIX
.REPEAT*
.SALL
.SEQ
.SFCOND
.STACK
.STARTUP*
.TFCOND
.TYPE
.UNTIL*
.UNTILCXZ*
.WHILE*
.XALL
.XCREF
.XLIST
.XLISTIF
.XLISTMACRO
ALIAS*
ALIGN
ASSUME
CATSTR
COMM
COMMENT
DB
DD
DF
DOSSEG
DQ
DT
DUP
DW
ECHO*
ELSE
ELSEIF
ELSEIF1
ELSEIF2
ELSEIFB
ELSEIFDEF
ELSEIFDIF
ELSEIFDIFI
ELSEIFE
ELSEIFIDN
ELSEIFIDNI
ELSEIFNB
ELSEIFNDEF
END
ENDIF
ENDM
ENDP
ENDS
EQ
EQU
EVEN
EXITM
EXTERN*
EXTERNDEF*
EXTRN
FOR*
FORC*
GE
GOTO*
GROUP
GT
HIGH

* Words in this appendix identified with an asterisk (*) are new to MASM 6.0.

HIGHWORD*
IF
IF1
IF2
IFB
IFDEF
IFDIF
IFDIFI
IFE
IFIDN
IFIDNI
IFNB
IFNDEF
INCLUDE
INCLUDELIB
INSTR
INVOKE*
IRP
IRPC
LABEL
LE
LENGTH
LENGTHOF*
LOCAL
LOW
LOWWORD*
LROFFSET*
LT
MACRO
MASK
MOD
.MSFLOAT
NAME
NE
OFFSET
OPATTR*
OPTION*
ORG
%OUT
PAGE
POPCONTEXT*
PROC
PROTO*
PTR
PUBLIC
PURGE
PUSHCONTEXT*
RECORD
REPEAT*
REPT
SEG
SEGMENT
SHORT
SIZE
SIZEOF*
SIZESTR
STRUC
STRUCT*
SUBSTR
SUBTITLE*
SUBTTL
TEXTEQU*
THIS
TITLE
TYPE
TYPEDEF*
UNION*
WHILE*
WIDTH

D.4 Processor Instructions

MASM processor instructions are not case sensitive.

D.4.1 8086/8088 Processor Instructions

AAA
AAD
AAM
AAS
ADC
ADD
AND
CALL
CBW
CLC
CLD
CLI
CMC
CMP
CMPS
CMPSB
CMPSW
CWD
DAA
DAS
DEC
DIV
ESC
HLT
IDIV
IMUL
IN
INC
INT
INTO
IRET
JA
JAE
JB
JBE
JC

* Words in this appendix identified with an asterisk (*) are new to MASM 6.0.

JCXZ
JE
JG
JGE
JL
JLE
JMP
JNA
JNAE
JNB
JNBE
JNC
JNE
JNG
JNGE
JNL
JNLE
JNO
JNP
JNS
JNZ
JO
JP
JPE
JPO
JS
JZ
LAHF
LDS
LEA
LES
LODS
LODSB
LODSW
LOOP
LOOPE
LOOPEW*
LOOPNE
LOOPNEW*
LOOPNZ
LOOPNZW*
LOOPW*
LOOPZ
LOOPZW*
MOV
MOVS
MOVSB
MOVSW
MUL
NEG
NOP
NOT
OR
OUT
POP
POPF
PUSH
PUSHF
RCL
RCR
RET
RETF
RETN
ROL
ROR
SAHF
SAL
SAR
SBB
SCAS
SCASB
SCASW
SHL
SHR
STC
STD
STI
STOS
STOSB
STOSW
SUB
TEST
WAIT
XCHG
XLAT
XLATB
XOR

D.4.2 80186 Processor Instructions

BOUND
ENTER
INS
INSB
INSW
LEAVE
OUTS
OUTSB
OUTSW
POPA
PUSHA
PUSHW*

* Words in this appendix identified with an asterisk (*) are new to MASM 6.0.

D.4.3 80286 Processor Instructions

ARPL	SIDT	VERR
LAR	SLDT	VERW
LSL	SMSW	
SGDT	STR	

D.4.4 80286 and 80386 Privileged-Mode Instructions

CLTS	LIDT	LMSW
LGDT	LLDT	LTR

D.4.5 80386 Processor Instructions

BSF	LSS	SETNAE
BSR	MOVSD	SETNB
BT	MOVSX	SETNBE
BTC	MOVZX	SETNC
BTR	OUTSD	SETNE
BTS	POPAD	SETNG
CDQ	POPFD	SETNGE
CMPSD	PUSHAD	SETNL
CWDE	PUSHD*	SETNLE
INSD	PUSHFD	SETNO
IRETD	SCASD	SETNP
IRETDF*	SETA	SETNS
IRETF*	SETAE	SETNZ
JECXZ	SETB	SETO
LFS	SETBE	SETP
LGS	SETC	SETPE
LODSD	SETE	SETPO
LOOPD*	SETG	SETS
LOOPED*	SETGE	SETZ
LOOPNED*	SETL	SHLD
LOOPNZD*	SETLE	SHRD
LOOPZD*	SETNA	STOSD

* Words in this appendix identified with an asterisk (*) are new to MASM 6.0.

D.4.6 80486 Processor Instructions

BSWAP*	INVD*	WBINVD*
CMPXCHG*	INVLPG*	XADD*

D.4.7 Instruction Prefixes

LOCK	REPE	REPNZ
REP	REPNE	REPZ

D.5 Coprocessor Instructions

MASM coprocessor instructions are not case sensitive.

D.5.1 8087 Coprocessor Instructions

F2XM1	FDIVRP	FLD
FABS	FENI	FLD1
FADD	FFREE	FLDCW
FADDP	FIADD	FLDENV
FBLD	FICOM	FLDENVW*
FBSTP	FICOMP	FLDL2E
FCHS	FIDIV	FLDL2T
FCLEX	FIDIVR	FLDLG2
FCOM	FILD	FLDLN2
FCOMP	FIMUL	FLDPI
FCOMPP	FINCSTP	FLDZ
FDECSTP	FINIT	FMUL
FDISI	FIST	FMULP
FDIV	FISTP	FNCLEX
FDIVP	FISUB	FNDISI
FDIVR	FISUBR	FNENI

* Words in this appendix identified with an asterisk (*) are new to MASM 6.0.

FNINIT
FNOP
FNSAVE
FNSAVEW*
FNSTCW
FNSTENV
FNSTENVW*
FNSTSW
FPATAN
FPREM
FPTAN
FRNDINT
FRSTOR
FRSTORW*
FSAVE
FSAVEW*
FSCALE
FSQRT
FST
FSTCW
FSTENV
FSTENVW*
FSTP
FSTSW
FSUB
FSUBP
FSUBR
FSUBRP
FTST
FWAIT
FXAM
FXCH
FXTRACT
FYL2X
FYL2XP1

D.5.2 80287 Privileged-Mode Instruction

FSETPM

D.5.3 80387 Instructions

FCOS
FLDENVD*
FNSAVED*
FNSTENVD*
FPREM1
FRSTORD*
FSAVED*
FSIN
FSINCOS
FSTENVD*
FUCOM
FUCOMP
FUCOMPP

* Words in this appendix identified with an asterisk (*) are new to MASM 6.0.

Appendix E
Default Segment Names

If you use simplified segment directives by themselves, you do not need to know the names assigned for each segment. However, it is possible to mix full segment definitions with simplified segment directives, in which case you need to know the segment names.

Table E.1 shows the default segment names created by each directive.

If you use **.MODEL**, a _TEXT segment is always defined, even if all **.CODE** directives specify a name. The default segment name used as part of far-code segment names is the filename of the module. The default name associated with the **.CODE** directive can be overridden, as can the default names for **.FARDATA** and **.FARDATA?**.

The segment and group table at the end of listings always shows the actual segment names. However, the **GROUP** and **ASSUME** statements generated by the **.MODEL** directive are not shown in listing files. For a program that uses all possible segments, group statements equivalent to the following would be generated:

```
DGROUP      GROUP    _DATA, CONST, _BSS, STACK
```

For the tiny model, these **ASSUME** statements would be generated:

```
ASSUME   cs:DGROUP, ds:DGROUP, ss:DGROUP
```

For small and compact models with **NEARSTACK**, these **ASSUME** statements would be generated:

```
ASSUME   cs: _TEXT, ds:DGROUP, ss:DGROUP
```

For medium, large, and huge models with **NEARSTACK**, these **ASSUME** statements would be generated:

```
ASSUME   cs:name_TEXT, ds:DGROUP, ss:DGROUP
```

Table E.1 Default Segments and Types for Standard Memory Models

Model	Directive	Name	Align	Combine	Class	Group
Tiny	.CODE	_TEXT	WORD	PUBLIC	'CODE'	DGROUP
	.FARDATA	FAR_DATA	PARA	PRIVATE	'FAR_DATA'	
	.FARDATA?	FAR_BSS	PARA	PRIVATE	'FAR_BSS'	
	.DATA	_DATA	WORD	PUBLIC	'DATA'	DGROUP
	.CONST	CONST	WORD	PUBLIC	'CONST'	DGROUP
	.DATA?	_BSS	WORD	PUBLIC	'BSS'	DGROUP
Small	.CODE	_TEXT	WORD	PUBLIC	'CODE'	
	.FARDATA	FAR_DATA	PARA	PRIVATE	'FAR_DATA'	
	.FARDATA?	FAR_BSS	PARA	PRIVATE	'FAR_BSS'	
	.DATA	_DATA	WORD	PUBLIC	'DATA'	DGROUP
	.CONST	CONST	WORD	PUBLIC	'CONST'	DGROUP
	.DATA?	_BSS	WORD	PUBLIC	'BSS'	DGROUP
	.STACK	STACK	PARA	STACK	'STACK'	DGROUP*
Medium	.CODE	*name*_TEXT	WORD	PUBLIC	'CODE'	
	.FARDATA	FAR_DATA	PARA	PRIVATE	'FAR_DATA'	
	.FARDATA?	FAR_BSS	PARA	PRIVATE	'FAR_BSS'	
	.DATA	_DATA	WORD	PUBLIC	'DATA'	DGROUP
	.CONST	CONST	WORD	PUBLIC	'CONST'	DGROUP
	.DATA?	_BSS	WORD	PUBLIC	'BSS'	DGROUP
	.STACK	STACK	PARA	STACK	'STACK'	DGROUP*
Compact	.CODE	_TEXT	WORD	PUBLIC	'CODE'	
	.FARDATA	FAR_DATA	PARA	PRIVATE	'FAR_DATA'	
	.FARDATA?	FAR_BSS	PARA	PRIVATE	'FAR_BSS'	
	.DATA	_DATA	WORD	PUBLIC	'DATA'	DGROUP
	.CONST	CONST	WORD	PUBLIC	'CONST'	DGROUP
	.DATA?	_BSS	WORD	PUBLIC	'BSS'	DGROUP
	.STACK	STACK	PARA	STACK	'STACK'	DGROUP*

Table E.1 *(continued)*

Model	Directive	Name	Align	Combine	Class	Group
Large or huge	**.CODE**	*name*_TEXT	**WORD**	**PUBLIC**	'CODE'	
	.FARDATA	FAR_DATA	**PARA**	**PRIVATE**	'FAR_DATA'	
	.FARDATA?	FAR_BSS	**PARA**	**PRIVATE**	'FAR_BSS'	
	.DATA	_DATA	**WORD**	**PUBLIC**	'DATA'	DGROUP
	.CONST	CONST	**WORD**	**PUBLIC**	'CONST'	DGROUP
	.DATA?	_BSS	**WORD**	**PUBLIC**	'BSS'	DGROUP
	.STACK	STACK	**PARA**	**STACK**	'STACK'	DGROUP*
Flat	**.CODE**	_TEXT	**DWORD**	**PUBLIC**	'CODE'	
	.FARDATA	_DATA	**DWORD**	**PUBLIC**	'DATA'	
	.FARDATA?	_BSS	**DWORD**	**PUBLIC**	'BSS'	
	.DATA	_DATA	**DWORD**	**PUBLIC**	'DATA'	
	.CONST	CONST	**DWORD**	**PUBLIC**	'CONST'	
	.DATA?	_BSS	**DWORD**	**PUBLIC**	'BSS'	
	.STACK	STACK	**DWORD**	**PUBLIC**	'STACK'	

* unless the stack type is **FARSTACK**

Appendix F
Error Messages

This appendix lists MASM 6.0 error and warning messages. Each message includes an explanation of what went wrong and what action to take to correct the problem.

Error numbers consist of a one- or two-letter prefix and four digits. The first digit indicates a severity level:

- Fatal errors stop execution and are numbered 1*xxx*.
- Errors numbered 2*xxx* are usually nonfatal; execution continues if possible.
- Warnings do not stop execution but indicate a possible problem; they are numbered 4*xxx*.

Error messages may also display the input file and line number where the error occurred.

F.1 BIND Error Messages

This section lists error messages generated by the Microsoft Bind Utility (BIND). BIND errors (U12*xx*) are always fatal.

Number	BIND Error Message
U1250	**invalid executable file**

The executable file cannot be bound. Either the header is invalid, or the executable file has an invalid magic number.

Repeat with a backup version of the executable file, or rebuild the file and repeat.

U1251 **cannot create file :** *filename*

BIND was unable to create a temporary file or the map file, probably because the disk was full.

U1252 **unrecoverable I/O error**

The system returned an I/O error when reading the executable file.

U1253 **cannot open file :** *filename*

The given file could not be opened.

The following are possible causes of this error:

- The file does not exist.
- The file is in use by another process.
- The disk is full.

U1254 **structure error in .EXE file**

The executable file has an invalid structure.

Rebuild the file.

U1255 **structure error in .LIB file :** *filename*

The given library file has an invalid structure. Library files must conform to Microsoft object module format.

Repeat with a backup version of the library file, or rebuild the library and repeat.

U1256 **out of memory**

There was insufficient memory for BIND to run.

U1257 **too many libraries specified,** *number* **allowed**

The BIND command line contained more than the given number of libraries.

Combine some libraries.

U1258 **resource tables not supported**

Protected-mode executable files that use resource tables cannot be bound because when the bound executable file runs in DOS mode the resources would be unknown.

U1259 **internal error — Lname not found :** *lname*

BIND encountered an internal error.

Repeat the attempt with a new copy of BIND. If the problem persists, note the circumstances of the error and notify Microsoft Corporation by following the instructions on the Microsoft Product Assistance Request form at the back of one of your manuals.

U1260 **import by ordinal not defined :** *dllname.ordinal*

The given DLL does not contain a function with the given ordinal value. As a result, fixups from function calls to this function cannot be made.

U1261 **system call *syscall* return error**

BIND encountered an internal error.

Repeat the attempt with a new copy of BIND. If the problem persists, note the circumstances of the error and notify Microsoft Corporation by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

U1262 **cannot find LINK.EXE in path**

BIND could not find LINK.EXE in any directory specified by the PATH environment variable.

BIND needs the linker to complete the binding operation.

U1263 **error during link of file, link error status : *status***

A linking error occurred during the LINK session invoked by BIND.

The following are possible causes of this error:

- Unresolved references exist in the files. BIND could not resolve references with API.LIB or other support libraries.
- APILMR.OBJ was used when the executable file was created, and LINK gave error L2044, `symbol multiply defined, use /NOE`. Relink using the LINK option /NOE, then rebind.
- There was not enough memory.
- A disk I/O error occurred.

U1264 **unrecognized option : *option***

The BIND command line contained the given unrecognized option.

U1265 **unrecognized argument : *string***

The given string is not a valid argument for the option it was specified with.

U1266 **no *infile* specified**

No executable file to be bound was named on the BIND command line.

U1267 **no *outfile* specified**

The option for naming an outfile, /o, was given on the command line, but no file was named.

U1268 **duplicate infile name given : *filename***

The given file was named in more than one place on the BIND command line.

U1269 **duplicate global name : *name***

The given global name was defined in more than one place in the specified libraries, making a unique fixup impossible.

This error can be caused by specifying both OS2.LIB and DOSCALLS.LIB. To correct the error, specify only OS2.LIB.

U1270 **terminated by user**

BIND was halted by CTRL+C or CTRL+BREAK.

U1271 **insufficient disk space**

There was not enough room on the disk. BIND creates temporary files that take up disk space.

Make some room on the disk and repeat.

U1272 **cannot bind a PROTMODE executable**

The module-definition file used to create the executable file contained a **PROTMODE** statement. This statement creates an executable file that cannot be run under DOS and prevents the file from being bound.

F.2 CodeView Error Messages

CodeView displays an error message whenever it detects a command it cannot execute. Most errors terminate the CodeView command in error, but do not terminate the debugger. Start-up errors terminate CodeView.

Depending on the context of the error, CodeView may display only the text of the message without the error number. This section is organized in alphabetical order by message text.

In some cases, CodeView may display the error number by itself. To obtain the error message and an explanation of the error in thoses cases, use online help. Click the right mouse button on the error number or use the Help (**H**) Command-window command. For example,

```
H CV1020
```

displays help for the error `Divide by zero`.

Error Message

Access denied (CV0013)

A specified file's permission setting does not allow the required access.

One of the following may have occurred:

- An attempt was made to write to a read-only file.
- A locking or sharing violation occurred.
- An attempt was made to open a directory instead of a file.

Address of register variable cannot be watched (CV1049)

An attempt was made to evaluate the address of a register variable. A register variable can be watched but not the address of a register variable.

One of the following occurred:

- The variable was declared as a register variable. Recompile the program with the register declaration removed.
- The optimizer converted an ordinary variable into a register variable to speed up execution. Recompile the program using the /Od option to turn optimization off.
- The function was defined with **_fastcall**, causing parameters to be passed in registers. Remove the **_fastcall** designation and recompile.

All threads blocked (CV3502)

The block may be due to a request for a system service semaphore. When the semaphore is cleared, the block will clear.

The block may also be due to a deadlock situation that will not clear until one or more of the threads are terminated.

Arg list too long (CV0007)

CodeView is not able to restart the program being debugged because the number of arguments to the executable program exceeds the limit of 128.

Argument to IMAG/DIMAG must be simple type (CV1121)

An invalid argument was specified to **IMAG** or **DIMAG**, such as an array with no subscripts.

Array must have subscript (CV1101)

An array was specified without any subscripts, such as `IARRAY+2`. A correct example would be `IARRAY(1)+2`.

Bad integer or real constant (CV1105)

An illegal numeric constant was specified in an expression.

Bad intrinsic function (CV1106)

An illegal intrinsic function name was specified in an expression.

Bad subscript (CV1100)

An illegal subscript expression was specified for an array.

For example, `IARRAY(3.3)` and `IARRAY((3,3))` are illegal. The correct expression is `IARRAY(3,3)`.

Badly formed type (CV1009)

CodeView detected corrupt information in the symbol table of the file being debugged.

Note the circumstances of the error and notify Microsoft Corporation by following the instructions on the Microsoft Product Assistance Request form at the back of one of your manuals.

Breakpoint number or '*' expected (CV1006)

A breakpoint was specified without a number or asterisk.

A Breakpoint Clear (**BC**), Breakpoint Disable (**BD**), or Breakpoint Enable (**BE**) command requires one or more numbers to specify the breakpoints, or an asterisk to specify all breakpoints.

For example, the following command causes this error:

```
bc a
```

Cannot cast complex constant component into REAL (CV1112)

Both the real and imaginary components of a **COMPLEX** constant must be compatible with the type **REAL**.

Cannot cast IMAG/DIMAG argument to COMPLEX (CV1122)

Arguments to **IMAG** and **DIMAG** must be simple numeric types.

Cannot create CURRENT.STS (CV1063)

CodeView could not find an existing state file (CURRENT.STS), and it tried to create one but failed.

One of the following may have occurred:

- There was not enough space either on the disk containing the program to be debugged or on the disk pointed to by the INIT environment variable.
- There were not enough free file handles. In DOS, increase the number of file handles by changing the FILES setting in CONFIG.SYS to allow a larger number of open files. FILES=20 is the recommended setting. In OS/2, if multiple processes are running, removing one or more of them may release enough file handles.
- The environment variable INIT pointed to a directory that does not exist. If the variable points to more than one directory, the first directory listed does not exist.

Cannot create \QUEUES\CVP (CV3601)

One or more of the required queues could not be created.

When debugging multiprocess programs, CodeView creates multiple copies of itself that intercommunicate through queues.

The failure to create queues may be due to lack of memory, or to having too many OS/2 processes running at one time.

Cannot create \SEM\CVP (CV3605)

The required semaphore could not be assigned.

When debugging multiprocess programs, CodeView allocates areas of shared memory for interprocess communication.

The failure to assign a semaphore may be due to lack of memory or to having too many OS/2 processes running at one time.

Cannot create \SHAREMEM\CVP (CV3604)

The required shared memory could not be assigned.

When debugging multiprocess programs, CodeView allocates areas of shared memory for interprocess communication.

The failure to assign shared memory may be due to lack of memory or to having too many OS/2 processes running at one time.

Cannot open CV.EXE (CV1310)

An error occurred while CV.EXE was being opened.

One of the following may have occurred:

- The file could be corrupt. Copy CV.EXE from the original disks and retry.
- The operating system could not find CV.EXE due to a disk error.
- There were not enough free file handles. In DOS, increase the number of file handles by changing the FILES setting in CONFIG.SYS to allow a larger number of open files. FILES=20 is the recommended setting. In OS/2, if multiple processes are running, removing one or more of them may release enough file handles.

If the error recurs, note the circumstances of the error and notify Microsoft Corporation by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

Cannot open \QUEUES\CVP (CV3602)

One or more of the required queues could not be opened.

When debugging multiprocess programs, CodeView creates multiple copies of itself that intercommunicate through queues.

The failure to open queues may be due to lack of memory or to having too many OS/2 processes running at one time.

Cannot open \SHAREMEM\CVP (CV3606)

The required shared memory could not be opened.

When debugging multiprocess programs, CodeView tries to access areas of shared memory for interprocess communication.

The failure to open shared memory may be due to lack of memory or to having too many OS/2 processes running at one time.

Cannot open \SEM\CVP (CV3607)

The required semaphore could not be opened.

When debugging multiprocess programs, CodeView tries to access areas of shared memory for interprocess communication.

The failure to open a semaphore may be due to lack of memory or to having too many OS/2 processes running at one time.

Cannot read CV.EXE (CV1311)

An error occurred while CV.EXE was being read. Possibly the operating system could not find CV.EXE due to a disk error.

If the error recurs, note the circumstances of the error and notify Microsoft Corporation by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

Cannot read file (CV5004)

A file was selected from a dialog box, and CodeView then opened the file. The read process failed while the file was being read.

Read the file again. If the second read fails, exit and restart CodeView. If the read process still fails, the file may be corrupt.

Cannot read this version of CURRENT.STS (CV1054)

The state file (CURRENT.STS) has a version number that is not recognized by this version of CodeView.

Check the directories for older copies and delete them.

Cannot restart program, out of system resources (CV3611)

The operating system reached its limit of one of the following resources:

- Memory
- Screen groups
- Threads

Cannot select (CV5001)

The cursor was not on the same line as an automatically selectable symbol.

Cannot understand entry in TOOLS.INI or CURRENT.STS (CV1056)

At least one line in the given file (either the state file or the TOOLS.INI file) could not be interpreted.

On start-up, CodeView reads the state file (CURRENT.STS) and the TOOLS.INI file (if the latter is available).

Examine the given file to find the problem.

Cannot use second monitor from VIO window (CV3610)

The operating system cannot support a second monitor from a virtual I/O window (Presentation-Manager text window).

Cannot use struct or union as scalar (CV1025)

A structure or union was used in an expression, but no element was specified.

When requesting display of a structure or union variable, the name of the variable may appear by itself, without a field qualifier. If a structure or union is used in an expression, it must be qualified with the specific element desired.

Specify the element whose value is to be used in the expression.

Character constant too long (CV1109)

A character constant was specified that is too long for the FORTRAN expression evaluator (the limit is 126 bytes).

Use the Radix (**N**) command to change the radix.

Character too big for current radix (CV1120)

A radix was specified in a constant that is larger than the current radix.

Command incompatible with history (CV2202)

The command entered is illegal while recording because it changes the state of CodeView and/or the program being debugged.

For example, the Restart (**L**) command cannot be used during recording.

Turn off history to use the command.

Constant too big (CV1028)

CodeView cannot accept an unsigned integer constant larger than 4,294,967,295 (0xFFFFFFFF hex), or a floating-point constant whose magnitude is larger than approximately 1.8E+308.

Corrupt debug OMF detected in *file*, discarding source line information (CV2206)

The linker used was not the current version of the Microsoft linker.

Conditions that require the most current linker include use of the **alloc_text** pragma in a C program and use of multiple segments in an assembly-language module.

Corrupted CV.EXE (CV1318)

The CV.EXE file has been corrupted. Copy CV.EXE from the original disks and retry.

If the error recurs, note the circumstances of the error and notify Microsoft Corporation by following the instructions on the Microsoft Product Assistance Request form at the back of one of your manuals.

CURRENT.STS not found—creating default (CV1057)

The state file (CURRENT.STS) could not be located at start-up, so CodeView created a state file.

Divide by zero (CV1020)

The expression contains a divisor of zero, which is illegal. This divisor might be the literal number zero, or it might be an expression that evaluates to zero.

/E: EMM driver not loaded (CV1304)

The EMM driver must be installed in order to use expanded memory.

/E: EMM internal error (CV1309)

An unexpected error from the EMM driver occurred. The driver may be corrupted or may have malfunctioned.

If replacing the EMM driver does not correct the problem, note the circumstances of the error and notify Microsoft Corporation by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

/E: EMM not LIM 4.0 or later (CV1305)

The EMM driver must be LIM EMS version 4.0 or later in order to contain the calls needed for CodeView to use expanded memory.

/E: no EMM handle available (CV1306)

No handle is available for the CodeView overlay code.

One of the following may be a solution:

- If possible, increase the number of EMM handles that are allocated when the EMM driver is loaded.
- If multiple applications are running, remove one or more of the applications that use expanded memory. This should free enough handles to permit CodeView to run properly.
- Some memory may have become inaccessible due to program error. Reboot in order to free the memory.

/E: not enough free expanded memory (CV1308)

There is currently not enough room in expanded memory to load overlays.

One of the following may be a solution:

- Decrease the size of memory allocated to SMARTDRV.SYS or RAMDRIVE.SYS to free expanded memory.
- Reconfigure the EMM driver or hardware to allocate more expanded memory.

/E: not enough total expanded memory (CV1307)

There is not enough room in all of expanded memory to load overlays. Freeing expanded memory will not help.

If possible, reconfigure the EMM driver or hardware to allocate more expanded memory.

EMM error (CV3010)

An unknown expanded memory error has occurred.

One of the following may have occurred:

- The EMM driver may be corrupted or may have malfunctioned. Reload the driver and retry.
- There was a disk error.

If the error recurs, note the circumstances of the error and notify Microsoft Corporation by following the instructions on the Microsoft Product Assistance Request form at the back of one of your manuals.

EMM hardware error (CV3001)

An error has occurred in expanded memory.

Exit CodeView and reboot the computer. If this does not correct the problem, the expanded memory board may need service.

EMM memory not found (CV3011)

CodeView cannot find expanded memory.

The EMM driver or expanded memory board is not installed, or the board has a malfunction.

EMM software error (CV3000)

An error has occurred in the EMM driver.

Exit CodeView and reboot the computer. If the problem recurs, replace the EMM driver file with a fresh copy from the EMM driver's distribution disk. If this does not correct the problem, the expanded memory board may need service.

Executable file format error (CV0008)

The system is not able to load the program to be debugged. The file is not an executable file, or it has an invalid format for this operating system.

Try to run the program outside of CodeView.

Expression not a memory address (CV1050)

The expression entered does not evaluate to an address. An address must be a numeric value.

An lvalue (so called because it appears on the left side of an assignment statement) is an expression that refers to a memory location.

For example, `buffer[count]` is a valid lvalue because it points to a specific memory location. The logical comparison `zed != 0` is not a valid lvalue because it evaluates to TRUE or FALSE, not a memory address.

Expression too complex (CV1019)

The expression entered was too complex for the amount of storage space available to the expression evaluator.

Overflow usually occurs because of too many pending calculations. Rearrange the expression so that each component of the expression can be evaluated immediately, rather than having to wait for other parts of the expression to be calculated.

Extra input ignored (CV1003)

The first part of the command line was interpreted correctly.

The remainder of the line could not be interpreted or was unnecessary.

File error (CV1041)

CodeView could not write to the disk.

One of the following may have occurred:

- There was not enough space on the disk.
- The file was locked by another process.

Flip/Swap option off—application output lost (CV1043)

The program being debugged wrote to the display when Flip/Swap was off. The program output was lost.

When flipping is on, video page 1 is normally reserved for CodeView, while programs by default write to video page 0. Programs that write to video page 1 must be debugged with swapping on.

Turn Flip/Swap on to be able to view program output.

Floating-point support not loaded (CV1048)

An attempt was made to access the math processor registers in a program that does not use floating-point arithmetic.

Several situations can cause this error:

- The math processor registers can only be accessed through the floating-point library code. This code is not loaded if the program does not perform floating-point calculations.
- If the program does not use floating-point instructions, this error may occur when attempting to access the math processor before any floating-point instructions have been performed. The C run-time library includes a floating-point instruction near the beginning so that the math processor registers are always accessible.
- If a floating-point instruction occurs in an assembly-language routine before such an instruction occurs in the C code that calls the routine, this error occurs.

Function argument may not be byte register (CV4058)

The register specified in the function does not accept a byte value. The register must be assigned a word or doubleword value.

Function call before stack frame initialization (CV1026)

A function call cannot be executed until after the BP (stack frame) register has been initialized.

Run the program to a statement that follows initialization of the BP register. This is usually set up as the first statement in the first function of the program.

Function returning struct/union not supported (CV1060)

CodeView cannot evaluate a function that returns a structure or union variable.

I/O error (CV0005)

An attempt was made to access an address that is not accessible to the program being debugged.

Check the previous command for numeric constants used as addresses and for pointers used for indirection.

Illegal instruction (CV4001)

An assembler instruction was not recognized.

The instruction may have been mistyped.

Illegal operand (CV4003)

The specified operand is not permitted with this instruction.

The operand name may have been mistyped.

Illegal size for item (CV4036)

The wrong size was specified for the data item.

Illegal usage of CS register (CV4059)

The CS (code segment) register cannot be addressed in the function specified.

Index out of bound (CV1102)

A subscript value was specified that is outside the bounds declared for the array.

Insufficient EMM memory (CV3007)

CodeView tried to allocate expanded memory, but there was not enough space.

Possible solutions include the following:

- Run CVPACK on the executable file to reduce the demand on memory for symbolic information.
- Recompile without symbolic information in some of the modules. CodeView requires memory to hold information about the program being debugged. Compile some modules with the /Zd option instead of /Zi, or don't use either option.
- If multiple applications are running, remove one or more of the applications that use expanded memory. This may free enough memory to permit CodeView to run properly.
- Allocate more expanded memory in the system configuration.

Internal debugger error: *n* (CV0100)

CodeView has encountered an internal error. Quit and restart.

If the error recurs, note the circumstances of the error and notify Microsoft Corporation by following the instructions on the Microsoft Product Assistance Request form at the back of one of your manuals.

Internal error—unrecoverable fault (CV1319)

The DOS extender encountered a general protection fault.

The CodeView file may be corrupt. Reboot and copy CV.EXE from the original disks and retry.

If the error recurs, note the circumstances of the error and notify Microsoft Corporation by following the instructions on the Microsoft Product Assistance Request form at the back of one of your manuals.

Invalid address (CV0014)

The View (**V**) command was followed by an argument that could not be interpreted as a valid address.

A name or constant may have been specified without the period (.) that indicates a filename or line number.

Invalid argument (CV0022)

An invalid value was given as an argument in the most recent command.

One of the following may have occurred:

- An invalid argument was passed to the Go (**G**) command.
- An invalid argument was passed to the Delete Watch Expression (**Y**) command.

Invalid breakpoint command (CV1001)

CodeView could not interpret the breakpoint command.

The command probably used an invalid symbol or the incorrect command format.

Invalid executable file—please relink (CV1046)

The executable file did not have a valid format.

One of the following may have occurred:

- The executable file was not created with the linker released with this version of CodeView. Relink the object code using the current version of LINK.EXE.
- The .EXE file may have been corrupted. Recompile and relink the program.

Invalid flag (CV1022)

An attempt was made to examine or change a flag, but the flag name was not valid.

Any flags preceding the invalid name were changed to the values specified. Any flags after the invalid name were not changed.

Use the flag mnemonics displayed after entering the **R FL** command.

Invalid format in CV.EXE (CV1313)

The CV.EXE file has been corrupted. Copy CV.EXE from the original disks and retry.

If the error recurs, note the circumstances of the error and notify Microsoft Corporation by following the instructions on the Microsoft Product Assistance Request form at the back of one of your manuals.

Invalid format specifier (use one of ABDILSTUW) (CV1021)

A Dump (**D**), Enter (**E**), or View Memory (**VM**) command included a format specifier that is not recognized by CodeView.

The valid format specifiers are

Specifier	Display Format
A	ASCII
B	Byte
I	Integer
U	Unsigned integer
W	Word
D	Doubleword
S	Short real

Specifier	Display Format
L	Long real
T	10-byte real

Invalid format string (CV1038)

An invalid format specifier followed an expression.

Invalid operation (CV1062)

An attempt was made to set the IP register to a line or address in a different segment.

Invalid process ID (CV3603)

An attempt was made to run a process using an ID that does not exist.

The ID may have been mistyped.

Invalid radix (use 8, 10, or 16) (CV1027)

The Radix (**N**) command takes three radixes: 8 (octal), 10 (decimal), and 16 (hexadecimal). Other radixes are not permitted. The new radix is always entered as a decimal number, regardless of the current radix.

Invalid register (CV1004)

The Register (**R**) command named a register that does not exist or cannot be displayed. CodeView can display the following registers:

```
AX      SP      DS      IP
BX      BP      ES      FL
CX      SI      SS
DX      DI      CS
```

When running under DOS or Windows on an 80386/486 machine, the 386 option can be selected to display the following registers:

```
EAX     ESP     DS      GS
EBX     EBP     ES      SS
ECX     ESI     FS      EIP
EDX     EDI     CS      EFL
```

Invalid tab setting—assumed 8 (CV2210)

The value for tabs cannot be less than 0 or greater than 19. If you supply a value that is not in this range, CodeView defaults to a tab value of 8.

Invalid thread ID (CV3500)

An attempt was made to run a thread using an ID that does not exist.

The ID may have been mistyped.

Invalid type cast (CV1008)

An attempt was made to cast a variable to an undefined or user-defined type.

A cast can be made only to fundamental C types.

Library module not loaded (CV1042)

The program being debugged uses load-on-demand DLLs. At least one of these libraries is needed but does not currently exist on the path specified by the LIB-PATH environment variable.

LIM 4.0 function not supported (CV3013)

CodeView required a function that is not supported in the EMM driver present on the system.

Either of the following must be done:

- Run CodeView without using expanded memory.
- Obtain an EMM driver that fully supports LIM EMS version 4.0 or later.

LIM 4.0 subfunction not supported (CV3014)

CodeView required a subfunction that is not supported in the EMM driver present on the system.

Either of the following must be done:

- Run CodeView without using expanded memory.
- Obtain an EMM driver that fully supports LIM EMS version 4.0 or later.

Loaded symbols for module *module* (CV2207)

CodeView automatically loaded the symbols for the given DLL. The DLL can now be debugged.

Losing History (CV5010)

When you restarted the program, CodeView was not able to maintain debug history.

Match not found (CV1016)

No string was found that matched the search pattern.

Missing ')' (CV1000)

The command contained a left parenthesis (( ) that lacked a matching right parenthesis ( )).

Missing ']' (CV1014)

The command contained a left bracket ([ ) that lacked a matching right bracket ( ]).

Missing '(' (CV1034)

The command contained a right parenthesis ( )) that lacked a matching left parenthesis (( ).

Missing '(' in complex constant (CV1110)

CodeView expected an opening parenthesis of a complex constant in an expression, but it was missing.

Missing ')' in complex constant (CV1111)

CodeView expected a closing parenthesis of a complex constant, but it was missing.

Missing '(' to intrinsic (CV1113)

CodeView expected an opening parenthesis for an intrinsic function, but it was missing.

Missing ')' to intrinsic (CV1114)

CodeView expected a closing parenthesis for an intrinsic function, but it was missing.

Missing ')' in substring (CV1119)

CodeView expected a closing parenthesis for a substring expression, but it was missing.

Missing or corrupt emulator info (CV1051)

Status information about the floating-point emulator is missing or corrupt.

The program probably wrote to this area of memory. Check all pointers to confirm that they refer to their intended objects.

No closing double quotation mark (CV1029)

The double quotation mark (") expected at the end of the string was missing.

No closing single quotation mark (CV1030)

The single quotation mark (') expected at the end of the character constant was missing.

No code at this line number (CV1023)

An attempt was made to set a breakpoint at a line that does not correspond to machine code. Such a line could be a blank line, a comment line, a line with program declarations, or a line moved or removed by compiler optimization.

To set a breakpoint at a line deleted by the optimizer, recompile the program with the /Od option to turn optimization off.

Note that in a multiline statement the code is associated only with one line of the statement.

No CodeView source information (CV1059)

There is no CodeView symbol listing for the source file or module being debugged.

Be sure the file was compiled with the /Zi option or the /Zd option. If linking in a separate step, be sure to use the /CO option.

No debugging information (CV5003)

The program file did not contain the debugging information needed.

Recompile the program using the /Zi option to include CodeView symbolic information. If linking in a separate step, use the LINK /CO option.

No file selected (CV5005)

A module must be selected before OK is chosen.

To exit the dialog box without selecting a module, choose Cancel.

No free EMM memory handles (CV3005)

No expanded memory handle is available for the symbolic information.

One of the following may be a solution:

- If multiple applications are running, remove one or more of the applications that use expanded memory. This should free enough handles to permit CodeView to run properly.
- Reconfigure the EMM driver to allow more handles.

No immediate mode (CV4056)

The instruction does not take an immediate-mode operand.

No match found (CV5008)

There was no match for the specified string in the file.

No previous regular expression (CV1011)

The Repeat Last Find command was executed, but no previous regular expression (search string) has been specified.

No process status, /O not specified (CV5002)

CodeView must be started with the /O option in order to debug multiprocess programs.

Exit and restart CodeView with the /O option.

No second monitor connected to system (CV1061)

CodeView was invoked with the /2 option, but there was no second monitor for CodeView to use.

No source lines at this address (CV1031)

An attempt was made to view an address which has no source code.

No Source window open (CV1058)

A command was entered to manipulate the contents of the Source window, but no Source window is open.

No space left on device (CV0028)

No more space for writing is available on the disk.

One of the following may have occurred:

- CodeView could not find room for writing a temporary file.
- An attempt was made to write to a disk that was full.

No such file or directory (CV0002)

The specified file does not exist or a pathname does not specify an existing directory.

Check the file or directory name in the most recent command.

One of the following may have occurred:

- The View (V) command or the Open Source command from the File menu was used to view a nonexistent file.
- An attempt was made to print to a nonexistent file or directory.

No symbolic information for *filename* (CV0101)

The executable file (or DLL if in OS/2) did not contain the symbols needed by CodeView.

Be sure to compile the program or DLL using the /Zi option. If linking in a separate step, be sure to use the /CO option. Use the most current version of LINK.

No watch variables to delete (CV5009)

An attempt was made to delete one or more watch variables (watch expressions), but no watch expressions are currently selected.

Not a text file (CV1039)

An attempt was made to load a file that is not a text file, possibly a binary-data file or an executable program file. This error can also occur if the first line of a file includes characters that are not in the range of ASCII 9 to 13 or ASCII 32 to 126.

The Source window only works with text files.

Not DOS 3.0 or later (CV1315)

CodeView requires DOS version 3.0 or later. CodeView does not support DOS versions 1.*x* and 2.*x*.

Not enough memory to load CV.EXE (CV1314)

There was not enough conventional memory to load CodeView.

Possible solutions include the following:

- Free memory by removing terminate-and-stay-resident software.
- Reduce the settings in CONFIG.SYS for FILES, BUFFERS, and LASTDRIVE.

Operand expected (CV4027)

The operation or instruction requires an operand, but none was specified.

Operand must be register (CV4018)

The operand for this instruction must be a register, not a label or variable.

Operand must have size (CV4035)

No variable size was specified for the operand.

Specify the size of the variable being accessed by using the **BY**, **WO**, or **DW** operator.

Operand types incorrect for this operation (CV1010)

The operand types specified are not legal for the operation.

For example, a pointer cannot be multiplied by any value.

Operand types must match (CV4031)

The command or instruction takes two or more operands, all of the same type.

Operator must have a struct/union type (CV1033)

Components of structure variables or unions must be fully qualified. Components cannot be entered without full specification.

Operator needs lvalue (CV1032)

An expression that does not evaluate to an lvalue was specified for an operation that requires an lvalue.

An lvalue (so called because it appears on the left side of an assignment statement) is an expression that refers to a memory location.

For example, `buffer[count]` is a valid lvalue because it points to a specific memory location. The logical comparison `zed != 0` is not a valid lvalue because it evaluates to TRUE or FALSE, not a memory address.

Outdated EMM software (LIM 4.0 required) (CV3012)

The EMM driver must be LIM EMS version 4.0 or later in order to contain the calls needed for CodeView to use expanded memory.

Out of memory (CV0012)

CodeView was unable to allocate or reallocate the memory that it required because not enough memory was available.

Possible solutions include the following:

- Run CVPACK on the executable file to reduce the demand on memory for symbolic information.
- Recompile without symbolic information in some of the modules. CodeView requires memory to hold information about the program being debugged. Compile some modules with the /Zd option instead of /Zi, or don't use either option.
- Remove other programs or drivers running in the system that could be consuming significant amounts of memory.
- Decrease the settings in CONFIG.SYS for FILES and BUFFERS.

Out of memory (CV3608)

CodeView needed additional conventional memory, but insufficient memory was available.

Possible solutions include the following:

- Run CVPACK on the executable file to reduce the demand on memory for symbolic information.
- Recompile without symbolic information in some of the modules. CodeView requires memory to hold information about the program being debugged. Compile some modules with the /Zd option instead of /Zi, or don't use either option.
- Remove other programs or drivers running in the system that could be consuming significant amounts of memory.
- Free some memory by removing terminate-and-stay-resident software.
- Remove unneeded watches or breakpoints.
- Compile some modules with optimizations enabled to reduce the demand on memory made by the program being debugged.

Overlay Manager stack overflow (CV1317)

The CodeView file may be corrupt. Copy CV.EXE from the original disks and retry.

If the error recurs, note the circumstances of the error and notify Microsoft Corporation by following the instructions on the Microsoft Product Assistance Request form at the back of one of your manuals.

Overlay not resident (CV1047)

An attempt was made to disassemble machine code from an overlay section of code that is not currently resident in memory.

Execute the program until the overlay is loaded.

Packed file (CV5012)

(DOS only)

CodeView cannot debug files in DOS that are linked with the /EXEPACK option. Relink without this option to debug the file and then switch back to linking with /EXEPACK for the release version of your program.

Path of execution different from history (CV5006)

The code executed during dynamic replay differed from the recorded history.

This may be normal if the program being debugged responds to asynchronous events.

Radix must be between 2 and 36 inclusive (CV1107)

A radix outside the allowable range was specified.

Register must be AX or AL (CV4060)

The destination register for the instruction must be AX or AL.

Register variable out of scope (CV1024)

An attempt was made to display a register variable outside the scope of the function containing it.

One of the following occurred:

- The variable was declared as a register variable. Recompile the program with the register declaration removed.
- The optimizer converted an ordinary variable into a register variable to speed up execution. Recompile the program using the /Od option to turn optimization off.
- The function was defined with **_fastcall**, causing parameters to be passed in registers. Remove the **_fastcall** designation and recompile.

Regular expression too long (CV1012)

The regular expression entered was too long or complex for CodeView to handle.

Use a simpler regular expression.

Relative jump out of range (CV4053)

An address jump was specified that is greater than permitted.

A jump may be forward no more than 127 bytes and backward no more than 128 bytes relative to the next instruction.

Restart illegal in child CodeView (CV3612)

A request was made to restart the program within a child copy of CodeView.

The Restart command cannot be used on a child process in a child CodeView. It is necessary to restart the parent program to begin the child process again.

Restart program to edit options (CV2204)

The program must be restarted before the recording or playback options can be modified.

Restart program to record (CV5007)

Recording cannot begin while the program is executing.

Restart the program before recording.

Resynchronizing the user tape (CV2205)

The command history and user input tapes are out of synchronization. CodeView automatically adjusted the user tape to be synchronized with the command tape.

Screen session ended—application output lost (CV1044)

Under OS/2, each screen display is handled by a different session. When CodeView tried to switch from one display to the other, the other display's session had ended and the output was gone.

Exit CodeView and restart it.

Simple variable can not have arguments (CV1115)

In an expression, an argument was specified to a simple variable.

For example, given the declaration `INTEGER NUM`, the expression `NUM(I)` is not allowed.

Specified number of lines not supported, using default (CV1052)

A display mode was selected that is not supported by either the monitor's hardware or the driver routines.

Exit CodeView, then restart it with an appropriate command-line option for the display mode, either /25, /43, or /50.

Substring range out of bound (CV1118)

A character expression exceeded the length specified in the **CHARACTER** statement.

Symbol not defined (CV4009)

The symbol specified has not been previously defined.

The symbol name may have been mistyped.

Syntax error (CV1017)

The command contained a syntax error.

The most likely cause is an invalid command or expression.

The program has terminated, restart to continue (CV0003)

CodeView has detected a termination request by the program being debugged.

The program cannot be executed because it has terminated and has not been restarted. Program memory remains allocated and may still be examined at this point.

To run the program again, reload it using the Restart command.

Thread blocked (CV3501)

The requested thread will not run because it is blocked by another thread.

If this is not expected behavior for the program being debugged, it may be necessary to terminate the threads that are blocking the requested thread.

Too few array bounds given (CV1103)

The bounds specified in an array subscript do not match the array declaration.

For example, given the array declaration `INTEGER IARRAY(3,4)`, the expression `IARRAY(I)` would produce this message.

Too many array bounds given (CV1104)

Too many subscripts were specified for the array.

For example, given the array declaration `INTEGER IARRAY(3,4)`, the expression `IARRAY(I,3,J)` would produce this error message.

Too many open files (CV0024)

CodeView could not open a file it needed because no more file handles are available.

In DOS, increase the number of file handles by changing the FILES setting in CONFIG.SYS to allow a larger number of open files. FILES=20 is the recommended setting. In OS/2, if multiple processes are running, removing one or more of them may release enough file handles.

The program being debugged may have so many files open that all available handles are exhausted. Check that the program has not left files open unnecessarily. The first four handles are reserved by the operating system.

Too many watch objects (CV1036)

More watch objects were specified than CodeView can handle.

The number of watch expressions that can be specified varies with the demands made upon CodeView's internal memory resources.

Remove one or more of the watch expressions, or remove some breakpoints.

TOOLS.INI not found (CV1053)

The directory listed in the INIT environment variable did not contain a TOOLS.INI file.

Check the INIT variable to be sure it points to the correct directory.

Type clash in function argument (CV1117)

The type of an actual parameter did not match the corresponding formal parameter.

This message also appears when a routine that uses alternate returns is called and the values of the return labels in the actual parameter list are not 0.

Type conversion too complex (CV1037)

Too many levels of type casting were specified.

Type casting is limited to two levels, as in

```
(char) ((int) (floatvar))
```

Unable to create tape (CV2200)

CodeView could not open a disk file (tape) to record commands and data for later replay.

Choosing the History On option from the Run menu causes CodeView to open disk files *program*.CVH and *program*.CVI to record all commands and data for a debugging session.

One of the following situations may have caused the error:

- There was not enough space on the disk containing the program to be debugged.
- There were not enough free file handles. In DOS, increase the number of file handles by changing the FILES setting in CONFIG.SYS to allow a larger number of open files. FILES=20 is the recommended setting. In OS/2, if multiple processes are running, removing one or more of them may release enough file handles to permit creating the tape.

Unable to open file (CV1007)

The file specified cannot be opened.

One of the following may have occurred:

- The file may not exist in the specified directory.
- The filename was misspelled.

- The file's attributes are set so that it cannot be opened.
- A locking or sharing violation occurred.

Unable to open tape (CV2201)

CodeView could not open the history file (tape) for replay.

Choosing the History On option from the Run menu causes CodeView to open disk files *program*.CVH and *program*.CVI to record all commands and data for a debugging session.

There probably were not enough free file handles. In DOS, increase the number of file handles by changing the FILES setting in CONFIG.SYS to allow a larger number of open files. FILES=20 is the recommended setting. In OS/2, if multiple processes are running, removing one or more of them may release enough file handles to permit opening the tape.

Unexpected EMM error (CV1316)

An unexpected error occurred when reading overlays into expanded memory.

One of the following has probably occurred:

- The EMM driver may be corrupted or may have malfunctioned. Reload the driver and retry.
- Expanded memory has been corrupted.

If the error recurs, note the circumstances of the error and notify Microsoft Corporation by following the instructions on the Microsoft Product Assistance Request form at the back of one of your manuals.

Unexpected end-of-file in CV.EXE (CV1312)

An unexpected end-of-file occurred while CV.EXE was being read.

The CodeView file may be corrupt. Copy CV.EXE from the original disks and retry.

If the error recurs, note the circumstances of the error and notify Microsoft Corporation by following the instructions on the Microsoft Product Assistance Request form at the back of one of your manuals.

Unknown queue request—ignored (CV3609)

One of the CodeView processes sent a command or data to another CodeView process that the latter process did not recognize. This is not a fatal error.

If this is a recurring error, please note the circumstances of the error and notify Microsoft Corporation by following the instructions on the Microsoft Product Assistance Request form at the back of one of your manuals.

Unknown symbol (CV1018)

The symbolic name specified could not be found.

One of the following may have occurred:

- The specified name was misspelled.
- The wrong case was used when case sensitivity was on. Case sensitivity is toggled by the Case Sensitivity command from the Options menu, or set by the Option (**O**) Command-window command.
- The module containing the specified symbol may not have been compiled with the /Zi option to include symbolic information.

User Tape Disabled (CV2208)

The current CodeView session was invoked with the /K option to disable the keyboard. CodeView issues this warning as a reminder that the recording ability is limited when /K is used. You can record CodeView commands made during a debugging session, but not user keystrokes.

User tape may be truncated (CV2203)

A request was made to start recording again without completely rerunning the original history tape. Any unexecuted commands will be discarded.

Value out of range (CV4050)

The value specified was out of range for the data item.

Video mode changed without /S option (CV1040)

The program being debugged changed screen modes, and CodeView was not set for swapping. The program output is now damaged or unrecoverable.

To be able to view program output, exit CodeView and restart it with the Swap (/S) option.

Wrong number of function arguments (CV1116)

An incorrect number of arguments was specified in a function call.

Wrong type of register (CV4019)

The register specified is not permitted for this operation or instruction.

The mnemonic for the register may have been mistyped.

/X: CPU in protected or virtual mode (CV1301)

The DOS extender was unable to switch to protected mode.

One of the following may have occurred:

- OS/2 is running.
- An EMM driver is running in protected mode.
- A protected-mode application is running.

/X: CPU not 80286 or later (CV1300)

The DOS extender runs in protected mode, which is supported only on the 80286 and later processors.

/X: HIMEM.SYS not loaded (CV1302)

HIMEM.SYS is used by the DOS extender to allocate extended memory and must be installed.

/X: not enough extended memory (CV1303)

There was not enough space in extended memory to load the DOS extender.

One of the following may be a solution:

- Remove programs that are using extended memory.
- Run CodeView without the /X option.

/X: Unexpected initialization error (CV1320)

The DOS extender encountered a general protection fault.

The CodeView file may be corrupt. Copy CV.EXE from the original disks and retry.

If the error recurs, note the circumstances of the error and notify Microsoft Corporation by following the instructions on the Microsoft Product Assistance Request form at the back of one of your manuals.

F.3 EXEHDR Error Messages

This section lists error messages generated by the Microsoft EXE File Header Utility (EXEHDR). EXEHDR errors (U11*xx*) are always fatal.

Number	EXEHDR Error Message

U1100 **invalid magic number** *number*

EXEHDR discovered an unknown signature in the header for the file.

The signature in the header for a file gives the operating system under which the executable file will run. EXEHDR recognizes signatures for DOS and OS/2 only.

U1101 **automatic data segment greater than 64K; correcting heap size**

There was not enough space in the automatic, or default, data segment (DGROUP) to accommodate the requested new heap size. EXEHDR adjusted the heap size to the maximum available space.

This error applies only to OS/2 programs.

U1102 **automatic data segment greater than 64K; correcting stack size**

There was not enough space in the automatic, or default, data segment (DGROUP) to accommodate the requested new stack size. EXEHDR adjusted the stack size to the maximum available space.

This error applies only to OS/2 programs.

U1103 **invalid .EXE file : actual length less than reported**

The second and third fields in the input-file header indicate a file size greater than the actual size.

U1104 **cannot change load-high program**

When the minimum allocation value and the maximum allocation value are both 0, the file cannot be modified.

U1105 **minimum allocation less than stack; correcting minimum**

If the minimum allocation is not enough to accommodate the stack (either the original stack request or the modified request), the minimum allocation value is adjusted.

This error applies only to DOS programs.

U1106 **minimum allocation greater than maximum; correcting maximum**

If the minimum allocation is greater than the maximum allocation, the maximum allocation value is adjusted.

If a display of DOS header values is requested, the values shown will be the values after the packed file is expanded.

This error applies only to DOS programs.

U1107 **unexpected end of resident/nonresident name table**

While decoding run-time relocation records, EXEHDR found the end of either the resident names table or the nonresident names table. The executable file is probably corrupted.

This error applies only to OS/2 and Windows programs.

U1108 **cannot display compressed relocation records**

EXEHDR cannot decode the information in the file header because the header is not in a standard format.

U1109 **illegal value** *argument*

The given argument was invalid for the EXEHDR option it was specified with.

U1110 **malformed number** *number*

A command-line option for EXEHDR required a value, but the specified number was mistyped.

U1111 **option requires value**

A command-line option for EXEHDR required a value, but no value was specified, or the specified value was in an illegal format for the given option.

U1112 **value out of legal range** *lower–upper*

A command-line option for EXEHDR required a value, but the specified number did not fall in the required decimal range.

U1113 **value out of legal range** *lower–upper*

A command-line option for EXEHDR required a value, but the specified number did not fall in the required hexadecimal range.

U1114 **missing option value; option** *option* **ignored**

A command-line option for EXEHDR required a value, but nothing was specified. EXEHDR ignored the option.

U1115 **option** *option* **ignored**

A command-line option for EXEHDR was ignored. This error usually occurs with error U1116, `unrecognized option`.

U1116 **unrecognized option:** *option*

A command-line option for EXEHDR was not recognized. This error usually occurs with either U1115, `option ignored`, or U1111, `option requires value`.

U1120 **input file missing**

No input file was specified on the EXEHDR command line.

U1121 **command line too long:** *commandline*

The given EXEHDR command line exceeded the limit of 512 characters.

U1130 **cannot read** *filename*

EXEHDR could not read the input file. Either the file is missing or the file attribute is set to prevent reading.

U1131 **invalid .EXE file**

The input file specified on the EXEHDR command line was not a valid executable file.

U1132 **unexpected end-of-file**

EXEHDR found an unexpected end-of-file condition while reading the executable file. The file is probably corrupt.

U1140 **out of memory**

There was not enough memory for EXEHDR to decode the header of the executable file.

F.4 HELPMAKE Error Messages

This section lists error messages generated by the Microsoft Help File Maintenance Utility (HELPMAKE):

- Fatal errors (H1*xxx*) cause HELPMAKE to stop execution. No output file is produced.
- Errors (H2*xxx*) do not prevent an output file from being produced, but parts of the conversion are not completed.

- Warnings (H4*xxx*) do not prevent an output file from being produced, but problems may exist in the output.

F.4.1 HELPMAKE Fatal Errors

Number	HELPMAKE Error Message
H1000	**/A requires character**

The /A option requires an application-specific control character.

The correct form is

/A*c*

where *c* is the control character.

H1001 **/E compression level must be numeric**

The /E option requires either no argument or a numeric value in the range 0–15. The correct form is

/E*n*

where *n* specifies the amount of compression requested.

H1002 **multiple /O parameters specified**

Only one output file can be specified with the /O option.

H1003 **invalid /S file-type identifier**

The /S option was given an argument other than 1, 2, or 3.

The /S option requires specification of the type of input file. An invalid file-type identifier was specified. The correct form is

/S*n*

where *n* specifies the format of the input help text file. The only valid values are 1 (RTF), 2 (QuickHelp format), and 3 (minimally formatted ASCII).

H1004 **/S requires file-type identifier**

The /S option requires specification of the type of input file. There was no file-type identifier specified.

The correct form is

/S*n*

where *n* specifies the format of the input help text file. The only valid values are 1 (RTF), 2 (QuickHelp format), and 3 (minimally formatted ASCII).

H1005 **/W fixed width invalid**

An invalid width was specified with the /W option. The valid range is 11–255.

H1006 **multiple /K parameters specified**

The option for specifying a keyword separator file, /K, was used more than once on the HELPMAKE command line.

Only one file containing separator characters may be specified.

H1050 **option invalid with /DS**

The /C, /L, and /O options for encoding are invalid with the /DS option for decoding.

H1051 **improper arguments for /D**

The /D option permits either no argument or an S or U argument. In addition, /D is invalid with the /C or /L option.

H1052 **encode requires /O option**

Database encoding was requested without a specified output-file name for the operation.

H1053 **compression level exceeds 15**

A value greater than 15 was specified with the /E option.

The /E option requires either no argument or a numeric value in the range 0–15. The correct form is

/E*n*

where *n* specifies the amount of compression requested.

H1097 **no operation specified**

The HELPMAKE command line did not contain an option for encoding, decoding, or help.

HELPMAKE requires the /E, /D, /H, or /? option.

H1098 **unrecognized option**

An unrecognized name followed the option indicator.

An option is specified by a forward slash (/) or a dash (–) and an option name.

H1099 **syntax error on command line**

HELPMAKE cannot interpret the command line.

H1100 **cannot open file**

One of the files specified on the HELPMAKE command line could not be found or created.

H1101 **error writing file**

The output file could not be written, probably because the disk is full.

H1102 **no input file specified**

In an encoding operation, no input help text file was specified.

H1103 **no context strings found**

No context strings were found in the input stream while encoding.

Either the file is empty, or the specified /S value does not correspond to the help text formatting.

H1104 **no topic text found**

No topic text was found in the help text file.

Either the file is empty, or the specified /S value does not correspond to the help text formatting.

H1107 **cannot overwrite input file**

The /DS option for splitting a concatenated help file was specified, but the help file contained a database with the same name as the help file.

Rename the help file to a filename other than one of the database names.

H1200 **insufficient memory to allocate context buffer**

There was insufficient memory to run HELPMAKE.

HELPMAKE requires 256K free memory.

H1201 **insufficient memory to allocate utility buffer**

There was insufficient memory to run HELPMAKE.

HELPMAKE requires 256K free memory.

H1250 **not a valid compressed help file**

The input file specified for a decompression operation is not a valid help database file.

H1251 **cannot decompress locked help file**

An attempt was made to decompress a help database file that is locked.

A file is locked if the /L option is specified when the help file is created.

H1300 **word too long in RTF processing**

A single word was longer than the specified format width (set by the /W option) or was found to be longer than 128 characters when HELPMAKE was reformatting a paragraph.

H1302 **attribute stack overflow processing RTF**

RTF attribute groups are nested too deeply. HELPMAKE supports a maximum of 50 levels of attribute-group nesting in RTF format.

H1303 **unknown RTF attribute**

An unknown RTF formatting command was found.

One of the following may have occurred:

- A new RTF attribute was used. HELPMAKE recognizes a set of attributes that were current at the time this version of HELPMAKE was created. It interprets some of the attributes and knows to ignore the others. Any RTF attribute defined after HELPMAKE was created is not known by HELPMAKE and will cause this error.
- The RTF file is corrupted.

H1304 **topic too large**

A topic exceeded the limit for the size of topics.

A single topic cannot exceed 64K.

H1305 **topic text without context string**

The source file contained topic text that was not preceded by a .context definition.

H1900 **internal virtual memory error**

This message indicates an internal HELPMAKE error.

Note the circumstances of the error and notify Microsoft Corporation by following the instructions on the Microsoft Product Assistance Request form at the back of one of your manuals.

H1901 **out of local memory**

This message indicates an internal HELPMAKE error.

Note the circumstances of the error and notify Microsoft Corporation by following the instructions on the Microsoft Product Assistance Request form at the back of one of your manuals.

H1902 **out of disk space for swap file**

The current drive or directory is full.

HELPMAKE uses a temporary swap file, written to the current drive and directory. The temporary file can grow to 1.5 times the size of the input files (for large help files) and is not removed until the final help file is completed.

H1903 **cannot open swap file**

HELPMAKE was unable to create its temporary swap file on the current drive and directory for one of the following reasons:

- The current drive or directory is full.
- The device cannot be written to.

H1990 **internal compression error**

This message indicates an internal HELPMAKE error.

Note the circumstances of the error and notify Microsoft Corporation by following the instructions in the Microsoft Product Assistance Request form at the back of one of your manuals.

F.4.2 HELPMAKE Errors

Number	HELPMAKE Error Message

H2000 **line too long, truncated**

A line exceeded the fixed width specified by the /W option or the default of 76 characters. HELPMAKE truncated the extra characters.

H2001 **duplicate context string**

A context string preceded more than one topic in a help database. A context string can be associated with only one block of topic text.

H2002 **zero length hot spot**

A cross-reference was specified, but the word or anchored text associated with it was of zero length.

With no visible text to associate with the cross-reference, the hot spot will be inoperative. This error is issued as a warning and does not prevent the building of a help file. However, some applications may not be able to use the resulting help file correctly.

The following example will cause this error:

```
\a\vcross_reference\v
```

H2003 **unrecognized dot command**

A line in the source file contained a dot (.) in column 1, but it was not followed by a command recognized by HELPMAKE.

F.4.3 HELPMAKE Warnings

Number	HELPMAKE Warning

H4000 **keyword compression analysis table size exceeded no further new words will be analyzed**

The maximum number (16,000) of unique keywords has been encountered during keyword compression. This happens only in very large help files. No further keywords will be included in the analysis. HELPMAKE continues to analyze how frequently words occur that it has already encountered.

H4002 **reference to undefined local context**

A string specifying a local context was used in a cross-reference but was not defined in a .context statement.

A local context begins with an at sign (@). Each local context that is used must be defined in a .context statement in one of the input files to HELPMAKE.

H4003 **negative left indent**

Topic text in an RTF file was formatted with a left indent to a position to the left of column 1. HELPMAKE deleted all text preceding column 1.

F.5 H2INC Error Messages

This section lists error messages generated by the C to MASM Include File Translator (H2INC). The error messages produced by the compiler fall into three categories:

- Fatal error messages
- Compilation error messages
- Warning messages

The messages for each category are listed below in numerical order, with a brief explanation of each error. To look up an error message, first determine the message category, then find the error number. All messages give the filename and line number where the error occurs.

Fatal Error Messages

Fatal error messages indicate a severe problem, one that prevents the compiler from processing your program any further. These messages have the following format:

filename (line) : `fatal error HI1xxx:` *messagetext*

After the compiler displays a fatal-error message, it terminates without producing an include file or checking for further errors.

Compilation Error Messages

Compilation error messages identify actual header errors. There messages appear in the following format:

filename (*line*) : `error HI2xxx:` *messagetext*

The compiler does not produce an include file for a header file that has compilation errors. When the compiler encounters such errors, it attempts to recover from the error. If possible, it continues to process the header file and produce error messages. If errors are too numerous or too severe, the compiler stops processing.

Warning Messages

Warning messages are informational only; they do not prevent compilation. These messages appear in the following format:

filename (*line*) : `warning HI4xxx:` *messagetext*

F.5.1 H2INC Fatal Errors

Number	Message
HI1003	**error count exceeds n; stopping compilation**
	Errors in the program were too numerous or too severe to allow recovery, and the compiler must terminate.
HI1004	**unexpected end-of-file found**
	The default disk drive did not contain sufficient space for the compiler to create temporary files. The space required is approximately two times the size of the source file.
	This message also appears when the **#if** directive occurs without a corresponding closing **#endif** directive while the **#if** test directs the compiler to skip the section.
HI1007	**unrecognized flag** *string* **in** *option*
	The *string* in the command-line *option* was not a valid option.
HI1008	**no input file specified**
	The compiler was not given a file to compile.
HI1009	**compiler limit : macros nested too deeply**
	Too many macros were being expanded at the same time.
	This error occurs when a macro definition contains macros to be expanded and those macros contain other macros.
	Try to split the nested macros into simpler macros.
HI1011	**compiler limit :** *identifier* **: macro definition too big**
	The macro definition was longer than allowed.
	Split the definition into shorter definitions.
HI1012	**unmatched parenthesis nesting - missing** *character*
	The parentheses in a preprocessor directive were not matched. The missing character is either a left, (, or right,), parenthesis.
HI1016	**#if[n]def expected an identifier**
	An identifier must be specified with the **#ifdef** and **#ifndef** directives.
HI1017	**invalid integer constant expression**
	The expression in an **#if** directive either did not exist or did not evaluate to a constant.

HI1018 **unexpected '#elif'**

The **#elif** directive is legal only when it appears within an **#if**, **#ifdef**, or **#ifndef** construct.

HI1019 **unexpected '#else'**

The **#else** directive is legal only when it appears within an **#if**, **#ifdef**, or **#ifndef** construct.

HI1020 **unexpected '#endif'**

An **#endif** directive appeared without a matching **#if**, **#ifdef**, or **#ifndef** directive.

HI1021 **invalid preprocessor command** *string*

The characters following the number sign (#) did not form a valid preprocessor directive.

HI1022 **expected '#endif'**

An **#if**, **#ifdef**, or **#ifndef** directive was not terminated with an **#endif** directive.

HI1023 **cannot open source file** *filename*

The given file either did not exist, could not be opened, or was not found.

Make sure the environment settings are valid and that the correct path name for the file is specified.

If this error appears without an error message, the compiler has run out of file handles. If in DOS, increase the number of file handles by changing the FILES setting CONFIG.SYS to allow a larger number of open files. FILES=20 is the recommended setting.

HI1024 **cannot open include file** *filename*

The specified file in an **#include** preprocessor directive could not be found.

Make sure settings for the INCLUDE and TMP environment variables are valid and that the correct path name for the file is specified.

If this error appears without an error message, the compiler has run out of file handles. If in DOS, increase the number of file handles by changing the FILES setting in CONFIG.SYS to allow a larger number of open files. FILES=20 is the recommended setting.

HI1026 **parser stack overflow, please simplify your program**

The program cannot be processed because the space required to parse the program causes a stack overflow in the compiler.

Simplify the program by decreasing the complexity of expressions. Decrease the level of nesting in for and switch statements by putting some of the more deeply nested statements in separate functions. Break up very long expressions involving ',' operators or function calls.

HI1033 **cannot open assembly language output file** *filename*

There are several possible causes for this error:

- The given name is not valid.
- The file cannot be opened for lack of space.
- A read-only file with the given name already exists.

HI1036 **cannot open source listing file** *filename*

There are several possible causes for this error:

- The given name is not valid.
- The file cannot be opened for lack of space.
- A read-only file with the given name already exists.

HI1039 **unrecoverable heap overflow in Pass 3**

The post-optimizer compiler pass overflowed the heap and could not continue.

One of the following may be a solution:

- Break up the function containing the line that caused the error.
- Recompile with the /Od option, removing optimization.
- In OS/2, recompile using the /B3 C3L option to invoke the large-model version of the third pass of the compiler.
- In DOS, remove other programs or drivers running in the system which could be consuming significant amounts of memory.
- In DOS, if using NMAKE, compile without using NMAKE.

HI1040 **unexpected end-of-file in source file** *filename*

The compiler detected an unexpected end-of-file condition while creating a source listing or mingled source/object listing.

This occurs under OS/2 if the source file is deleted or overwritten while it is being read.

HI1047 **limit of** *option* **exceeded at** *string*

The given option was specified too many times. The given string is the argument to the option that caused the error.

If the CL or H2INC environment variables have been set, options in these variables are read before options specified on the command line. The CL environment variable is read before the H2INC environment variable.

HI1048 **unknown option** *character* **in** *option*

The given character was not a valid letter for the option.

For example, the following line

```
#pragma optimize("q", on)
```

causes the following error

```
unknown option 'q' in '#pragma optimize'
```

HI1049 **invalid numerical argument** *string*

A numerical argument was expected instead of the given string.

HI1050 *segment* **: code segment too large**

A code segment grew to within 36 bytes of 64K during compilation.

A 36-byte pad is used because of a bug in some 80286 chips that can cause programs to exhibit strange behavior when, among other conditions, the size of a code segment is within 36 bytes of 64K.

HI1052 **compiler limit : #if/#ifdef nested too deeply**

The program exceeded the maximum of 32 nesting levels for **#if** and **#ifdef** directives.

HI1053 **compiler limit : struct/union nested too deeply**

A structure or union definition was nested to more than 15 levels.

Break the structure or union into two parts by defining one or more of the nested structures using **typedef**.

HI1090 *segment* **data allocation exceeds 64K**

The size of the named segment exceeds 64K.

This error occurs with **_based** allocation.

HI1800 *option* **: unrecognized option**

A command-line option was specified that was not understood by H2INC.

F.5.2 H2INC Compilation Errors

Number	Message

HI2000 **UNKNOWN ERROR**
Contact Microsoft Product Support Services

The compiler detected an unknown error condition.

Note the circumstances of the error and notify Microsoft Corporation by following the instructions on the Microsoft Product Assistance Request form at the back of one of your manuals.

HI2001 **newline in constant**

A string constant was continued onto a second line without either a backslash or closing and opening quotes.

To break a string constant onto two lines in the source file, do one of the following:

- End the first line with the line-continuation character, a backslash, \.
- Close the string on the first line with a double quotation mark, and open the string on the next line with another quotation mark.

It is not sufficient to end the first line with \n, the escape sequence for embedding a newline character in a string constant.

The following two examples demonstrate causes of this error:

```
printf("Hello,
world");
```

or

```
printf("Hello,\n
world");
```

The following two examples show ways to correct this error:

```
printf("Hello,\
world");
```

or

```
printf("Hello,"
" world");
```

Note that any spaces at the beginning of the next line after a line-continuation character are included in the string constant. Note, also, that neither solution actually places a newline character into the string constant. To embed this character:

```
printf("Hello,\n\
world");
```

or

```
printf("Hello,\
\nworld");
```

or

```
printf("Hello,\n"
"world");
```

or

```
printf("Hello,"
"\nworld");
```

HI2003 **expected** *defined id*

An identifier was expected after the preprocessing keyword *defined.*

HI2004 **expected** *defined(id)*

An identifier was expected after the left parenthesis, (, following the preprocessing keyword *defined.*

HI2005 **#line expected a line number, found** *token*

A **#line** directive lacked the required line-number specification.

HI2006 **#include expected a file name, found** *token*

An **#include** directive lacked the required filename specification.

HI2007 **#define syntax**

An identifier was expected following **#define** in a preprocessing directive.

HI2008 *character* **: unexpected in macro definition**

The given character was found immediately following the name of the macro.

HI2009 **reuse of macro formal** *identifier*

The given identifier was used more than once in the formal-parameter list of a macro definition.

HI2010 *character* **: unexpected in macro formal-parameter list**

The given character was used incorrectly in the formal-parameter list of a macro definition.

HI2012 **missing name following '<'**

An **#include** directive lacked the required filename specification.

HI2013 **missing '>'**

The closing angle bracket (>) was missing from an **#include** directive.

HI2014 **preprocessor command must start as first non-white-space**

Non-white-space characters appeared before the number sign (#) of a preprocessor directive on the same line.

HI2015 **too many characters in constant**

A character constant contained more than one character.

Note that an escape sequence (for example, \t for tab) is converted to a single character.

HI2016 **no closing single quotation mark**

A newline character was found before the closing single quotation mark of a character constant.

HI2017 **illegal escape sequence**

An escape sequence appeared where one was not expected.

An escape sequence (a backslash, \ , followed by a number or letter) may occur only in a character or string constant.

HI2018 **unknown character** *hexnumber*

The ASCII character corresponding to the given hexadecimal number appeared in the source file but is an illegal character.

One possible cause of this error is corruption of the source file. Edit the file and look at the line on which the error occurred.

HI2019 **expected preprocessor directive, found** *character*

The given character followed a number sign (#), but it was not the first letter of a preprocessor directive.

HI2021 **expected exponent value, not** *character*

The given character was used as the exponent of a floating-point constant but was not a valid number.

HI2022 *number* **: too big for character**

The octal number following a backslash (\) in a character or string constant was too large to be represented as a character.

HI2025 *identifier* **: enum/struct/union type redefinition**

The given identifier had already been used for an enumeration, structure, or union tag.

HI2026 *identifier* **: member of enum redefinition**

The given identifier has already been used for an enumeration constant, either within the same enumeration type or within another visible enumeration type.

HI2027 **use of undefined enum/struct/union** *identifier*

The given identifier referred to a structure or union type that was not defined.

HI2028 **struct/union member needs to be inside a struct/union**

Structure and union members must be declared within the structure or union.

This error may be caused by an enumeration declaration containing a declaration of a structure member, as in the following example:

```
enum a {
january,
february,
int march;    /* Illegal structure declaration */
};
```

HI2030 *identifier* **: struct/union member redefinition**

The identifier was used for more than one member of the same structure or union.

HI2031 *identifier* **: function cannot be struct/union member**

The given function was declared to be a member of a structure or union.

To correct this error, use a pointer to the function instead.

HI2033 *identifier* **: bit field cannot have indirection**

The given bit field was declared as a pointer (*), which is not allowed.

HI2034 *identifier* **: type of bit field too small for number of bits**

The number of bits specified in the bit-field declaration exceeded the number of bits in the given base type.

HI2035 **struct/union** *identifier* **: unknown size**

The given structure or union had an undefined size.

Usually this occurs when referencing a declared but not defined structure or union tag.

For example, the following causes this error:

```
struct s_tag *ps;
ps = &my_var;
*ps = 17;    /* This line causes the error */
```

HI2037 **left of** *operator* **specifies undefined struct/union** *identifier*

The expression before the member-selection operator (**->** or **.**) identified a structure or union type that was not defined.

HI2038 *identifier* **: not struct/union member**

The given identifier was used in a context that required a structure or union member.

HI2041 **illegal digit** *character* **for base** *number*

The given character was not a legal digit for the base used.

HI2042 **signed/unsigned keywords mutually exclusive**

The keywords *signed* and *unsigned* were both used in a single declaration, as in the following example:

```
unsigned signed int i;
```

HI2056 **illegal expression**

An expression was illegal because of a previous error, which may not have produced an error message.

HI2057 **expected constant expression**

The context requires a constant expression.

HI2058 **constant expression is not integral**

The context requires an integral constant expression.

HI2059 **syntax error :** *token*

The token caused a syntax error.

HI2060 **syntax error : end-of-file found**

The compiler expected at least one more token.

Some causes of this error include:

Omitting a semicolon (;), as in

```
int *p
```

Omitting a closing brace (}) from the last function, as in

```
main()
{
```

HI2061 **syntax error : identifier** *identifier*

The identifier caused a syntax error.

HI2062 **type** *type* **unexpected**

The compiler did not expect the given type to appear here, possibly because it already had a required type.

HI2063 *identifier* **: not a function**

The given identifier was not declared as a function, but an attempt was made to use it as a function.

HI2064 **term does not evaluate to a function**

An attempt was made to call a function through an expression that did not evaluate to a function pointer.

HI2065 *identifier* **: undefined**

An attempt was made to use an identifier that was not defined.

HI2066 **cast to function type is illegal**

An object was cast to a function type, which is illegal.

However, it is legal to cast an object to a function pointer.

HI2067 **cast to array type is illegal**

An object was cast to an array type.

HI2068 **illegal cast**

A type used in a cast operation was not legal for this expression.

HI2069 **cast of void term to nonvoid**

The void type was cast to a different type.

HI2070 **illegal sizeof operand**

The operand of a **sizeof** expression was not an identifier or a type name.

HI2071 *identifier* **: illegal storage class**

The given storage class cannot be used in this context.

HI2072 *identifier* **: initialization of a function**

An attempt was made to initialize a function.

HI2043 **illegal break**

A break statement is legal only within a **do**, **for**, **while**, or **switch** statement.

HI2044 **illegal continue**

A continue statement is legal only within a **do**, **for**, or **while** statement.

HI2045 *identifier* **: label redefined**

The label appeared before more than one statement in the same function.

HI2046 **illegal case**

The keyword case may appear only within a **switch** statement.

HI2047 **illegal default**

The keyword default may appear only within a **switch** statement.

HI2048 **more than one default**

A **switch** statement contained more than one default label.

HI2049 **case value** *value* **already used**

The case value was already used in this **switch** statement.

HI2050 **nonintegral switch expression**

A **switch** expression did not evaluate to an integral value.

HI2051 **case expression not constant**

Case expressions must be integral constants.

HI2052 **case expression not integral**

Case expressions must be integral constants.

HI2054 **expected '(' to follow** *identifier*

The context requires parentheses after the function identifier.

One cause of this error is forgetting an equal sign (=) on a complex initialization, as in

```
int array1[]    /* Missing = */
{
1,2,3
};
```

HI2055 **expected formal-parameter list, not a type list**

An argument-type list appeared in a function definition instead of a formal-parameter list.

HI2075 *identifier* **: array initialization needs curly braces**

There were no curly braces, { }, around the given array initializer.

HI2076 *identifier* **: struct/union initialization needs curly braces**

There were no curly braces, { }, around the given structure or union initializer.

HI2077 **nonscalar field initializer** *identifier*

An attempt was made to initialize a bit-field member of a structure with a nonscalar value.

HI2078 **too many initializers**

The number of initializers exceeded the number of objects to be initialized.

HI2079 *identifier* **uses undefined struct/union** *name*

The *identifier* was declared as structure or union type name, but the *name* had not been defined. This error may also occur if an attempt is made to initialize an anonymous union.

HI2080 **illegal far _fastcall function**

A far **_fastcall** function may not be compiled with the /Gw option, nor with the /Gq option if stack checking is enabled.

HI2082 **redefinition of formal parameter** *identifier*

A formal parameter to a function was redeclared within the function body.

HI2084 **function** *function* **already has a body**

The function has already been defined.

HI2086 *identifier* **: redefinition**

The given identifier was defined more than once, or a subsequent declaration differed from a previous one.

The following are ways to cause this error:

```
int a;
char a;
main()
{
}
main()
{
int a;
int a;
}
```

However, the following does not cause this error:

```
int a;
int a;
main()
{
}
```

HI2087 *identifier* **: missing subscript**

The definition of an array with multiple subscripts was missing a subscript value for a dimension other than the first dimension.

The following is an example of an illegal definition:

```
int func(a)
char a[10][];
{ }
```

The following is an example of a legal definition:

```
int func(a)
char a[][5];
{ }
```

HI2090 **function returns array**

A function cannot return an array. It can return a pointer to an array.

HI2091 **function returns function**

A function cannot return a function. It can return a pointer to a function.

HI2092 **array element type cannot be function**

Arrays of functions are not allowed. Arrays of pointers to functions are allowed.

HI2095 *function* **: actual has type void : parameter** *number*

An attempt was made to pass a void argument to a function. The given number indicates which argument was in error.

Formal parameters and arguments to functions cannot have type void. They can, however, have type void * (pointer to void).

HI2100 **illegal indirection**

The indirection operator (*) was applied to a nonpointer value.

HI2101 **'&' on constant**

The address-of operator (**&**) did not have an **lvalue** as its operand.

HI2102 **'&' requires lvalue**

The address-of operator (**&**) must be applied to an **lvalue** expression.

HI2103 **'&' on register variable**

An attempt was made to take the address of a register variable.

HI2104 **'&' on bit field ignored**

An attempt was made to take the address of a bit field.

HI2105 *operator* **needs lvalue**

The given operator did not have an **lvalue** operand.

HI2106 *operator* **: left operand must be lvalue**

The left operand of the given operator was not an **lvalue**.

HI2107 **illegal index, indirection not allowed**

A subscript was applied to an expression that did not evaluate to a pointer.

HI2108 **nonintegral index**

A nonintegral expression was used in an array subscript.

HI2109 **subscript on nonarray**

A subscript was used on a variable that was not an array.

HI2110 **pointer + pointer**

An attempt was made to add one pointer to another using the plus (+) operator.

HI2111 **pointer + nonintegral value**

An attempt was made to add a nonintegral value to a pointer.

HI2112 **illegal pointer subtraction**

An attempt was made to subtract pointers that did not point to the same type.

HI2113 **pointer subtracted from nonpointer**

The right operand in a subtraction operation using the minus (-) operator was a pointer, but the left operand was not.

HI2114 *operator* **: pointer on left; needs integral right**

The left operand of the given operator was a pointer, so the right operand must be an integral value.

HI2115 *identifier* **: incompatible types**

An expression contained incompatible types.

HI2117 *operator* **: illegal for struct/union**

Structure and union type values are not allowed with the given operator.

HI2118 **negative subscript**

A value defining an array size was negative.

HI2120 **void illegal with all types**

The void type was used in a declaration with another type.

HI2121 *operator* **: bad left/right operand**

The left or right operand of the given operator was illegal for that operator.

HI2124 **divide or mod by zero**

A constant expression was evaluated and found to have a zero denominator.

HI2128 *identifier* **: huge array cannot be aligned to segment boundary**

The given huge array was large enough to cross two segment boundaries, but could not be aligned to both boundaries to prevent an individual array element from crossing a boundary.

If the size of a huge array causes it to cross two boundaries, the size of each array element must be a power of two, so that a whole number of elements will fit between two segment boundaries.

HI2129 **static function** *function* **not found**

A forward reference was made to a static function that was never defined.

HI2130 **#line expected a string containing the file name, found** *token*

The optional token following the line number on a **#line** directive was not a string.

HI2131 **more than one memory attribute**

More than one of the keywords **_near**, **_far**, **_huge**, or **_based** were applied to an item, as in the following example:

```
typedef int _near nint;
nint _far a;    /* Illegal */
```

HI2132 **syntax error : unexpected identifier**

An identifier appeared in a syntactically illegal context.

HI2133 *identifier* **: unknown size**

An attempt was made to declare an unsized array as a local variable.

HI2134 *identifier* **: struct/union too large**

The size of a structure or union exceeded the 64K compiler limit.

HI2136 *function* **: prototype must have parameter types**

A function prototype declarator had formal-parameter names, but no types were provided for the parameters.

A formal parameter in a function prototype must either have a type or be represented by an ellipsis (...) to indicate a variable number of arguments and no type checking.

One cause of this error is a misspelling of a type name in a prototype that does not provide the names of formal parameters.

HI2137 **empty character constant**

The illegal empty-character constant (‘’) was used.

HI2139 **type following** *identifier* **is illegal**

Two types were used in the same declaration.

For example:

```
int double a;
```

HI2141 **value out of range for enum constant**

An enumeration constant had a value outside the range of values allowed for type **int**.

HI2143 **syntax error : missing** *token1* **before** *token2*

The compiler expected *token1* to appear before *token2*.

This message may appear if a required closing brace (}), right parenthesis ()), or semicolon (;) is missing.

HI2144 **syntax error : missing** *token* **before type** *type*

The compiler expected the given token to appear before the given type name.

This message may appear if a required closing brace (}), right parenthesis ()), or semicolon (;) is missing.

HI2145 **syntax error : missing** *token* **before identifier**

The compiler expected the given token to appear before an identifier.

This message may appear if a semicolon (;) does not appear after the last declaration of a block.

HI2146 **syntax error : missing** *token* **before identifier** *identifier*

The compiler expected the given token to appear before the given identifier.

HI2147 **unknown size**

An attempt was made to increment an index or pointer to an array whose base type has not yet been declared.

HI2148 **array too large**

An array exceeded the maximum legal size of 64K.

Either reduce the size of the array, or declare it with **_huge**.

HI2149 *identifier* **: named bit field cannot have 0 width**

The given named bit field had zero width. Only unnamed bit fields are allowed to have zero width.

HI2150 *identifier* **: bit field must have type int, signed int, or unsigned int**

The ANSI C standard requires bit fields to have types of **int**, **signed int**, or **unsigned int**. This message appears only when compiling with the /Za option.

HI2151 **more than one language attribute**

More than one keyword specifying a calling convention for a function was given.

HI2152 *identifier* **: pointers to functions with different attributes**

An attempt was made to assign a pointer to a function declared with one calling convention (**_cdecl**, **_fortran**, **_pascal**, or **_fastcall**) to a pointer to a function declared with a different calling convention.

HI2153 **hex constants must have at least 1 hex digit**

The hexadecimal constants 0x, 0X, and \x are illegal. At least one hexadecimal digit must follow the x or X.

HI2154 *segment* **: does not refer to a segment name**

A **_based**-allocated variable must be allocated in a segment unless it is extern and uninitialized.

HI2156 **pragma must be outside function**

A pragma that must be specified at a global level, outside a function body, occurred within a function.

For example, the following causes this error:

```
main()
{
#pragma optimize("l", on)
}
```

HI2157 *function* **: must be declared before use in pragma list**

The function name in the list of functions for an **alloc_text** pragma has not been declared prior to being referenced in the list.

HI2158 *identifier* **: is a function**

The given identifier was specified in the list of variables in a **same_seg** pragma but was previously declared as a function.

HI2159 **more than one storage class specified**

A declaration contained more than one storage class, as in

```
extern static int i;
```

HI2160 **## cannot occur at the beginning of a macro definition**

A macro definition began with a token-pasting operator (##), as in

```
#define mac(a,b) ##a
```

HI2161 **## cannot occur at the end of a macro definition**

A macro definition ended with a token-pasting operator (##), as in

```
#define mac(a,b) a##
```

HI2162 **expected macro formal parameter**

The token following a stringizing operator (#) was not a formal-parameter name.

For example:

```
#define print(a) printf(#b)
```

HI2165 *keyword* **: cannot modify pointers to data**

The **_fortran**, **_pascal**, **_cdecl**, or **_fastcall** keyword was used illegally to modify a pointer to data, as in the following example:

```
char _pascal *p;
```

HI2166 **lvalue specifies const object**

An attempt was made to modify an item declared with const type.

HI2167 *function* **: too many actual parameters for intrinsic function**

A reference to the intrinsic function name contained too many actual parameters.

HI2168 *function* **: too few actual parameters for intrinsic function**

A reference to the intrinsic function name contained too few actual parameters.

HI2171 *operator* **: illegal operand**

The given unary operator was used with an illegal operand type, as in the following example:

```
int (*fp)();
double d,d1;
fp++;
d = ~d1;
```

HI2172 *function* **: actual is not a pointer : parameter** *number*

An attempt was made to pass an argument that was not a pointer to a function that expected a pointer. The given number indicates which argument was in error.

HI2173 *function* **: actual is not a pointer : parameter** *number1*, **parameter list** *number2*

An attempt was made to pass a nonpointer argument to a function that expected a pointer.

This error occurs in calls that return a pointer to a function. The first number indicates which argument was in error; the second number indicates which argument list contained the invalid argument.

HI2174 *function* **: actual has type void : parameter** *number1*, **parameter list** *number2*

An attempt was made to pass a void argument to a function. Formal parameters and arguments to functions cannot have type void. They can, however, have type void * (pointer to void).

This error occurs in calls that return a pointer to a function. The first number indicates which argument was in error; the second number indicates which argument list contained the invalid argument.

HI2177 **constant too big**

Information was lost because a constant value was too large to be represented in the type to which it was assigned.

HI2178 *identifier* **: storage class for same_seg variables must be extern**

The given variable was specified in a **same_seg** pragma, but it was not declared with extern storage class.

HI2179 *identifier* **: was used in same_seg, but storage class is no longer extern**

The given variable was specified in a **same_seg** pragma, but it was redeclared with a storage class other than extern.

HI2185 *identifier* **: illegal _based allocation**

A **_based**-allocated variable that explicitly has extern storage class and is uninitialized may not have a base of any of the following:

```
(_segment) & var
_segname("_STACK")
(_segment)_self
void
```

If the variable does not explicitly have extern storage class or it is uninitialized, then its base must use _segname("*string*") where *string* is any segment name or reserved segment name except "_STACK".

HI2187 **cast of near function pointer to far function pointer**

An attempt was made to cast a near function pointer as a far function pointer.

HI2189 **#error :** *string*

An **#error** directive was encountered. The *string* is the descriptive text supplied in the directive.

HI2193 *identifier* **: already in a segment**

A variable in the **same_seg** pragma has already been allocated in a segment, using **_based**.

HI2194 *segment* **: is a text segment**

The given text segment was used where a data, const, or bss segment was expected.

HI2195 *segment* **: is a data segment**

The given data segment was used where a text segment was expected.

HI2200 *function* **: function has already been defined**

A function name passed as an argument in an **alloc_text** pragma has already been defined.

HI2201 *function* **: storage class must be extern**

A function declaration appears within a block, but the function is not declared extern. This causes an error if the /Za option is in effect.

For example, the following causes this error, when compiled with /Za:

```
main()
{
static int func1();
}
```

HI2205 *identifier* **: cannot initialize extern block-scoped variables**

A variable with extern storage class may not be initialized in a function.

HI2208 **no members defined using this type**

An **enum**, **struct**, or **union** was defined without any members. This is an error only when compiling with /Za; otherwise, it is a warning.

HI2209 **type cast in _based construct must be (_segment)**

The only type allowed within a cast in a **_based** declarator is (**_segment**).

HI2210 *identifier* **: must be near/far data pointer**

The base in a **_based** declarator may not be an array, a function, or a **_based** pointer.

HI2211 **(_segment) applied to function identifier** *function*

The item cast in a **_based** declarator must not be a function.

HI2212 *identifier* **: _based not available for functions/pointers to functions**

Functions cannot be **_based**-allocated. Use the **alloc_text** pragma.

HI2213 *identifier* **: illegal argument to _based**

A symbol used as a base must have type **_segment** or be a near or far pointer.

HI2214 **pointers based on void require the use of :>**

A **_based** pointer based on void cannot be dereferenced. Use the **:>** operator to create an address that can be dereferenced.

HI2215 **:> operator only for objects based on void**

The right operand of the **:>** operator must be a pointer based on void, as in

```
char _based(void) *cbvpi
```

HI2216 *attribute1* **may not be used with** *attribute2*

The given function attributes are incompatible.

Some combinations of attributes that cause this error are

- **_saveregs** and **_interrupt**
- **_fastcall** and **_saveregs**
- **_fastcall** and **_interrupt**
- **_fastcall** and **_export**

HI2217 *attribute1* **must be used with** *attribute2*

The first function attribute requires the second attribute to be used.

Some causes for this error include

- An interrupt function explicitly declared as near. Interrupt functions must be far.
- An interrupt function or a function with a variable number of arguments, when that function is declared with the **_fortran**, **_pascal**, or **_fastcall** attribute. Functions declared with the **_interrupt** attribute or with a variable number of arguments must use the C calling conventions. Remove the **_fortran**, **_pascal**, or **_fastcall** attribute from the function declaration.

HI2218 **type in _based construct must be void**

The only type allowed within a **_based** construct is void.

HI2219 **syntax error : type qualifier must be after '*'**

Either const or volatile appeared where a type or qualifier is not allowed, as in

```
int (const *p);
```

HI2220 **warning treated as error - no object file generated**

When the compiler option /WX is used, the first warning generated by the compiler causes this error message to be displayed.

Either correct the condition that caused the warning, or compile at a lower warning level or without /WX.

HI2221 **'.' : left operand points to struct/union, use ->**

The left operand of the '.' operator must be a **struct/union** type. It cannot be a pointer to a **struct/union** type.

This error usually means that a **->** operator must be used.

HI2222 **-> : left operand has struct/union type, use '.'**

The left operand of the **->** operator must be a pointer to a **struct/union** type. It cannot be a **struct/union** type.

This error usually means that a '.' operator must be used.

HI2223 **left of** *->member* **must point to struct/union**

The left operand of the **->** operator is not a pointer to a **struct/union** type.

This error can occur when the left operand is an undefined variable. Undefined variables have type **int**.

HI2224 **left of** *.member* **must have struct/union type**

The left operand of the '.' operator is not a **struct/union** type.

This error can occur when the left operand is an undefined variable. Undefined variables have type **int**.

HI2225 *tagname* **: first member of struct is unnamed**

The **struct** with the given tag started with an unnamed member (an alignment member). **Struct** definitions must start with a named member.

F.5.3 H2INC Warnings

Number	Message
HI4000	**UNKNOWN WARNING** **Contact Microsoft Product Support Services** The compiler detected an unknown error condition. Note the circumstances of the error and notify Microsoft Corporation by following the instructions on the Microsoft Product Assistance Request form at the back of one of your manuals.
HI4001	**nonstandard extension used -** *extension* The given nonstandard language extension was used when the /Ze option was specified. This is a level 4 warning, except in the case of a function pointer cast to data when the Quick Compile option, /qc, is in use, which produces a level 1 warning. If the /Za option has been specified, this condition generates a syntax error.
HI4002	**too many actual parameters for macro** *identifier* The number of actual arguments specified with the given identifier was greater than the number of formal parameters given in the macro definition of the identifier. The additional actual parameters are collected but ignored during expansion of the macro.
HI4003	**not enough actual parameters for macro** *identifier* The number of actual arguments specified with the given identifier was fewer than the number of formal parameters given in the macro definition of the identifier. When a formal parameter is referenced in the definition and the corresponding actual parameter has not been provided, empty text is substituted in the macro expansion.
HI4004	**missing ')' after** *defined* The closing parenthesis was missing from an **#if** defined phrase. The compiler assumes a right parenthesis,), after the first identifier it finds. It then attempts to compile the remainder of the line, which may result in another warning or error.

The following example causes this warning and a fatal error:

```
#if defined( ID1 ) || ( ID2 )
```

The compiler assumed a right parenthesis after ID1, then found a mismatched parenthesis in the remainder of the line. The following avoids this problem:

```
#if defined( ID1 ) || defined( ID2 )
```

HI4005 *identifier* **: macro redefinition**

The given identifier was defined twice. The compiler assumed the new macro definition.

To eliminate the warning, either remove one of the definitions or use an **#undef** directive before the second definition.

This warning is caused in situations where a macro is defined both on the command line and in the code with a **#define** directive.

HI4006 **#undef expected an identifier**

The name of the identifier whose definition was to be removed was not given with the **#undef** directive. The **#undef** was ignored.

HI4007 *identifier* **: must be** *attribute*

The attribute of the given function was not explicitly stated. The compiler forced the attribute.

For example, the function main must have the **_cdecl** attribute.

HI4008 *identifier* **: _fastcall attribute on data ignored**

The **_fastcall** attribute on the given data identifier was ignored.

HI4009 **string too big, trailing characters truncated**

A string exceeded the compiler limit of 2047 on string size. The excess characters at the end of the string were truncated.

To correct this problem, break the string into two or more strings.

HI4011 **identifier truncated to** *identifier*

Only the first 31 characters of an identifier are significant. The characters after the limit were truncated.

This may mean that two identifiers that are different before truncation may have the same identifier name after truncation.

HI4015 *identifier* **: bit-field type must be integral**

The given bit field was not declared as an integral type. The compiler assumed the base type of the bit field to be unsigned.

Bit fields must be declared as unsigned integral types.

HI4016 *function* **: no function return type, using int as default**

The given function had not yet been declared or defined, so the return type was unknown. A default return type of **int** was assumed.

HI4017 **cast of int expression to far pointer**

A far pointer represents a full segmented address. On an 8086/8088 processor, casting an **int** value to a far pointer may produce an address with a meaningless segment value.

The compiler extended the **int** expression to a four-byte value.

HI4020 *function* **: too many actual parameters**

The number of arguments specified in a function call was greater than the number of parameters specified in the function prototype or function definition.

The extra parameters were passed according to the calling convention used on the function.

HI4021 *function* **: too few actual parameters**

The number of arguments specified in a function call was less than the number of parameters specified in the function prototype or function definition.

Only the provided actual parameters are passed. If the called function references a variable that was not passed, the results are undefined and may be unexpected.

HI4022 *function* **: pointer mismatch : parameter** *number*

The pointer type of the given parameter was different from the pointer type specified in the argument-type list or function definition.

The parameter will be passed without change. Its value will be interpreted as a pointer within the called function.

HI4023 *function* **: _based pointer passed to unprototyped function : parameter** *number*

When in a near data model, only the offset portion of a **_based** pointer is passed to an unprototyped function. If the function expects a far pointer, the resulting code will be wrong. In any data model, if the function is defined to take a **_based** pointer with a different base, the resulting code may be unpredictable.

If a prototype is used before the call, the call will be generated correctly.

HI4024 *function* **: different types : parameter** *number*

The type of the given parameter in a function call did not agree with the type given in the argument-type list or function definition.

The parameter will be passed without change. The function will interpret the parameter's type as the type expected by the function.

HI4028 **parameter** *number* **declaration different**

The type of the given parameter did not agree with the corresponding type in the argument-type list or with the corresponding formal parameter.

The original declaration was used.

HI4030 **first parameter list longer than the second**

A function was declared more than once with different parameter lists.

The first declaration was used.

HI4031 **second parameter list is longer than the first**

A function was declared more than once with different parameter lists.

The first declaration was used.

HI4034 **sizeof returns 0**

The **sizeof** operator was applied to an operand that yielded a size of zero.

This warning is informational.

HI4040 **memory attribute on** *identifier* **ignored**

The **_near**, **_far**, **_huge**, or **_based** keyword has no effect in the declaration of the given identifier and is ignored.

One cause of this warning is a huge array that is not declared globally. Declare huge arrays outside of **main**.

HI4042 *identifier* **: has bad storage class**

The storage class specified for *identifier* cannot be used in this context.

The default storage class for this context was used in place of the illegal class:

- If *identifier* was a function, the compiler assumed extern class.
- If *identifier* was a formal parameter or local variable, the compiler assumed auto class.
- If *identifier* was a global variable, the compiler assumed that the variable was declared with no storage class.

HI4044 **_huge on *identifier* ignored, must be an array**

The compiler ignored the **_huge** memory attribute on the given identifier. Only arrays may be declared with the **_huge** memory attribute. On pointers, **_huge** must be used as a modifier, not as a memory attribute.

HI4047 ***operator*** **: different levels of indirection**

An expression involving the specified operator had inconsistent levels of indirection.

If both operands are of arithmetic type, or if both are not (such as two arrays or pointers), then they are used without change, though the compiler may DS-extend one of the operands if one is far and one is near. If one is arithmetic and one is not, the arithmetic operator is converted to the type of the other operator.

For example, the following code causes this warning but is compiled without change:

```
char **p;
char *q;
p = q;   /* Warning */
```

HI4048 **array's declared subscripts different**

An expression involved pointers to arrays of different size.

The pointers were used without conversion.

HI4049 ***operator*** **: indirection to different types**

The pointer expressions used with the given operator had different base types.

The expressions were used without conversion.

For example, the following code causes this warning:

```
struct ts1 *s1;
struct ts2 *s2;
s2 = s1;   /* Warning */
```

HI4050 ***operator*** **: different code attributes**

The function-pointer expressions used with *operator* had different code attributes. The attribute involved is either **_export** or **_loadds**.

This is a warning and not an error, because **_export** and **_loadds** affect only entry sequences and not calling conventions.

HI4051 **type conversion, possible loss of data**

Two data items in an expression had different base types, causing the type of one item to be converted. During the conversion, a data item was truncated.

HI4053 **at least one void operand**

An expression with type void was used as an operand.

The expression was evaluated using an undefined value for the void operand.

HI4063 *function* **: function too large for post-optimizer**

Not enough space was available to optimize the given function.

One of the following may be a solution:

- Recompile with fewer optimizations.
- Divide the function into two or more smaller functions.
- In OS/2, recompile using the /B3 C3L option to invoke the large-model version of the third pass of the compiler.

HI4066 **local symbol-table overflow - some local symbols may be missing in listings**

The listing generator ran out of heap space for local variables, so the source listing may not contain symbol-table information for all local variables.

HI4067 **unexpected characters following** *directive* **directive - newline expected**

Extra characters followed a preprocessor directive and were ignored. This warning appears only when compiling with the /Za option.

For example, the following code causes this warning:

```
#endif    NO_EXT_KEYS
```

To remove the warning, compile with /Ze or use comment delimiters:

```
#endif    /* NO_EXT_KEYS */
```

HI4071 *function* **: no function prototype given**

The given function was called before the compiler found the corresponding function prototype.

The function will be called using the default rules for calling a function without a prototype.

HI4072 *function* **: no function prototype on _fastcall function**

A **_fastcall** function was called without first being prototyped.

Functions that are **_fastcall** should be prototyped to guarantee that the registers assigned at each point of call are the same as the registers assumed when the function is defined. A function defined in the new ANSI style is a prototype.

A prototype must be added when this warning appears, unless the function takes no arguments or takes only arguments that cannot be passed in the general-purpose registers.

HI4073 **scoping too deep, deepest scoping merged when debugging**

Declarations appeared at a static nesting level greater than 13. As a result, all declarations beyond this level will seem to appear at the same level.

HI4076 *type* **: may be used on integral types only**

The signed or unsigned type modifier was used with a nonintegral type.

The given qualifier was ignored.

The following example causes this warning:

```
unsigned double x;
```

HI4079 **unexpected token** *token*

An unexpected separator token was found in the argument list of a pragma.

The remainder of the pragma was ignored.

HI4082 **expected an identifier, found** *token*

An identifier was missing from the argument list.

The remainder of the pragma was ignored.

HI4083 **expected '(', found** *token*

A left parenthesis, (, was missing from a pragma's argument list.

The pragma was ignored.

The following example causes this warning:

```
#pragma check_pointer on)
```

HI4084 **expected a pragma keyword, found** *token*

The *token* following **#pragma** was not recognized as a directive.

The pragma was ignored.

The following example causes this warning:

```
#pragma (on)
```

HI4085 **expected [on | off]**

The pragma expected an on or off parameter, but the specified parameter was unrecognized or missing.

The pragma was ignored.

HI4086 **expected [1 | 2 | 4]**

The pragma expected a parameter of either 1, 2, or 4, but the specified parameter was unrecognized or missing.

HI4087 *function* **: declared with void parameter list**

The given function was declared as taking no parameters, but a call to the function specified actual parameters.

The extra parameters were passed according to the calling convention used on the function.

The following example causes this warning:

```
int f1(void);
f1(10);
```

HI4088 *function* **: pointer mismatch : parameter** *number*, **parameter list** *number*

The argument passed to the given function had a different level of indirection from the given parameter in the function definition.

The parameter will be passed without change. Its value will be interpreted as a pointer within the called function.

HI4089 *function* **: different types : parameter** *number*, **parameter list** *number*

The argument passed to the given function did not have the same type as the given parameter in the function definition.

The parameter will be passed without change. The function will interpret the parameter's type as the type expected by the function.

HI4090 **different const/volatile qualifiers**

A pointer to an item declared as const was assigned to a pointer that was not declared as const. As a result, the const item pointed to could be modified without being detected.

The expression was compiled without modification.

The following example causes this warning:

```
const char *p = "abcde";
int str(char *s);
str(p);
```

HI4091 **no symbols were declared**

The compiler detected an empty declaration, as in the following example:

```
int ;
```

The declaration was ignored.

HI4092 **untagged enum/struct/union declared no symbols**

The compiler detected an empty declaration using an untagged structure, union, or enumerated variable. The declaration was ignored.

For example, the following code causes this warning:

```
struct { . . . };
```

HI4093 **unescaped newline in character constant in inactive code**

The constant expression of an **#if**, **#elif**, **#ifdef**, or **#ifndef** preprocessor directive evaluated to 0, making the code that follows inactive. Within that inactive code, a newline character appeared within a set of single or double quotation marks.

All text until the next double quotation mark was considered to be within a character constant.

HI4095 **expected ')', found** *token*

More than one argument was given for a pragma that can take only one argument.

The compiler assumed the expected parenthesis and ignored the remainder of the line.

HI4096 *attribute1* **must be used with** *attribute2*

The use of attribute2 requires the use of attribute1.

For example, using a variable number of arguments (...) requires that **_cdecl** be used. Also, **_interrupt** functions must be **_far** and **_cdecl**.

The compiler assumed *attribute1* for the function.

HI4098 **void function returning a value**

A function declared with a void return type also returned a value.

A function was declared with a void return type but was defined as a value.

The compiler assumed the function returns a value of type **int**.

HI4104 *identifier* **: near data in same_seg pragma, ignored**

The given near variable was specified in a **same_seg** pragma.

The *identifier* was ignored.

HI4105 *identifier* **: code modifiers only on function or pointer to function**

The given identifier was declared with a code modifier that can be used only with a function or function pointer.

The code modifier was ignored.

HI4109 **unexpected identifier** *identifier*

The pragma contained an unexpected token.

The pragma was ignored.

HI4110 **unexpected token** *int constant*

The pragma contained an unexpected integer constant.

The pragma was ignored.

HI4111 **unexpected token** *string*

The pragma contained an unexpected string.

The pragma was ignored.

HI4112 **macro name** *name* **is reserved,** *command* **ignored**

The given command attempted to define or undefine the predefined macro *name* or the preprocessor operator *defined.* The given command is displayed as either **#define** or **#undef**, even if the attempt was made using command-line options.

The command was ignored.

HI4113 **function parameter lists differed**

A function pointer was assigned to a function pointer, but the parameter lists of the functions do not agree.

The expression was compiled without modification.

HI4114 **same type qualifier used more than once**

A type qualifier (const, volatile, signed, or unsigned) was used more than once in the same type.

The second occurrence of the qualifier was ignored.

HI4115 *tag* **: type definition in formal parameter list**

The given tag was used to define a **struct**, **union**, or **enum** in the formal parameter list of a function.

The compiler assumed the definition was at the global level.

HI4116 **(no tag) : type definition in formal parameter list**

A **struct**, **union**, or **enum** type with no tag was defined in the formal parameter list of a function.

The compiler assumed the definition was at the global level.

HI4119 **different bases** *name1* **and** *name2* **specified**

The **_based** pointers in the expression have different symbolic bases. There may be truncation or loss in the code generated.

HI4120 **_based/unbased mismatch**

The expression contains a conversion between a **_based** pointer and another pointer that is unbased. Some information may have been truncated.

This warning commonly occurs when a **_based** pointer is passed to a function that accepts a near or far pointer.

HI4123 **different base expressions specified**

The expression contains a conversion between **_based** pointers, but the base expressions of the **_based** pointers are different. Some of the **_based** conversions may be unexpected.

HI4125 **decimal digit terminates octal escape sequence**

An octal escape sequence in a character or string constant was terminated with a decimal digit.

The compiler evaluated the octal number without the decimal digit and assumed the decimal digit was a character.

The following example causes this warning:

```
char array1[] = "\709";
```

If the digit 9 was intended as a character and was not a typing error, correct the example as follows:

```
char array[] = "\0709";  /* String containing "89" */
```

HI4126 *flag* **: unknown memory model flag**

The flag used with the /A option was not recognized and was ignored.

HI4128 **storage-class specifier after type**

A storage-class specifier (auto, extern, register, static) appears after a type in a declaration. The compiler assumed that the storage class specifier occurred before the type.

New-style code places the storage-class specifier first.

HI4129 *character* **: unrecognized character escape sequence**

The *character* following a backslash in a character or string constant was not recognized as a valid escape sequence.

As a result, the backslash is ignored and not printed, and the character following the backslash is printed.

To print a single backslash (\), specify a double backslash (\\).

HI4130 *operator* **: logical operation on address of string constant**

The operator was used with the address of a string literal. Unexpected code was generated.

For example, the following code causes this warning:

```
char *pc;
pc = "Hello";
if (pc == "Hello") ...
```

The if statement compares the value stored in the pointer pc to the address of the string "Hello", which is separately allocated each time it occurs in the code. It does not compare the string pointed to by pc with the string "Hello".

To compare strings, use the **strcmp** function.

HI4131 *function* **: uses old-style declarator**

The function declaration or definition is not a prototype.

New-style function declarations are in prototype form.

- old style

```
int addrec( name, id )
char *name;
int id;
{ }
```

- new style

```
int addrec( char *name, int id )
{ }
```

HI4132 *object* **: const object should be initialized**

The value of a const object cannot be changed, so the only way to give the const object a value is to initialize it.

It will not be possible to assign a value to *object*.

HI4135 **conversion between different integral types**

Information was lost between two integral types.

For example, the following code causes this warning:

```
int intvar;
long longvar;
intvar = longvar;
```

If the information is merely interpreted differently, this warning is not given, as in the following example:

```
unsigned uintvar = intvar;
```

HI4136 **conversion between different floating types**

Information was lost or truncated between two floating types.

For example, the following code causes this warning:

```
double doublevar;
float floatvar;
floatvar = doublevar;
```

Note that unsuffixed floating-point constants have type double, so the following code causes this warning:

```
floatvar = 1.0;
```

If the floating-point constant should be treated as float type, use the F (or f) suffix on the constant to prevent the following warning:

```
floatvar = 1.0F;
```

HI4138 ***/ found outside of comment**

The compiler found a closing comment delimiter (*/) without a preceding opening delimiter. It assumed a space between the asterisk (*) and the forward slash (/).

The following example causes this warning:

```
int */*comment*/ptr;
```

In this example, the compiler assumed a space before the first comment delimiter (/*) and issued the warning but compiled the line normally. To remove the warning, insert the assumed space.

Usually, the cause of this warning is an attempt to nest comments.

To comment out sections of code that may contain comments, enclose the code in an **#if/#endif** block and set the controlling expression to zero, as in:

```
#if 0
int my_variable;   /* Declaration currently not needed */
#endif
```

HI4139 *hexnumber* **: hex escape sequence is out of range**

A hex escape sequence appearing in a character or string constant was too large to be converted to a character.

If in a string constant, the compiler cast the low byte of the hexadecimal number to a char. If in a char constant, the compiler made the cast and then sign extended the result. If in a char constant and compiled with /J, the compiler cast the value to an unsigned char.

For example, *\x1ff* is out of range for a character. Note that the following code causes this warning:

```
printf("\x7Bell\n");
```

The number 7be is a legal hex number but is too large for a character. To correct this example, use three hex digits:

```
printf("\x007Bell\n");
```

HI4186 **string too long - truncated to 40 characters**

The string argument for a title or subtitle pragma exceeded the maximum allowable length and was truncated.

HI4200 **local variable** *identifier* **used without having been initialized**

A reference was made to a local variable that had not been assigned a value. As a result, the value of the variable is unpredictable.

This warning is given only when compiling with global register allocation on (/Oe).

HI4201 **local variable** *identifier* **may be used without having been initialized**

A reference was made to a local variable that might not have been assigned a value. As a result, the value of the variable may be unpredictable.

This warning is given only when compiling with the global register allocation on (/Oe).

HI4202 **unreachable code**

The flow of control can never reach the indicated line.

This warning is given only when compiling with one of the global optimizations (/Oe, /Og, or /Ol).

HI4203 *function* **: function too large for global optimizations**

The named function was too large to fit in memory and be compiled with the selected optimization. The compiler did not perform any global optimizations (/Oe, /Og, or /Ol). Other /O optimizations, such as /Oa and /Oi, are still performed.

One of the following may remove this warning:

- Recompile with fewer optimizations.
- Divide the function into two or more smaller functions.
- In OS/2, recompile using the /B2 C2L option to invoke the large-model version of the second pass of the compiler.

HI4204 *function* **: in-line assembler precludes global optimizations**

The use of in-line assembler in the named function prevented the specified global optimizations (/Oe, /Og, or /Ol) from being performed.

HI4205 **statement has no effect**

The indicated statement will have no effect on the program execution.

Some examples of statements with no effect:

```
1;
a + 1;
b == c;
```

HI4209 **comma operator within array index expression**

The value used as an index into an array was the last one of multiple expressions separated by the comma operator.

An array index legally may be the value of the last expression in a series of expressions separated by the comma operator. However, the intent may have been to use the expressions to specify multiple indexes into a multidimensional array.

For example, the following line, which causes this warning, is legal in C, and specifies the index c into array a:

```
a[b,c]
```

However, the following line uses both b and c as indexes into a two-dimensional array:

```
a[b][c]
```

HI4300 insufficient memory to process debugging information

The program was compiled with the /Zi option, but not enough memory was available to create the required debugging information.

One of the following may be a solution:

- Split the current file into two or more files and compile them separately.
- Remove other programs or drivers running in the system which could be consuming significant amounts of memory.
- In OS/2, recompile using the /B3 C3L option to invoke the large-model version of the third pass of the compiler.

HI4301 loss of debugging information caused by optimization

Some optimizations, such as code motion, cause references to nested variables to be moved. The information about the level at which the variables are declared may be lost. As a result, all declarations will seem to be at nesting level 1.

HI4323 potential divide by 0

The second operand in a divide operation evaluated to zero at compile time, giving undefined results.

The 0 operand may have been generated by the compiler, as in the following example:

```
func1() { int i,j,k; i /= j && k; }
```

HI4324 potential mod by 0

The second operand in a remainder operation evaluated to zero at compile time, giving undefined results.

HI4800 more than one memory model specified

There was more than one memory model given at the command line. The /AT, /AS, /AM, /AC, /AL, and /AH options specify the memory model.

This error is caused by conflicting options specified at the command line and in the CL and H2INC environment variables.

HI4801 more than one target processor specified

There was more than one processor type given at the command line. The /G0, /G1, and /G2 options specify the processor type.

This error is caused by conflicting options specified at the command line and in the CL and H2INC environment variables.

HI4802 **ignoring invalid /Zp value** *value*

The alignment value specified to the /Zp option was not 1, 2, or 4. The default of 1 was assumed.

HI4810 **untranslatable basic type size**

H2INC could not translate the item to a MASM type.

The C void type cannot be translated to a similar MASM type.

HI4811 **static function prototype not translated**

H2INC does not translate static items, as they are not visible outside the C source file.

HI4812 **static variable declaration not accepted with /Mn switch**

H2INC does not translate static items, as they are not visible outside the C source file.

HI4815 *string* **: EQU string truncated to 254 characters**

A **#define** statement exceeded 254 characters, the maximum length of a MASM **EQU** statement. The string was truncated.

HI4816 **ignoring _fastcall function definition**

H2INC does not translate function declarations or prototypes with the **_fastcall** attribute. The **_fastcall** calling convention cannot be used directly with MASM. See the documentation with your C compiler for details on **_fastcall**.

HI4820 **ignoring function definition :** *function*()

H2INC does not translate function bodies.

H2INC translates header information only; it cannot convert program code.

F.6 IMPLIB Error Messages

This section lists error messages generated by the Microsoft Import Library Manager (IMPLIB):

- Fatal errors (`IM16`*xx*) cause IMPLIB to stop execution.
- Errors (`IM26`*xx*) prevent IMPLIB from creating an import library.

F.6.1 IMPLIB Fatal Errors

Number	IMPLIB Error Message
IM1600	**out of space on output file** The drive or directory where the import library is being created is full.
IM1601	**out of heap space** There was not enough room in memory for the heap needed by IMPLIB. Increase the available memory.
IM1602	**syntax error in the module definitions file** IMPLIB could not understand the contents of the module-definition file.
IM1603	*filename* **: cannot create file** IMPLIB could not create the given file. One of the following may be a cause: ■ The file already exists with a read-only attribute. ■ There is insufficient disk space to create the file. ■ The drive cannot be written to.
IM1604	*filename* **: cannot open file** IMPLIB could not find the specified module-definition file or DLL.

F.6.2 IMPLIB Errors

Number	IMPLIB Error Message
IM2600	**string too long in line** *number*; **truncated to 512 characters** The given line in the module-definition file exceeded the limit on line length. IMPLIB ignored text after the first 512 characters.
IM2601	*symbol* **multiply defined** The given symbol was defined more than once in the input files.
IM2602	**unexpected end of name table in DLL** A DLL input file was corrupted.

IM2603 *filename* **: invalid .DLL file**

The given DLL input file was corrupted.

F.7 LIB Error Messages

This section lists error messages generated by the Microsoft Library Manager (LIB):

- Fatal errors (U11*xx*) cause LIB to stop execution.
- Errors (U21*xx*) do not stop execution but prevent LIB from creating a library.
- Warnings (U41*xx*) indicate possible problems in the library being created.

F.7.1 LIB Fatal Errors

Number	LIB Error Message

U1150 **page size too small**

The page size of an input library was too small, indicating an invalid input .LIB file.

U1151 **syntax error : illegal file specification**

A command operator was not followed by a module name or filename.

One possible cause of this error is an option specified with a dash (-) instead of a forward slash (/).

U1152 **syntax error : option name missing**

A forward slash (/) appeared on the command line without an option name after it.

U1153 **syntax error : option value missing**

The /PAGE option was given without a value following it.

U1154 **unrecognized option**

An unrecognized name followed the option indicator.

An option is specified by a forward slash (/) and a name. The name can be specified by a legal abbreviation of the full name.

U1155 **syntax error : illegal input**

A specified command did not follow correct LIB syntax.

U1156 **syntax error**

A specified command did not follow correct LIB syntax.

U1157 **comma or newline missing**

A comma or carriage return was expected in the command line but did not appear.

This may indicate an incorrectly placed comma, as in the following command line:

```
LIB math.lib, -mod1 +mod2;
```

The line must be entered as follows:

```
LIB math.lib -mod1 +mod2;
```

U1158 **terminator missing**

The last line of the response file used to start LIB did not end with a carriage return.

U1159 **option argument missing**

An expected argument to an option or command was missing from the command line.

U1160 **invalid page size**

The argument specified with the /PAGE option was not valid for that option. The value must be an integer power of 2 between 16 and 32,768.

U1161 **cannot rename old library**

LIB could not rename the old library with a .BAK extension because the .BAK version already existed with read-only protection.

Change the protection on the old .BAK version.

U1162 **cannot reopen library**

The old library could not be reopened after it was renamed with a .BAK extension.

One of the following may have occurred:

- Another process deleted the file or changed it to read-only.
- The floppy disk containing the file was removed.
- A hard-disk error occurred.

U1163 **error writing to cross-reference file**

The disk or root directory was full.

Delete or move files to make space.

U1164 **name length exceeds 255 characters**

A filename specified on the command line exceeded the LIB limit of 255 characters. Reduce the number of characters in the name.

U1170 **too many symbols**

The number of symbols in all object files and libraries exceeded the capacity of the dictionary created by LIB.

Create two or more smaller libraries.

U1171 **insufficient memory**

LIB did not have enough memory to run.

Remove any shells or resident programs and try again, or add more memory.

U1172 **no more virtual memory**

The LIB session required more memory than the one-megabyte limit imposed by LIB.

Try using the /NOE option or reducing the number of object modules.

U1173 **internal failure**

Note the circumstances of the error and notify Microsoft Corporation by following the instructions on the Microsoft Product Assistance Request form at the back of one of your manuals.

U1174 **mark : not allocated**

Note the circumstances of the error and notify Microsoft Corporation by following the instructions on the Microsoft Product Assistance Request form at the back of one of your manuals.

U1175 **free : not allocated**

Note the circumstances of the error and notify Microsoft Corporation by following the instructions on the Microsoft Product Assistance Request form at the back of one of your manuals.

U1180 **write to extract file failed**

The disk or root directory was full.

Delete or move files to make space.

U1181 **write to library file failed**

The disk or root directory was full.

Delete or move files to make space.

U1182 *filename* **: cannot create extract file**

The disk or root directory was full, or the given extract file already existed with read-only protection.

Make space on the disk or change the protection of the extract file.

U1183 **cannot open response file**

The response file was not found.

U1184 **unexpected end-of-file on command input**

An end-of-file character was received prematurely in response to a prompt.

U1185 **cannot create new library**

The disk or root directory was full, or the library file already existed with read-only protection.

Make space on the disk or change the protection of the library file.

U1186 **error writing to new library**

The disk or root directory was full.

Delete or move files to make space.

U1187 **cannot open temporary file VM.TMP**

The disk or root directory was full.

Delete or move files to make space.

U1188 **insufficient disk space for temporary file**

The library manager cannot write to the virtual memory.

Note the circumstances of the error and notify Microsoft Corporation by following the instructions on the Microsoft Product Assistance Request form at the back of one of your manuals.

U1189 **cannot read from temporary file**

The library manager cannot read the virtual memory.

Note the circumstances of the error and notify Microsoft Corporation by following the instructions on the Microsoft Product Assistance Request form at the back of one of your manuals.

U1190 **interrupted by user**

LIB was interrupted during its operation, with either CTRL+C or CTRL+BREAK.

U1200 *filename* **: invalid library header**

The input library file had an invalid format.

Either it was not a library file, or it had been corrupted.

U1203 *filename* **: invalid object file near** *location*

The given file was not a valid object file or was corrupted at the given location.

F.7.2 LIB Errors

Number LIB Error Message

U2152 *filename* **: cannot create listing**

One of the following may have occurred:

- The directory or disk was full.
- The cross-reference-listing file already existed with read-only protection.

U2155 *module* **: module not in library; ignored**

The specified module was not found in the input library.

One cause of this error is a filename or directory name containing a hyphen, also called a dash (-). LIB interprets the dash as the operator for the delete command. This error occurs if you install a Microsoft language product in a directory that has a dash in its pathname, such as C:\MS-C. The SETUP program calls LIB to create the Microsoft combined libraries, but the dash in the command line passed to LIB causes the library-building session to fail.

Another possible cause of this error is an option specified with a dash (-) instead of a forward slash (/).

U2157 *filename* **: cannot access file**

LIB was unable to open the specified file, probably because the file did not exist.

Check the path specification and filename.

U2158 *library* **: invalid library header; file ignored**

The given library had an incorrect format and was not combined.

U2159 *filename* **: invalid format (***number***); file ignored**

The given file was not recognized as a XENIX archive and was not combined.

F.7.3 LIB Warnings

Number	LIB Warning

U4150 *module* **: module redefinition ignored**

A module was specified with the + operator to be added to a library, but a module having that name was already in the library.

One cause of this error is an incorrect specification of the replace operator, – +.

U4151 *symbol* **: symbol defined in module** *module***; redefinition ignored**

The given symbol was defined in more than one module.

U4153 *option* **:** *value* **: page size invalid; ignored**

The argument specified with the /PAGE option was not valid for that option. The value must be an integer power of 2 between 16 and 32,768. LIB assumed an existing page size from a library being combined.

U4155 *modulename* **: module not in library**

The given module specified with a command operator does not exist in the library.

If the replacement command (-+) was specified, LIB addded the file anyway. If the delete (-), copy (*), or move (-*) command was specified, LIB ignored the command.

U4156 *library* **: output-library specification ignored**

A new library was created because the filename specified in the *oldlibrary* field did not exist, but a filename was also specified in the *newlibrary* field. LIB ignored the *newlibrary* specification.

For example, both of the following command lines cause this error if project.lib does not already exist:

```
LIB project.lib +one.obj, new.lst, project.lib

LIB project.lib +one.obj, new.lst, new.lib
```

U4157 **insufficient memory, extended dictionary not created**

Insufficient memory prevented LIB from creating an extended dictionary.

The library is still valid, but the linker cannot take advantage of the extended dictionary to speed linking.

U4158 **internal error, extended dictionary not created**

An internal error prevented LIB from creating an extended dictionary.

The library is still valid, but the linker cannot take advantage of the extended dictionary to speed linking.

F.8 LINK Error Messages

This section lists error messages generated by the Microsoft Segmented-Executable Linker (LINK):

- Fatal errors (L1*xxx*) cause LINK to stop execution.
- Errors (L2*xxx*) do not stop execution but prevent LINK from creating an output file.
- Warnings (L4*xxx*) indicate possible problems in the output file being created.

F.8.1 LINK Fatal Errors

Number	LINK Error Message

L1001 *option* **: option name ambiguous**

A unique option name did not appear after the option indicator.

An option is specified by a forward-slash indicator (/) and a name. The name can be specified by an abbreviation of the full name, but the abbreviation must be unambiguous.

For example, many options begin with the letter N, so the following command causes this error:

```
LINK /N main;
```

L1003 **/Q and /EXEPACK incompatible**

LINK cannot be given both the /Q option and the /EXEPACK option.

L1004 *value* **: invalid numeric value**

An incorrect value appeared for a LINK option. For example, this error occurs when a character string is specified with an option that requires a numeric value.

L1005 *option* **: packing limit exceeds 64K**

The value specified with the /PACKC or /PACKD option exceeded the limit of 65,536 bytes.

L1006 *number* **: stack size exceeds 64K-1**

The value given as a parameter to the /STACK option exceeded the allowed maximum of 65,535 bytes.

L1007 **/OVERLAYINTERRUPT : interrupt number exceeds 255**

An overlay interrupt number greater than 255 was specified with the /OV option value.

Check the *DOS Technical Reference* or other DOS technical manual for information about interrupts.

L1008 **/SEGMENTS : segment limit set too high**

The /SEG option was specified with a limit on the number of definitions of logical segments that was impossible to satisfy.

L1009 *value* **: /CPARM : illegal value**

The value specified with the /CPARM option was not in the range 1–65,535.

L1020 **no object modules specified**

No object-file names were specified to the linker.

L1021 **cannot nest response files**

A response file occurred within a response file.

L1022 **response line too long**

A line in a response file was longer than 255 characters.

L1023 **terminated by user**

CTRL+C was entered.

L1024 **nested right parentheses**

The contents of an overlay were typed incorrectly on the command line.

L1025 **nested left parentheses**

The contents of an overlay were typed incorrectly on the command line.

L1026 **unmatched right parenthesis**

A right parenthesis was missing from the contents specification of an overlay on the command line.

L1027 **unmatched left parenthesis**

A left parenthesis was missing from the contents specification of an overlay on the command line.

L1030 **missing internal name**

An **IMPORTS** statement specified an ordinal in the module-definition file without including the internal name of the routine.

The name must be given if the import is by ordinal.

L1031 **module description redefined**

A **DESCRIPTION** statement in the module-definition file was specified more than once.

L1032 **module name redefined**

The module name was specified more than once (in a **NAME** or **LIBRARY** statement).

L1040 **too many exported entries**

The program exceeded the limit of 65,535 exported names.

L1041 **resident names table overflow**

The size of the resident names table exceeded 65,535 bytes.

An entry in the resident names table is made for each exported routine designated **RESIDENTNAME** and consists of the name plus three bytes of information. The first entry is the module name.

Reduce the number of exported routines or change some to nonresident status.

L1042 **nonresident names table overflow**

The size of the nonresident names table exceeded 65,535 bytes.

An entry in the nonresident names table is made for each exported routine not designated **RESIDENTNAME** and consists of the name plus three bytes of information. The first entry is the **DESCRIPTION** statement.

Reduce the number of exported routines or change some to resident status.

L1043 **relocation table overflow**

More than 32,768 long calls, long jumps, or other long pointers appeared in the program.

Try replacing long references with short references where possible.

L1044 **imported names table overflow**

The size of the imported names table exceeds 65,535 bytes.

An entry in the imported names table is made for each new name given in the IMPORTS section, including the module names, and consists of the name plus one byte.

Reduce the number of imports.

L1045 **too many TYPDEF records**

An object module contained more than 255 TYPDEF records. These records describe communal variables.

This error can appear only with programs produced by the Microsoft FORTRAN Compiler or other compilers that support communal variables. (TYPDEF is a DOS term. It is explained in the *Microsoft MS-DOS Programmer's Reference* and in other reference books on DOS.)

L1046 **too many external symbols in one module**

An object module specified more than the limit of 1,023 external symbols.

Break the module into smaller parts.

L1047 **too many group, segment, and class names in one module**

The program contained too many group, segment, and class names.

Reduce the number of groups, segments, or classes. Re-create the object file.

L1048 **too many segments in one module**

An object module had more than 255 segments.

Split the module or combine segments.

L1049 **too many segments**

The program had more than the maximum number of segments.

Use the /SEG option when linking to specify the maximum legal number of segments. The range of valid settings is 0–3,072. The default is 128.

L1050 **too many groups in one module**

LINK encountered more than 21 group definitions (GRPDEF) in a single module.

Reduce the number of group definitions or split the module. (Group definitions are explained in the *Microsoft MS-DOS Programmer's Reference* and in other reference books on DOS.)

L1051 **too many groups**

The program defined more than 20 groups, not counting DGROUP.

Reduce the number of groups.

L1052 **too many libraries**

An attempt was made to link with more than 32 libraries.

Combine libraries, or use modules that require fewer libraries.

L1053 **out of memory for symbol table**

The program had more symbolic information (such as public, external, segment, group, class, and file names) than could fit in available memory.

Try freeing memory by linking from the DOS command level instead of from a MAKE file or an editor. Otherwise, combine modules or segments and try to eliminate as many public symbols as possible.

L1054 **requested segment limit too high**

LINK did not have enough memory to allocate tables describing the number of segments requested. The number of segments is the default of 128 or the value specified with the /SEG option.

Try linking again by using the /SEG option to select a smaller number of segments (for example, use 64 if the default was used previously), or free some memory by eliminating resident programs or shells.

L1056 **too many overlays**

The program defined more than 63 overlays.

L1057 **data record too large**

An LEDATA record (in an object module) contained more than 1,024 bytes of data. This is a translator error. (LEDATA is a DOS term explained in the *Microsoft MS-DOS Programmer's Reference* and in other DOS reference books.)

Note which translator (compiler or assembler) produced the incorrect object module. Please report the circumstances of the error to Microsoft Corporation by following the instructions on the Microsoft Product Assistance Request form at the back of one of your manuals.

L1061 **out of memory for /INCR**

LINK ran out of memory when trying to process the additional information required for ILINK support.

Disable incremental linking.

L1062 **too many symbols for /INCR**

The program had more symbols than can be stored in the .SYM file.

Reduce the number of symbols or disable incremental linking.

L1063 **out of memory for CodeView information**

LINK was given too many object files with debug information, and it ran out of space to store them.

Reduce the number of object files that have full debug information by compiling some files with either /Zd instead of /Zi or no CodeView option at all.

L1064 **out of memory—*near/far* heap exhausted**

LINK was not able to allocate enough memory for the given heap.

One of the following may be a solution:

- Under OS/2, increase the swap space.
- Reduce the size of code, data, and symbols in the program.
- Under OS/2, split the program into dynamic-link libraries.

L1070 *segment* **: segment size exceeds 64K**

A single segment contained more than 64K of code or data.

Try changing the memory model to use far code or data as appropriate. If the program is in C, use CL's /NT option or the **#pragma alloc_text** to build smaller segments.

L1071 **segment _TEXT exceeds 64K – 16**

This error is likely to occur only in small-model C programs, but it can occur when any program with a segment named _TEXT is linked using the /DOSSEG option of the LINK command.

Small-model C programs must reserve code addresses 0 and 1; this range is increased to 16 for alignment purposes.

Try compiling and linking using the medium or large model. If the program is in C, use CL's /NT option or the **#pragma alloc_text** to build smaller segments.

L1072 **common area exceeds 64K**

The program had more than 65,536 bytes of communal variables. This error occurs only with programs produced by the Microsoft FORTRAN Compiler or other compilers that support communal variables.

L1073 **file-segment limit exceeded**

The number of physical or file segments exceeded the limit of 255 imposed by OS/2 protected mode and by Windows for each application or dynamic-link library.

A file segment is created for each group definition, nonpacked logical segment, and set of packed segments.

Reduce the number of segments, or put more information into each segment. Use the /PACKC option or the /PACKD option or both.

L1074 *group* **: group exceeds 64K**

The given group exceeds the limit of 65,536 bytes.

Reduce the size of the group, or remove any unneeded segments from the group. Refer to the map file for a listing of segments.

L1075 **entry table exceeds 64K – 1**

The entry table exceeded the limit of 65,535 bytes.

There is an entry in this table for each exported routine. The table also includes an entry for each address that is the target of a far relocation, when one of the following conditions is true:

- The target segment is designated **IOPL** (specific to OS/2).
- **PROTMODE** is not enabled and the target segment is designated **MOVABLE** (specific to Windows).

Declare **PROTMODE** if applicable, or reduce the number of exported routines, or make some segments **FIXED** or **NOIOPL** if possible.

L1078 **file-segment alignment too small**

The segment-alignment size specified with the /ALIGN option was too small.

L1080 **cannot open list file**

The disk or the root directory was full.

Delete or move files to make space.

L1081 **out of space for run file**

The disk on which the executable file was being written became full. Free more space on the disk and restart LINK.

L1082 *filename* **: stub file not found**

LINK could not open the file given in the **STUB** statement in the module-definition file.

The file must be in the current directory or in a directory specified by the PATH environment variable.

L1083 **cannot open run file**

One of the following may have occurred:

- The disk or the root directory was full.
- Another process opened or deleted the file.
- A read-only file existed with the same name.
- The floppy disk containing the file was removed.
- A hard-disk error occurred.

L1084 **cannot create temporary file**

One of the following may have occurred:

- The disk or the root directory was full.
- The directory specified in the TMP environment variable did not exist.

L1085 **cannot open temporary file**

One of the following may have occurred:

- The disk or the root directory was full.
- The directory specified in the TMP environment variable did not exist.

L1086 **scratch file missing**

An internal error has occurred.

Note the circumstances of the error and notify Microsoft Corporation by following the instructions on the Microsoft Product Assistance Request form at the back of one of your manuals.

L1087 **unexpected end-of-file on scratch file**

The disk with the temporary linker-output file was removed.

L1088 **out of space for list file**

The disk where the listing file was being written is full.

Free more space on the disk and restart LINK.

L1089 *filename* **: cannot open response file**

LINK could not find the specified response file.

Check that the name of the response file is spelled correctly.

L1090 **cannot reopen list file**

The original floppy disk was not replaced at the prompt.

Restart the link session.

L1091 **unexpected end-of-file on library**

The floppy disk containing the library was probably removed.

Replace the disk containing the library and run LINK again.

L1092 **cannot open module-definition file**

LINK could not open the module-definition file specified on the command line or in the response file.

L1093 *filename* **: object not found**

LINK could not find the given object file.

Check the specification of the object file.

L1094 *filename* **: cannot open file for writing**

LINK was unable to open the file with write permission.

Check file permissions.

L1095 *filename* **: out of space on file**

LINK ran out of disk space for the specified output file.

Delete or move files to make space.

L1100 **stub .EXE file invalid**

The file specified in the **STUB** statement is not a valid real-mode executable file.

L1101 **invalid object module**

One of the object modules was invalid.

Check that the correct version of LINK is being used.

If the error persists after recompiling, note the circumstances of the error and notify Microsoft Corporation by following the instructions on the Microsoft Product Assistance Request form at the back of one of your manuals.

L1102 **unexpected end-of-file**

An invalid format for a library was encountered.

L1103 **attempt to access data outside segment bounds**

A data record in an object module specified data extending beyond the end of a segment. This is a translator error.

Note which translator (compiler or assembler) produced the incorrect object module and the circumstances in which it was produced. Please report this error to Microsoft Corporation by following the instructions on the Microsoft Product Assistance Request form at the back of one of your manuals.

L1104 *filename* **: invalid library**

The specified file was not a valid library file.

L1105 **invalid object due to aborted incremental compile**

Delete the object file, recompile the program, and relink.

L1113 **unresolved COMDEF; internal error**

Note the circumstances of the error and notify Microsoft Corporation by following the instructions on the Microsoft Product Assistance Request form at the back of one of your manuals.

L1115 *option* **: option incompatible with overlays**

The given option is not compatible with overlays.

Remove the option, or do not use overlaid modules.

L1116 **/EXEPACK valid only for OS/2 and real-mode DOS**

The /EXEPACK option is incompatible with Windows programs.

L1123 *segment* **: segment defined both 16-bit and 32-bit**

Define the segment as either 16-bit or 32-bit.

L1126 **conflicting pwords value**

An exported name was specified in the module-definition file with an IOPL-parameter-words (pwords) value, and the same name was specified as an export by the Microsoft C export pragma with a different pwords value.

L1127 **far segment references not allowed with /TINY**

The /TINY option for producing a .COM file was used in a program that has a far segment reference.

Far segment references are not compatible with the .COM-file format. High-level-language programs cause this error unless the language supports the tiny memory model. An assembly-language program that references a segment address also causes this error.

For example:

```
mov     ax, seg mydata
```

F.8.2 LINK Errors

Number	LINK Error Message

L2000 **imported starting address**

The program starting address as specified in the END statement in an assembly-language file is an imported routine. This is not supported by OS/2 or Windows.

L2002 **fixup overflow at** *number* **in segment** *segment*

This error message will be followed by either

```
target external symbol
```

or

```
frm seg name1, tgt seg name2, tgt offset number
```

A fixup overflow is an attempted reference to code or data that is impossible because the source location (where the reference is made "from") and the target address (where the reference is made "to") are too far apart. Usually the problem is corrected by examining the source location.

For information about frame and target segments, see the *Microsoft MS-DOS Programmer's Reference.*

L2003 **near reference to far target at** *offset* **in segment** *segment*
pos: *offset* **target external** *name*

The program issued a near call or jump to a label in a different segment.

This error occurs most often when specifically declaring an external procedure to be near that should be declared as far.

This error can be caused by compiling a small-model C program with CL's /NT option.

L2005 **fixup type unsupported at** *number* **in segment** *segment*

A fixup type occurred that is not supported by LINK. This is probably a compiler error.

Note the circumstances of the error and notify Microsoft Corporation by following the instructions on the Microsoft Product Assistance Request form at the back of one of your manuals.

L2010 **too many fixups in LIDATA record**

The number of far relocations (pointer- or base-type) in an LIDATA record exceeds the limit imposed by LINK.

The cause is usually a **DUP** statement in an assembly-language program. The limit is dynamic: a 1,024-byte buffer is shared by relocations and the contents of the LIDATA record; there are eight bytes per relocation.

Reduce the number of far relocations in the DUP statement.

L2011 *identifier* **: NEAR/HUGE conflict**

Conflicting **NEAR** and **HUGE** attributes were given for a communal variable. This error can occur only with programs produced by the Microsoft FORTRAN Compiler or other compilers that support communal variables.

L2012 *arrayname* **: array-element size mismatch**

A far communal array was declared with two or more different array-element sizes (for instance, an array was declared once as an array of characters and once as an array of real numbers). This error occurs only with the Microsoft FORTRAN Compiler and any other compiler that supports far communal arrays.

L2013 **LIDATA record too large**

An LIDATA record contained more than 512 bytes. This is probably a compiler error.

Note the circumstances of the error and notify Microsoft Corporation by following the instructions on the Microsoft Product Assistance Request form at the back of one of your manuals.

L2022 *entry* **(alias** *internalname***) : export undefined**

The internal name of the given exported function or data item is undefined.

L2023 *entry* **(alias** *internalname***) : export imported**

The internal name of the given exported function or data item conflicts with the internal name of a previously imported function or data item.

L2024 *symbol* **: special symbol already defined**

The program defined a symbol name already used by LINK for one of its own low-level symbols. For example, LINK generates special symbols used in overlay support and other operations.

Choose another name for the symbol to avoid conflict.

L2025 *symbol* **: symbol defined more than once**

The same symbol has been found in two different object files.

L2026 **entry ordinal** *number***, name** *name* **: multiple definitions for same ordinal**

The given exported name with the given ordinal number conflicted with a different exported name previously assigned to the same ordinal. Only one name can be associated with a particular ordinal.

L2027 *name* **: ordinal too large for export**

The given exported name was assigned an ordinal that exceeded the limit of 65,535 (64K–1).

L2028 **automatic data segment plus heap exceed 64K**

The total size of data declared in DGROUP, plus the value given in **HEAPSIZE** in the module-definition file, plus the stack size given by the /STACK option or **STACKSIZE** module-definition file statement, exceeds 64K.

Reduce near-data allocation, **HEAPSIZE**, or stack.

L2029 *symbol* **: unresolved external**

A symbol was declared to be external in one or more modules, but it was not publicly defined in any module or library.

The name of the unresolved external symbol is given, then a list of object modules that contain references to this symbol. This message and the list are written to the map file, if one exists.

One cause of this error is using the /NOI option for files that use case inconsistently.

L2030 **starting address not code (use class *CODE*)**

The program starting address, as specified in the **END** statement of an .ASM file, should be in a code segment. Code segments are recognized if their class name ends in *CODE*. This is an error in OS/2 protected mode.

The error message may be disabled by including the **REALMODE** statement in the module-definition file.

L2041 **stack plus data exceed 64K**

If the total of near data and requested stack size exceeds 64K, the program will not run correctly. LINK checks for this condition only when /DOSSEG is enabled, which is the case in the library start-up module for Microsoft language libraries.

For object modules compiled with the Microsoft C or FORTRAN optimizing compilers, recompile with the /Gt command-line option to set the data-size threshold to a smaller number.

This is a fatal LINK error.

L2043 **Quick library support module missing**

The required module QUICKLIB.OBJ was missing.

The module QUICKLIB.OBJ must be linked in when creating a Quick library.

L2044 *symbol* **: symbol multiply defined, use /NOE**

LINK found what it interprets as a public-symbol redefinition, probably because a symbol defined in a library was redefined.

Relink with the /NOE option. If error L2025 results for the same symbol, then this is a genuine symbol-redefinition error.

L2045 *segment* **: segment with > 1 class name not allowed with /INCR**

The program defined a segment more than once, giving the segment different class names. This is incompatible with the /INCR option. This error appears only with assembly-language programs.

For example, the following two statements define two distinct segments with the same name but different classes:

```
_BSS segment 'BSS'

_BSS segment 'DATA'
```

L2047 **IOPL attribute conflict - segment** *segment* **in group** *group*

The specified segment is a member of the specified group but has an **IOPL** attribute that is different from other segments in the group.

L2048 **Microsoft Overlay Manager module not found**

Overlays were designated, but the Microsoft Overlay Manager module was not found. This module is defined in the default library.

L2049 **no segments defined**

No code or initialized data was defined in the program. The resulting executable file is not likely to be valid.

L2050 **USE16/USE32 attribute conflict - segment** *segment* **in group** *group*

16-bit segments cannot be grouped with 32-bit segments.

L2051 **start address not equal to 0x100 for /TINY**

The program starting address, as specified in the .COM file, must have a starting value equal to 100 hexadecimal (0x100 or 0x0). Any other value is illegal.

Put the following line of assembly source code in front of the code segment:

```
ORG 100h
```

L2052 *symbol* **: unresolved external; possible calling convention mismatch**

A symbol was declared to be external in one or more modules, but LINK could not find it publicly defined in any module or library.

The name of the unresolved external symbol is given, then a list of object modules that contain references to this symbol. The error message and the list are written to the map file, if one exists.

This error occurs in a C-language program when a prototype for an externally defined function is omitted and the program is compiled with CL's /Gr option. The calling convention for **_fastcall** does not match the assumptions that are made when a prototype is not included for an external function.

Either include a prototype for the function, or compile without the /Gr option.

F.8.3 LINK Warnings

Number	LINK Warning

L4000 **segment displacement included near** *offset* **in segment** *segment*

This is the warning generated by the /W option.

L4001 **frame-relative fixup, frame ignored near** *offset* **in segment** *segment*

A reference was made relative to a segment or group that is different from the target segment of the reference.

For example, if `_id1` is defined in segment `_TEXT`, the instruction call `DGROUP:_id1` produces this warning. The frame `DGROUP` is ignored, so LINK treats the call as if it were call `_TEXT:_id1`.

L4002 **frame-relative absolute fixup near** *offset* **in segment** *segment*

A reference was made relative to a segment or group that was different from the target segment of the reference, and both segments are absolute (defined with AT).

LINK assumed that the executable file will be run only under DOS.

L4004 **possible fixup overflow at** *offset* **in segment** *segment*

A near call or jump was made to another segment which was not a member of the same group as the segment from which the call or jump was made.

This can cause an incorrect real-mode address calculation when the distance between the paragraph address (frame number) of the segment group and the target segment is greater than 64K, even though the distance between the segment where the call or jump was actually made and the target segment is less than 64K.

L4010 **invalid alignment specification**

The number specified in the /ALIGN option must be a power of 2 in the range 2–32,768.

L4011 **/PACKC value exceeding 64K-36 unreliable**

The packing limit specified with the /PACKC option was in the range 65,501–65,536 bytes. Code segments with a size in this range are unreliable on some versions of the 80286 processor.

L4012 **/HIGH disables /EXEPACK**

The /HIGH and /EXEPACK options cannot be used at the same time.

L4013 *option* **: option ignored for segmented executable file**

The given option is not allowed with OS/2 or Windows programs.

L4014 *option* **: option ignored for DOS executable file**

The given option is not allowed with DOS programs.

L4015 **/CO disables /DSALLOC**

The /CO and /DSALLOC options cannot be used at the same time.

L4016 **/CO disables /EXEPACK**

The /CO and /EXEPACK options cannot be used at the same time.

L4017 *option* **: unrecognized option name; option ignored**

An unrecognized name followed the option indicator. LINK ignored the option specification.

An option is specified by a forward slash (/) and a name. The name can be specified by a legal abbreviation of the full name.

For example, the following command causes this warning:

```
LINK /NODEFAULTLIBSEARCH main
```

This error can also occur if the wrong version of LINK is used. Check the directories in the PATH environment variable for other versions of LINK.EXE.

L4018 **missing or unrecognized application type; option** *option* **ignored**

The /PM option accepts only the keywords **PM**, **VIO**, and **NOVIO**.

L4019 **/TINY disables /INCR**

The /TINY and /INCR options are incompatible. A .COM file always requires a full link and cannot be incrementally linked. LINK ignored /INCR.

L4020 *segment* **: code-segment size exceeds 64K-36**

Code segments in the range 65,501–65,536 bytes in length may be unreliable on some versions of the 80286 processor.

L4021 **no stack segment**

The program did not contain a stack segment defined with the STACK combine type.

Normally, every program should have a stack segment with the combine type specified as STACK. This message may be ignored if there is a specific reason for not defining a stack or for defining one without the STACK combine type. Linking with versions of LINK earlier than version 2.40 might cause this message, since these linkers search libraries only once.

L4022 *group1*, *group2* **: groups overlap**

The given groups overlap. Since a group is assigned to a physical segment, groups cannot overlap in OS/2 or Windows executable files.

Reorganize segments and group definitions so the groups do not overlap. Refer to the map file.

L4023 *entry* (*internalname*) **: export internal name conflict**

The internal name of the given exported function or data item conflicted with the internal name of a previous import definition or export definition.

L4024 *name* **: multiple definitions for export name**

The given name was exported more than once, an action that is not allowed.

L4025 *modulename.entry*(*internalname*) **: import internal name conflict**

The internal name of the given imported function or data item conflicted with the internal name of a previous export or import. (The given *entry* is either a name or an ordinal number.)

L4026 *modulename.entry*(*internalname*) **: self-imported**

The given function or data item was imported from the module being linked. This is not supported on some systems.

L4027 *name* **: multiple definitions for import internal name**

The given internal name was imported more than once. Previous import definitions are ignored.

L4028 *segment* **: segment already defined**

The given segment was defined more than once in the **SEGMENTS** statement of the module-definition file.

L4029 *segment* **: DGROUP segment converted to type DATA**

The given logical segment in the group DGROUP was defined as a code segment.

DGROUP cannot contain code segments because LINK always considers DGROUP to be a data segment. The name DGROUP is predefined as the automatic (or default) data segment.

LINK converted the named segment to type DATA.

L4030 *segment* **: segment attributes changed to conform with automatic data segment**

The given logical segment in the group DGROUP was given sharing attributes (**SHARED/NONSHARED**) that differed from the automatic data attributes as

declared by the DATA instance specification (**SINGLE/MULTIPLE**). The attributes are converted to conform to those of DGROUP.

The name DGROUP is predefined as the automatic (or default) data segment. DGROUP cannot contain code segments because LINK always considers DGROUP to be a data segment.

L4031 *segment* **: segment declared in more than one group**

A segment was declared to be a member of two different groups.

L4032 *segment* **: code-group size exceeds 64K-36**

The given code group has a size in the range 65,501–65,536 bytes, a size that is unreliable on some versions of the 80286 processor.

L4033 **first segment in mixed group** *group* **is a USE32 segment**

A 16-bit segment must be first in a group created with both **USE16** and **USE32** segments.

LINK continued to build the executable file, but the resulting file may not run correctly.

L4034 **more than 239 overlay segments; extra put in root**

The link command line or response file designated too many segments to go into overlays.

The limit on the number of segments that can go into overlays is 239. Segments starting with the 240th segment are assigned to the permanently resident portion of the program (the root).

L4036 **no automatic data segment**

The application did not define a group named DGROUP.

DGROUP has special meaning to LINK, which uses it to identify the automatic (or default) data segment used by the operating system. Most OS/2 and Windows applications require DGROUP.

This warning will not be issued if **DATA NONE** is declared or if the executable file is a dynamic-link library.

L4038 **program has no starting address**

The OS/2 or Windows application had no starting address, which will usually cause the program to fail. High-level languages automatically specify a starting address.

If you are writing an assembly-language program, specify a starting address with the END statement.

DOS programs and dynamic-link libraries should never receive this message, regardless of whether they have starting addresses.

L4040 **stack size ignored for /TINY**

LINK ignores stack size if the /TINY option is used and if the stack segment has been defined in front of the code segment.

L4042 **cannot open old version**

The file specified in the **OLD** statement in the module-definition file could not be opened.

L4043 **old version not segmented executable format**

The file specified in the **OLD** statement in the module-definition file was not a valid OS/2 or Windows executable file.

L4045 **name of output file is** *filename*

LINK used the given filename for the output file.

If the output filename is specified without an extension, LINK assumes the default extension .EXE. Creating a Quick library, DLL, or .COM file forces LINK to use a different extension:

/TINY option	.COM
/Q option	.QLB
LIBRARY statement	.DLL

L4047 **Multiple code segments in module of overlaid program incompatible with /CO**

If there are multiple code segments defined in one object file by use of the C compiler **#pragma alloc_text()** and the program is built as an overlaid program, you can access the CodeView symbolic information for only the first code segment in an overlay. Symbolic information is not accessible for other code segments in the overlay.

L4050 **file not suitable for /EXEPACK; relink without**

LINK could not pack the file because the size of the packed load image plus packing overhead was larger than that of the unpacked load image.

L4051 *filename* **: cannot find library**

LINK could not find the given library file.

One of the following may be a cause:

- The specified file does not exist. Enter the name or full path specification of a library file.
- The LIB environment variable is not set correctly. Check for incorrect directory specifications, mistyping, and a space, semicolon, or hidden character at the end of the line.
- An earlier version of LINK is being run. Check the path environment variable and delete or rename earlier linkers.

L4053 **VM.TMP : illegal filename; ignored**

VM.TMP appeared as an object-file name.

Rename the file and rerun LINK.

L4054 *filename* **: cannot find file**

LINK could not find the specified file.

Enter a new filename, a new path specification, or both.

L4067 **changing default resolution for weak external** *symbol*
from *oldresolution* **to** *newresolution*

LINK found conflicting default resolutions for a weak external. It ignored the first resolution and used the second.

L4068 **ignoring stack size greater than 64K**

A stack was defined with an invalid size. LINK assumed 64K.

L4069 **filename truncated to** *filename*

A filename specification exceeded the length allowed. LINK assumed the given filename.

L4070 **too many public symbols for sorting**

LINK uses the stack and all available memory in the near heap to sort public symbols for the /MAP option. This warning is issued if the number of public symbols exceeds the space available for them. In addition, the symbols are not sorted in the map file but are listed in an arbitrary order.

L4080 **changing substitute name for alias** *symbol* **from** *oldalias* **to** *newalias*

LINK found conflicting alias names. It ignored the first alias and used the second.

F.9 ML Error Messages

The error messages produced by the assembler fall into three categories:

- Fatal error messages
- Assembly error messages
- Warning messages

The messages for each category are listed below in numerical order, with a brief explanation of each error. To look up an error message, first determine the message category, then find the error number. All messages give the filename and line number where the error occurs.

Fatal Error Messages

Fatal error messages indicate a severe problem, one that prevents the assembler from processing your program any further. These messages have the following format:

filename (*line*) : `fatal error A1xxx`: *messagetext*

After the assembler displays a fatal-error message, it terminates without producing an object file or checking for further errors.

Assembly Error Messages

Assembly error messages identify actual program errors. There messages appear in the following format:

filename (*line*) : `error A2xxx`: *messagetext*

The assembler does not produce an object file for a source file that has assembly errors in the program. When the assembler encounters such errors, it attempts to recover from the error. If possible, it continues to process the source file and produce error messages. If errors are too numerous or too severe, the assembler stops processing.

Warning Messages

Warning messages are informational only; they do not prevent assembly and linking. These messages appear in the following format:

filename (*line*) : `warning A4xxx`: *messagetext*

F.9.1 ML Fatal Errors

Number	Message
A1000	**cannot open file:** *filename*

The assembler was unable to open a source, include, or output file.

One of the following may be a cause:

- The file does not exist.
- The file is in use by another process.
- The filename is not valid.
- A read-only file with the output filename already exists.
- Not enough file handles exist. In DOS, increase the number of file handles by changing the FILES setting in CONFIG.SYS to allow a larger number of open files. FILES=20 is the recommended setting.
- The current drive is full.
- The current directory is the root and is full.
- The device cannot be written to.
- The drive is not ready.

A1001 **I/O error closing file**

The operating system returned an error when the assembler attempted to close a file.

This error can be caused by having a corrupt file system or by removing a disk before the file could be closed.

A1002 **I/O error writing file**

The assembler was unable to write to an output file.

One of the following may be a cause:

- The current drive is full.
- The current directory is the root and is full.
- The device cannot be written to.
- The drive is not ready.

A1003 **I/O error reading file**

The assembler encountered an error when trying to read a file.

One of the following may be a cause:

- The disk has a bad sector.
- The file-access attribute is set to prevent reading.
- The drive is not ready.

A1004 **out of far memory, use /VM command-line option**

There was insufficient memory to assemble the program.

One of the following may be a solution:

- In DOS, use the /VM command-line option to enable virtual memory.
- If you are using the NMAKE utility, try using NMK or assembling outside of NMAKE.
- In PWB, try exiting and assembling using ML.
- In OS/2, try increasing the swap space.
- In DOS, remove terminate-and-stay-resident (TSR) software.
- Change CONFIG.SYS to specify a lower number of buffers (the BUFFERS= command) and fewer drives (the LASTDRIVE= command).
- Eliminate unnecessary **INCLUDE** directives.

A1005 **assembler limit : macro parameter name table full**

Too many parameters, locals, or macro labels were defined for a macro. There was no more room in the macro name table.

Define shorter or fewer names, or remove unnecessary macros.

A1006 **invalid command-line option:** *option*

ML did not recognize the given parameter as an option.

A1007 **nesting level too deep**

The assembler reached its nesting limit. The limit is 20 levels except where noted otherwise.

One of the following was nested too deeply:

- A high-level directive such as **.IF**, **.REPEAT**, or **.WHILE**
- A structure definition
- A conditional-assembly directive
- A procedure definition
- A **PUSHCONTEXT** directive (The limit is 10.)
- A segment definition
- An include file
- A macro

A1008 **unmatched macro nesting**

Either a macro was not terminated before the end of the file, or the terminating directive **ENDM** was found outside of a macro block.

One cause of this error is omission of the dot before **.REPEAT** or **.WHILE**.

A1009 **line too long**

A line in a source file exceeded the limit of 512 characters.

If multiple physical lines are concatenated with the line-continuation character (\), the resulting logical line is still limited to 512 characters.

A1010 **unmatched block nesting :**

A block beginning did not have a matching end, or a block end did not have a matching beginning. One of the following may be involved:

- A high-level directive such as **.IF**, **.REPEAT**, or **.WHILE**
- A conditional-assembly directive such as **IF, REPEAT, or WHILE**
- A structure or union definition
- A procedure definition

- A segment definition
- A **POPCONTEXT** directive
- A conditional-assembly directive, such as an **ELSE**, **ELSEIF**, or **ENDIF** without a matching **IF**

A1011 **directive must be in control block**

The assembler found a high-level directive where one was not expected. One of the following directives was found:

- **.ELSE** without **.IF**
- **.ENDIF** without **.IF**
- **.ENDW** without **.WHILE**
- **.UNTIL[[CXZ]]** without **.REPEAT**
- **.CONTINUE** without **.WHILE** or **.REPEAT**
- **.BREAK** without **.WHILE** or **.REPEAT**
- **.ELSE** following **.ELSE**

A1012 **error count exceeds 100; stopping assembly**

The number of nonfatal errors exceeded the assembler limit of 100.

Nonfatal errors are in the range A2xxx. When warnings are treated as errors they are included in the count. Warnings are considered errors if you use the /Wx command-line option, or if you set the Warnings Treated as Errors option in the Macro Assembler Global Options dialog box of PWB.

A1013 **invalid numerical command-line argument :** *number*

The argument specified with an option was not a number or was an invalid number.

A1014 **too many arguments**

There was insufficient memory to hold all of the command-line arguments.

This error usually occurs while expanding input filename wildcards (* and ?). To eliminate this error, assemble multiple source files separately.

A1015 **statement too complex**

The assembler ran out of stack space while trying to parse the specified statement.

One or more of the following changes may eliminate this error:

- Break the statement into several shorter statements.
- Reorganize the statement to reduce the amount of parenthetical nesting.
- If the statement is part of a macro, break the macro into several shorter macros.

A1016 **out of virtual memory**

The assembler was unable to allocate enough virtual memory to assemble this file.

To eliminate this error, free some space on the drive specified by the TMP environment variable, or reassign TMP to a location where there is more free space. The assembler uses the current directory to store VM files if the TMP environment varible does not exist.

A1017 **out of near memory**

There was insufficient memory to assemble the program.

One of the following may be a solution:

- If you are using the NMAKE utility, try using NMK or assembling outside of NMAKE.
- In PWB, try exiting and assembling using ML.
- In OS/2, try increasing the swap space.
- In DOS, remove terminate-and-stay-resident (TSR) software.
- Change CONFIG.SYS to specify a lower number of buffers (the BUFFERS= command) and fewer drives (the LASTDRIVE= command).
- Eliminate unnecessary **INCLUDE** directives.

A1018 **missing source filename**

ML could not find a file to assemble or pass to the linker.

This error is generated when you give ML command-line options without specifying a filename to act upon. To assemble files that do not have a .ASM extension, use the /Ta command-line option.

This error can also be generated by invoking ML with no parameters if the ML environment variable contains command-line options.

A1901 **Internal Assembler Error**
Contact Microsoft Product Support Services

The MASM driver called ML.EXE, which generated a system error.

Note the circumstances of the error and notify Microsoft Corporation by following the instructions on the Microsoft Product Assistance Request form at the back of one of your manuals.

F.9.2 ML Errors

Number	Message

A2000 **memory operand not allowed in context**

A memory operand was given to an instruction that cannot take a memory operand.

A2001 **immediate operand not allowed**

A constant or memory offset was given to an instruction that cannot take an immediate operand.

A2002 **cannot have more than one ELSE clause per IF block**

The assembler found an **ELSE** directive after an existing **ELSE** directive in a conditional-assembly block (**IF** block).

Only one **ELSE** can be used in an **IF** block. An **IF** block begins with an **IF**, **IFE**, **IFB**, **IFNB**, **IFDEF**, **IFNDEF**, **IFDIF**, or **IFIDN** directive. There can be several **ELSEIF** statements in an **IF** block.

One cause of this error is omission of an **ENDIF** statement from a nested **IF** block.

A2003 **extra characters after statement**

A directive was followed by unexpected characters.

A2004 **symbol type conflict :** *identifier*

The **EXTERNDEF** or **LABEL** directive was used on a variable, symbol, data structure, or label that was defined in the same module but with a different type.

A2005 **symbol redefinition :** *identifier*

The given nonredefinable symbol was defined in two places.

A2006 **undefined symbol : *identifier***

An attempt was made to use a symbol that was not defined.

One of the following may have occurred:

- A symbol was not defined.
- A field was not a member of the specified structure.
- A symbol was defined in an include file that was not included.
- An external symbol was used without an **EXTERN** or **EXTERNDEF** directive.
- A symbol name was misspelled.
- A local code label was referenced outside of its scope.

A2008 **syntax error :**

A token at the current location caused a syntax error.

One of the following may have occurred:

- A dot prefix was added to or omitted from a directive.
- A reserved word (such as **C** or **SIZE**) was used as an identifier.
- An instruction was used that was not available with the current processor or coprocessor selection.
- A comparison run-time operator (such as ==) was used in a conditional assembly statement instead of a relational operator (such as **EQ**).
- An instruction or directive was given too few operands.
- An obsolete directive was used.

A2009 **syntax error in expression**

An expression on the current line contained a syntax error. This error message may also be a side-effect of a preceding program error.

A2010 **invalid type expression**

The operand to **THIS** or **PTR** was not a valid type expression.

A2011 **distance invalid for word size of current segment**

A procedure definition or a code label defined with **LABEL** specified an address size that was incompatible with the current segment size.

One of the following occurred:

- A **NEAR16** or **FAR16** procedure was defined in a 32-bit segment.
- A **NEAR32** or **FAR32** procedure was defined in a 16-bit segment.
- A code label defined with **LABEL**specified **FAR16** or **NEAR16** in a 32-bit segment.
- A code label defined with **LABEL** specified **FAR32** or **NEAR32** in a 16-bit segment.

A2012 **PROC, MACRO, or macro repeat directive must precede LOCAL**

A **LOCAL** directive must be immediately preceded by a **MACRO**, **PROC**, macro repeat directive (such as **REPEAT**, **WHILE**, or **FOR**), or another **LOCAL** directive.

A2013 **.MODEL must precede this directive**

A simplified segment directive or a **.STARTUP** or **.EXIT** directive was not preceded by a **.MODEL** directive.

A **.MODEL** directive must specify the model defaults before a simplified segment directive, or a **.STARTUP** or **.EXIT** directive may be used.

A2014 **cannot define as public or external :** *identifier*

Only labels, procedures, and numeric equates can be made public or external using **PUBLIC**, **EXTERN**, or **EXTERNDEF**. Local code labels cannot be made public.

A2015 **segment attributes cannot change :** *attribute*

A segment was reopened with different attributes than it was opened with originally.

When a **SEGMENT** directive opens a previously defined segment, the newly opened segment inherits the attributes the segment was defined with.

A2016 **expression expected**

The assembler expected an expression at the current location but found one of the following:

- A unary operator without an operand
- A binary operator without two operands
- An empty pair of parentheses, ( ), or brackets, [ ]

A2017 **operator expected**

An expression operator was expected at the current location.

One possible cause of this error is a missing comma between expressions in an expression list.

A2018 **invalid use of external symbol : *identifier***

An attempt was made to compare the given external symbol using a relational operator.

The comparison cannot be made because the value or address of an external symbol is not known at assembly time.

A2019 **operand must be RECORD type or field**

The operand following the **WIDTH** or **MASK** operator was not valid.

The **WIDTH** operator takes an operand that is the name of a field or a record. The **MASK** operator takes an operand that is the name of a field or a record type.

A2020 **identifier not a record : *identifier***

A record type was expected at the current location.

A2021 **record constants cannot span line breaks**

A record constant must be defined on one physical line. A line ended in the middle of the definition of a record constant.

A2022 **instruction operands must be the same size**

The operands to an instruction did not have the same size.

A2023 **instruction operand must have size**

At least one of the operands to an instruction must have a known size.

A2024 **invalid operand size for instruction**

The size of an operand was not valid.

A2025 **operands must be in same segment**

Relocatable operands used with a relational or minus operator were not located in the same segment.

A2026 **constant expected**

The assembler expected a constant expression at the current location. A constant expression is a numeric expression that can be resolved at assembly time.

A2027 **operand must be a memory expression**

The right operand of a **PTR** expression was not a memory expression.

When the left operand of the **PTR** operator is a structure or union type, the right operand must be a memory expression.

A2028 **expression must be a code address**

An expression evaluating to a code address was expected.

One of the following occurred:

- **SHORT** was not followed by a code address.
- **NEAR PTR** or **FAR PTR** was applied to something that was not a code address.

A2029 **multiple base registers not allowed**

An attempt was made to combine two base registers in a memory expression.

For example, the following expressions cause this error:

```
[bx+bp]
```

```
[bx][bp]
```

In another example, given the following definition:

```
id1 proc arg1:byte
```

either of the following lines causes this error:

```
mov al, [bx].arg1
```

```
lea ax, arg1[bx]
```

A2030 **multiple index registers not allowed**

An attempt was made to combine two index registers in a memory expression.

For example, the following expressions cause this error:

```
[si+di]
```

```
[di][si]
```

A2031 **must be index or base register**

An attempt was made to use a register that was not a base or index register in a memory expression.

For example, the following expressions cause this error:

```
[ax]
```

```
[bl]
```

A2032 **invalid use of register**

An attempt was made to use a register that was not valid for the intended use.

One of the following occurred:

- **OFFSET** was applied to a register. (**OFFSET** can be applied to a register under the **M510** option.)
- A special 386 register was used in an invalid context.
- A register was cast with **PTR** to a type of invalid size.
- A register was specified as the right operand of a segment override operator (:).
- A register was specified as the right operand of a binary minus operator (-).
- An attempt was made to multiply registers using the * operator.
- Brackets ([ ]) were missing around a register that was added to something.

A2033 **invalid INVOKE argument : argument** *number*

The **INVOKE** directive was passed a special 386 register, or a register pair containing a byte register or special 386 register. These registers are illegal with **INVOKE**.

A2034 **must be in segment block**

One of the following was found outside of a segment block:

- An instruction
- A label definition
- A **THIS** operator
- A $ operator
- A procedure definition
- An **ALIGN** directive
- An **ORG** directive

A2035 **DUP too complex**

A declaration using the **DUP** operator resulted in a data structure with an internal representation that was too large.

A2036 **too many initial values for structure:** *structure*

The given structure was defined with more initializers than the number of fields in the type declaration of the structure.

A2037 **statement not allowed inside structure definition**

A structure definition contained an invalid statement.

A structure cannot contain instructions, labels, procedures, control-flow directives, **.STARTUP**, or **.EXIT**.

A2038 **missing operand for macro operator**

The assembler found the end of a macro's parameter list immediately after the **!** or **%** operator.

A2039 **line too long**

A source-file line exceeded the limit of 512 characters.

If multiple physical lines are concatenated with the line- continuation character (\), the resulting logical line is still limited to 512 characters.

A2040 **segment register not allowed in context**

A segment register was specified for an instruction that cannot take a segment register.

A2041 **string or text literal too long**

A string or text literal, or a macro function return value, exceeded the limit of 255 characters.

A2042 **statement too complex**

A statement was too complex for the assembler to parse.

Reduce either the number of tokens or the number of forward-referenced identifiers.

A2043 **identifier too long**

An identifier exceeded the limit of 247 characters.

A2044 **invalid character in file**

The source file contained a character outside a comment, string, or literal that was not recognized as an operator or other legal character.

A2045 **missing angle bracket or brace in literal**

An unmatched angle bracket (either < or >) or brace (either { or }) was found in a literal constant or an initializer.

One of the following occurred:

- A pair of angle brackets or braces was not complete.
- An angle bracket was intended to be literal, but it was not preceded by an exclamation point (!) to indicate a literal character.

A2046 **missing single or double quotation mark in string**

An unmatched quotation mark (either ' or ") was found in a string.

One of the following may have occurred:

- A pair of quotation marks around a string was not complete.
- A pair of quotation marks around a string was formed of one single and one double quotation mark.
- A single or double quotation mark was intended to be literal, but the surrounding quotation marks were the same kind as the literal one.

A2047 **empty (null) string**

A string consisted of a delimiting pair of quotation marks and no characters within.

For a string to be valid, it must contain 1–255 characters.

A2048 **nondigit in number**

A number contained a character that was not in the set of characters used by the current radix (base).

This error can occur if a B or D radix specifier is used when the default radix is one that includes that letter as a valid digit.

A2049 **syntax error in floating-point constant**

A floating-point constant contained an invalid character.

A2050 **real or BCD number not allowed**

A floating-point (real) number or binary coded decimal (BCD) constant was used other than as a data initializer.

One of the following occurred:

- A real number or a BCD was used in an expression.
- A real number was used to initialize a directive other than **DWORD**, **QWORD**, or **TBYTE**.
- A BCD was used to initialize a directive other than **TBYTE**.

A2051 **text item required**

A literal constant or text macro was expected.

One of the following was expected:

- A literal constant, which is text enclosed in < >
- A text macro name
- A macro function call
- A % followed by a constant expression

A2052 **forced error**

The conditional-error directive **.ERR** or **.ERR1** was used to generate this error.

A2053 **forced error : value equal to 0**

The conditional-error directive **.ERRE** was used to generate this error.

A2054 **forced error : value not equal to 0**

The conditional-error directive **.ERRNZ** was used to generate this error.

A2055 **forced error : symbol not defined**

The conditional-error directive **.ERRNDEF** was used to generate this error.

A2056 **forced error : symbol defined**

The conditional-error directive **.ERRDEF** was used to generate this error.

A2057 **forced error : string blank**

The conditional-error directive **.ERRB** was used to generate this error.

A2058 **forced error : string not blank**

The conditional-error directive **.ERRNB** was used to generate this error.

A2059 **forced error : strings equal**

The conditional-error directive **.ERRIDN** or **.ERRIDNI** was used to generate this error.

A2060 **forced error : strings not equal**

The conditional-error directive **.ERRDIF** or **.ERRDIFI** was used to generate this error.

A2061 **[[ELSE]]IF2/.ERR2 not allowed : single-pass assembler**

A directive for a two-pass assembler was found.

The Microsoft Macro Assembler (MASM) is a one-pass assembler. MASM does not accept the **IF2**, **ELSEIF2**, and **.ERR2** directives.

This error also occurs if an **ELSE** directive follows an **IF1** directive.

A2062 **expression too complex for .UNTILCXZ**

An expression used in the condition that follows **.UNTILCXZ** was too complex.

The **.UNTILCXZ** directive can take only one expression, which can contain only == or !=. It cannot take other comparison operators or more complex expressions using operators such as ||.

A2063 **can ALIGN only to power of 2 :** *expression*

The expression specified with the **ALIGN** directive was invalid.

The **ALIGN** expression must be a power of 2 between 2 and 256, and must be less than or equal to the alignment of the current segment, structure, or union.

A2064 **structure alignment must be 1, 2, or 4**

The alignment specified in a structure definition was invalid.

A2065 **expected :** *token*

The assembler expected the given token.

A2066 **incompatible CPU mode and segment size**

An attempt was made to open a segment with a **USE16**, **USE32**, or **FLAT** attribute that was not compatible with the specified CPU, or to change to a 16-bit CPU while in a 32-bit segment.

The **USE32** and **FLAT** attributes must be preceded by one of the following processor directives: **.386**, **.386C**, **.386P**, **.486**, or **.486P**.

A2067 **LOCK must be followed by a memory operation**

The **LOCK** prefix preceded an invalid instruction. No instruction can take the **LOCK** prefix unless one of its operands is a memory expression.

A2068 **instruction prefix not allowed**

One of the prefixes **REP**, **REPE**, **REPNE**, or **LOCK** preceded an instruction for which it was not valid.

A2069 **no operands allowed for this instruction**

One or more operands were specified with an instruction that takes no operands.

A2070 **invalid instruction operands**

One or more operands were not valid for the instruction they were specified with.

A2071 **initializer too large for specified size**

An initializer value was too large for the data area it was initializing.

A2072 **cannot access symbol in given segment or group:** *identifier*

The given identifier cannot be addressed from the segment or group specified.

A2073 **operands have different frames**

Two operands in an expression were in different frames.

Subtraction of pointers requires the pointers to be in the same frame. Subtraction of two expressions that have different effective frames is not allowed. An effective frame is calculated from the segment, group, or segment register.

A2074 **cannot access label through segment registers**

An attempt was made to access a label through a segment register that was not assumed to its segment or group.

A2075 **jump destination too far [: by 'n' bytes]**

The destination specified with a jump instruction was too far from the instruction.

One of the following may be a solution:

- Enable the **LJMP** option.
- Remove the **SHORT** operator. If **SHORT** has forced a jump that is too far, *n* is the number of bytes out of range.
- Rearrange code so that the jump is no longer out of range.

A2076 **jump destination must specify a label**

A direct jump's destination must be relative to a code label.

A2077 **instruction does not allow NEAR indirect addressing**

A conditional jump or loop cannot take a memory operand. It must be given a relative address or label.

A2078 **instruction does not allow FAR indirect addressing**

A conditional jump or loop cannot take a memory operand. It must be given a relative address or label.

A2079 **instruction does not allow FAR direct addressing**

A conditional jump or loop cannot be to a different segment or group.

A2080 **jump distance not possible in current CPU mode**

A distance was specified with a jump instruction that was incompatible with the current processor mode.

For example, 48-bit jumps require **.386** or above.

A2081 **missing operand after unary operator**

An operator required an operand, but no operand followed.

A2082 **cannot mix 16- and 32-bit registers**

An address expression contained both 16- and 32-bit registers.

For example, the following expression causes this error:

```
[bx+edi]
```

A2083 **invalid scale value**

A register scale was specified that was not 1, 2, 4, or 8.

A2084 **constant value too large**

A constant was specified that was too big for the context in which it was used.

A2085 **instruction or register not accepted in current CPU mode**

An attempt was made to use an instruction, register, or keyword that was not valid for the current processor mode.

For example, 32-bit registers require **.386** or above. Control registers such as CR0 require privileged mode **.386P** or above. This error will also be generated for the **NEAR32**, **FAR32**, and **FLAT** keywords, which require .386 or above.

A2086 **reserved word expected**

One or more items in the list specified with a **NOKEYWORD** option were not recognized as reserved words.

A2087 **instruction form requires 80386/486**

An instruction was used that was not compatible with the current processor mode.

One of the following processor directives must precede the instruction: **.386**, **.386C**, **.386P**, **.486**, or **.486P**.

A2088 **END directive required at end of file**

The assembler reached the end of the main source file and did not find an **.END** directive.

A2089 **too many bits in RECORD :** *identifier*

One of the following occurred:

- Too many bits were defined for the given record field.
- Too many total bits were defined for the given record.

The size limit for a record or a field in a record is 16 bits when doing 16-bit arithmetic or 32 bits when doing 32-bit arithmetic.

A2090 **positive value expected**

A positive value was not found in one of the following situations:

- The starting position specified for **SUBSTR** or **@SubStr**
- The number of data objects specified for **COMM**
- The element size specified for **COMM**

A2091 **index value past end of string**

An index value exceeded the length of the string it referred to when used with **INSTR**, **SUBSTR**, **@InStr**, or **@SubStr**.

A2092 **count must be positive or zero**

The operand specified to the **SUBSTR** directive, **@SubStr** macro function, **SHL** operator, **SHR** operator, or **DUP** operator was negative.

A2093 **count value too large**

The length argument specified for **SUBSTR** or **@SubStr** exceeded the length of the specified string.

A2094 **operand must be relocatable**

An operand was not relative to a label.

One of the following occurred:

- An operand specified with the **END** directive was not relative to a label.
- An operand to the **SEG** operator was not relative to a label.
- The right operand to the minus operator was relative to a label, but the left operand was not.
- The operands to a relational operator were either not both integer constants or not both memory operands. Relational operators can take operands that are both addresses or both non-addresses but not one of each.

A2095 **constant or relocatable label expected**

The operand specified must be a constant expression or a memory offset.

A2096 **segment, group, or segment register expected**

A segment or group was expected but was not found.

One of the following occurred:

- The left operand specified with the segment override operator (:) was not a segment register (CS, DS, SS, ES, FS, or GS), group name, segment name, or segment expression.
- The **ASSUME** directive was given a segment register without a valid segment address, segment register, group, or the special **FLAT** group.

A2097 **segment expected : *identifier***

The **GROUP** directive was given an identifier that was not a defined segment.

A2098 **invalid operand for OFFSET**

The expression following the **OFFSET** operator must be a memory expression or an immediate expression.

A2099 **invalid use of external absolute**

An attempt was made to subtract a constant defined in another module from an expression.

You can avoid this error by placing constants in include files rather than making them external.

A2100 **segment or group not allowed**

An attempt was made to use a segment or group in a way that was not valid. Segments or groups cannot be added.

A2101 **cannot add two relocatable labels**

An attempt was made to add two expressions that were both relative to a label.

A2102 **cannot add memory expression and code label**

An attempt was made to add a code label to a memory expression.

A2103 **segment exceeds 64K limit**

A 16-bit segment exceeded the size limit of 64K.

A2104 **invalid type for data declaration : *type***

The given type was not valid for a data declaration.

A2105 **HIGH and LOW require immediate operands**

The operand specified with either the **HIGH** or the **LOW** operator was not an immediate expression.

A2107 **cannot have implicit far jump or call to near label**

An attempt was made to make an implicit far jump or call to a near label in another segment.

A2108 **use of register assumed to ERROR**

An attempt was made to use a register that had been assumed to ERROR with the **ASSUME** directive.

A2109 **only white space or comment can follow backslash**

A character other than a semicolon (;) or a white-space character (spaces or TAB characters) was found after a line-continuation character (\).

A2110 **COMMENT delimiter expected**

A delimiter character was not specified for a **COMMENT** directive.

The delimiter character is specified by the first character that is not white space (spaces or TAB characters) after the **COMMENT** directive. The comment consists of all text following the delimiter until the end of the line containing the next appearance of the delimiter.

A2111 **conflicting parameter definition**

A procedure defined with the **PROC** directive did not match its prototype as defined with the **PROTO** directive.

A2112 **PROC and prototype calling conventions conflict**

A procedure was defined in a prototype (using the **PROTO**, **EXTERNDEF**, or **EXTERN** directive), but the calling convention did not match the corresponding **PROC** directive.

A2113 **invalid radix tag**

The specified radix was not a number in the range 2–16.

A2114 **INVOKE argument type mismatch : argument** *number*

The type of the arguments passed using the **INVOKE** directive did not match the type of the parameters in the prototype of the procedure being invoked.

A2115 **invalid coprocessor register**

The coprocessor index specified was negative or greater than 7.

A2116 **instructions and initialized data not allowed in AT segments**

An instruction or initialized data was found in a segment defined with the **AT** attribute.

Data in **AT** segments must be declared with the **?** initializer.

A2117 **/AT option requires TINY memory model**

The /AT option was specified on the assembler command line, but the program being assembled did not specify the **TINY** memory model with the **.MODEL** directive.

This error is only generated for modules that specify a start address or use the **.STARTUP** directive.

A2118 **cannot have segment address references with TINY model**

An attempt was made to reference a segment in a **TINY** model program.

All **TINY** model code and data must be accessed with **NEAR** addresses.

A2119 **language type must be specified**

A procedure definition or prototype was not given a language type.

A language type must be declared in each procedure definition or prototype if a default language type is not specified. A default language type is set using either the **.MODEL** directive, **OPTION LANG**, or the ML command-line options /Gc or /Gd.

A2120 **PROLOGUE must be macro function**

The identifier specified with the **OPTION PROLOGUE** directive was not recognized as a defined macro function.

The user-defined prologue must be a macro function that returns the number of bytes needed for local varaiables and any extra space needed for the macro function.

A2121 **EPILOGUE must be macro procedure**

The identifier specified with the **OPTION EPILOGUE** directive was not recognized as a defined macro procedure.

The user-defined epilogue macro cannot return a value.

A2122 **alternate identifier not allowed with EXTERNDEF**

An attempt was made to specify an alternate identifier with an **EXTERNDEF** directive.

You can specify an optional alternate identifier with the **EXTERN** directive but not with **EXTERNDEF**.

A2123 **text macro nesting level too deep**

A text macro was nested too deeply. The nesting limit for text macros is 40.

A2125 **missing macro argument**

A required argument to **@InStr**, **@SubStr**, or a user-defined macro was not specified.

A2126 **EXITM used inconsistently**

The **EXITM** directive was used both with and without a return value in the same macro.

A macro procedure returns a value; a macro function does not.

A2127 **macro function argument list too long**

There were too many characters in a macro function's argument list. This error applies also to a prologue macro function called implicitly by the **PROC** directive.

A2129 **VARARG parameter must be last parameter**

A parameter other than the last one was given the **VARARG** attribute.

The **:VARARG** specification can be applied only to the last parameter in a parameter list for macro and procedure definitions and prototypes. You cannot use multiple **:VARARG** specifications in a macro.

A2130 **VARARG parameter not allowed with LOCAL**

An attempt was made to specify **:VARARG** as the type in a procedure's **LOCAL** declaration.

A2131 **VARARG parameter requires C calling convention**

A **VARARG** parameter was specified in a procedure definition or prototype, but the **C**, **SYSCALL**, or **STDCALL** calling convention was not specified.

A2132 **ORG needs a constant or local offset**

The expression specified with the **ORG** directive was not valid.

ORG requires an immediate expression with no reference to an external label or to a label outside the current segment.

A2133 **register value overwritten by INVOKE**

A register was passed as an argument to a procedure, but the code generated by **INVOKE** to pass other arguments destroyed the contents of the register.

The AX, AL, AH, EAX, DX, DL, DH, and EDX registers may be used by the assembler to perform data conversion.

Use a different register.

A2134 **structure too large to pass with INVOKE : argument** *number*

An attempt was made with **INVOKE** to pass a structure that exceeded 255 bytes.

Pass structures by reference if they are larger than 255 bytes.

A2136 **too many arguments to INVOKE**

The number of arguments passed using the **INVOKE** directive exceeded the number of parameters in the prototype for the procedure being invoked.

A2137 **too few arguments to INVOKE**

The number of arguments passed using the **INVOKE** directive was fewer than the number of required parameters specified in the prototype for the procedure being invoked.

A2138 **invalid data initializer**

The initializer list for a data definition was invalid.

This error can be caused by using the R radix override with too few digits.

A2140 **RET operand too large**

The operand specified to **RET**, **RETN**, or **RETF** exceeded two bytes.

A2141 **too many operands to instruction**

Too many operands were specified with a string control instruction.

A2142 **cannot have more than one .ELSE clause per .IF block**

The assembler found more than one **.ELSE** clause within the current **.IF** block.

Use **.ELSEIF** for all but the last block.

A2143 **expected data label**

The **LENGTHOF**, **SIZEOF**, **LENGTH**, or **SIZE** operator was applied to a non-data label, or the **SIZEOF** or **SIZE** operator was applied to a type.

A2144 **cannot nest procedures**

An attempt was made to nest a procedure containing a parameter, local variable, **USES** clause, or a statement that generated a new segment or group.

A2145 **EXPORT must be FAR :** *procedure*

The given procedure was given **EXPORT** visibility and **NEAR** distance.

All **EXPORT** procedures must be **FAR**. The default visibility may have been set with the **OPTION PROC:EXPORT** statement or the **SMALL** or **COMPACT** memory models.

A2146 **procedure declared with two visibility attributes :** *procedure*

The given procedure was given conflicting visibilities.

A procedure was declared with two different visibilities (**PUBLIC**, **PRIVATE**, or **EXPORT**). The **PROC** and **PROTO** statements for a procedure must have the same visibility.

A2147 **macro label not defined :** *macrolabel*

The given macro label was not found.

A macro label is defined with **:***macrolabel.*

A2148 **invalid symbol type in expression :** *identifier*

The given identifier was used in an expression in which it was not valid.

For example, a macro procedure name is not allowed in an expression.

A2149 **byte register cannot be first operand**

A byte register was specified to an instruction that cannot take it as the first operand.

A2150 **word register cannot be first operand**

A word register was specified to an instruction that cannot take it as the first operand.

A2151 **special register cannot be first operand**

A special register was specified to an instruction that cannot take it as the first operand.

A2152 **coprocessor register cannot be first operand**

A coprocessor (stack) register was specified to an instruction that cannot take it as the first operand.

A2153 **cannot change size of expression computations**

An attempt was made to set the expression word size when the size had been already set using the **EXPR16**, **EXPR32**, **SEGMENT:USE32**, or **SEGMENT:FLAT** option or the **.386** or higher processor selection directive.

A2154 **syntax error in control-flow directive**

The condition for a control-flow directive (such as **.IF** or **.WHILE**) contained a syntax error.

A2155 **cannot use 16-bit register with a 32-bit address**

An attempt was made to mix 16-bit and 32-bit offsets in an expression.

Use a 32-bit register with a symbol defined in a 32-bit segment.

For example, if `id1` is defined in a 32-bit segment, the following causes this error:

```
id1[bx]
```

A2156 **constant value out of range**

An invalid value was specified for the **PAGE** directive.

The first parameter of the **PAGE** directive can be either 0 or a value in the range 10–255. The second parameter of the **PAGE** directive can be either 0 or a value in the range 60–255.

A2157 **missing right parenthesis**

A right parenthesis,), was missing from a macro function call.

Be sure that parentheses are in pairs if nested.

A2158 **type is wrong size for register**

An attempt was made to assume a general-purpose register to a type with a different size than the register.

For example, the following pair of statements causes this error:

```
ASSUME bx:far ptr byte ; far pointer is 4 or 6 bytes
ASSUME al:word         ; al is a byte reg, cannot hold word
```

A2159 **structure cannot be instanced**

An attempt was made to create an instance of a structure when there were no fields or data defined in the structure definition or when **ORG** was used in the structure definition.

A2160 **non-benign structure redefinition : label incorrect**

A label given in a structure redefinition either did not exist in the original definition or was out of order in the redefinition.

A2161 **non-benign structure redefinition : too few labels**

Not enough members were defined in a structure redefinition.

A2162 **OLDSTRUCT/NOOLDSTRUCT state cannot be changed**

Once the **OLDSTRUCTS** or **NOOLDSTRUCTS** option has been specified and a structure has been defined, the structure scoping cannot be altered or respecified in the same module.

A2163 **non-benign structure redefinition : incorrect initializers**

A **STRUCT** or **UNION** was redefined with a different initializer value.

When structures and unions are defined more than once, the definitions must be identical. This error can be caused by using a variable as an initializer and having the value of the variable change between definitions.

A2164 **non-benign structure redefinition : too few initializers**

A **STRUCT** or **UNION** was redefined with too few initializers.

When structures and unions are defined more than once, the definitions must be identical.

A2165 **non-benign structure redefinition : label has incorrect offset**

The offset of a label in a redefined **STRUCT** or **UNION** differs from the original definition.

When structures and unions are defined more than once, the definitions must be identical. This error can be caused by a missing member or by a member that has a different size than in its original definition.

A2166 **structure field expected**

The right-hand side of a dot operator (.) is not a structure field.

This error may occur with some code acceptable to previous versions of the assembler. To enable the old behavior, use **OPTION OLDSTRUCTS**, which is automatically enabled by **OPTION M510** or the /Zm command-line option.

A2167 **unexpected literal found in expression**

A literal was found where an expression was expected.

One of the following may have occurred:

- A literal was used as an initializer
- A record tag was omitted from a record constant

A2169 **divide by zero in expression**

An expression contains a divisor whose value is equal to zero.

Check that the syntax of the expression is correct and that the divisor (whether constant or variable) is correctly initialized.

A2170 **directive must appear inside a macro**

A **GOTO** or **EXITM** directive was found outside the body of a macro.

A2171 **cannot expand macro function**

A syntax error prevented the assembler from expanding the macro function.

A2172 **too few bits in RECORD**

There was an attempt to define a record field of 0 bits.

A2173 **macro function cannot redefine itself**

There was an attempt to define a macro function inside the body of a macro function with the same name. This error can also occur when a member of a chain of macros attempts to redefine a previous member of the chain.

A2175 **invalid qualified type**

An identifier was encountered in a qualified type that was not a type, structure, record, union, or prototype.

A2176 **floating point initializer on an integer variable**

An attempt was made to use a floating-point initializer with **DWORD**,**QWORD**, or **TBYTE**. Only integer initializers are allowed.

A2177 **nested structure improperly initialized**

The nested structure initialization could not be resolved.

This error can be caused by using different beginning and ending delimiters in a nested structure initialization.

A2178 **invalid use of FLAT**

There was an ambiguous reference to **FLAT** as a group.

This error is generated when there is a reference to **FLAT** instead of a **FLAT** subgroup. For example,

```
mov    ax, FLAT              ; Generates A2178
mov    ax, SEG FLAT:_data    ; Correct
```

A2179 **structure improperly initialized**

There was an error in a structure initializer.

One of the following occurred:

- The initializer is not a valid expression.
- The initializer is an invalid **DUP** statement.

A2180 **improper list initialization**

In a structure, there was an attempt to initialize a list of items with a value or list of values of the wrong size.

A2181 **initializer must be a string or single item**

There was an attempt to initialize a structure element with something other than a single item or string.

This error can be caused by omitting braces ({ }) around an initializer.

A2182 **initializer must be a single item**

There was an attempt to initialize a structure element with something other than a single item.

This error can be caused by omitting braces ({ }) around an initializer.

A2183 **initializer must be a single byte**

There was an attempt to initialize a structure element of byte size with something other than a single byte.

A2184 **improper use of list initializer**

The assembler did not expect an opening brace ({) at this point.

A2185 **improper literal initialization**

A literal structure initializer was not properly delimited.

This error can be caused by missing angle brackets (< >) or braces ({ }) around an initializer or by extra characters after the end of an initializer.

A2186 **extra characters in literal initialization**

A literal structure initializer was not properly delimited.

One of the following may have occurred:

- There were missing or mismatched angle brackets (< >) or braces ({ }) around an initializer.
- There were extra characters after the end of an initializer.
- There was a syntax error in the structure initialization.

A2187 **must use floating point initializer**

A variable declared with the **REAL4**, **REAL8**, and **REAL10** directives must be initialized with a floating-point number or a question mark (**?**).

This error can be caused by giving an initializer in integer form (such as 18) instead of in floating-point form (18.0).

A2188 **cannot use .EXIT for OS_OS2 with .8086**

The **INVOKE** generated by the **.EXIT** statement under **OS_OS2** requires the **.186** (or higher) directive, since it must be able to use the **PUSH** instruction to push immediates directly.

A2189 **invalid combination with segment alignment**

The alignment specified by the **ALIGN** or **EVEN** directive was greater than the current segment alignment as specified by the **SEGMENT** directive.

A2190 **INVOKE requires prototype for procedure**

The **INVOKE** directive must be preceded by a **PROTO** statement for the procedure being called.

When using **INVOKE** with an address rather than an explicit procedure name, you must precede the address with a pointer to the prototype.

A2191 **cannot include structure in self**

You cannot reference a structure recursively (inside its own definition).

A2192 **symbol language attribute conflict**

Two declarations for the same symbol have conflicting language attributes (such as **C** and **PASCAL**). The attributes should be identical or compatible.

A2193 **non-benign COMM redefinition**

A variable was redefined with the **COMM** directive to a different language type, distance, size, or instance count.

Multiple **COMM** definitions of a variable must be identical.

A2194 **COMM variable exceeds 64K**

A variable declared with the **COMM** directive in a 16-bit segment was greater than 64K.

A2195 **parameter or local cannot have void type**

The assembler attemped to create an argument or create a local without a type.

This error can be caused by declaring or passing a symbol followed by a colon without specifying a type or by using a user-defined type defined as void.

A2196 **cannot use TINY model with OS_OS2**

A **.MODEL** statement specified the **TINY** memory model and the **OS_OS2** operating system. The tiny memory model is not allowed under OS/2.

A2197 **expression size must be 32-bits**

There was an attempt to use the 16-bit expression evaluator in a 32-bit segment. In a 32-bit segment (**USE32** or **FLAT**), you cannot use the default 16-bit expression evaluator (**OPTION EXPR16**).

A2198 **.EXIT does not work with 32-bit segments**

The **.EXIT** directive cannot be used in a 32-bit segment; it is valid only under MS-DOS and OS/2 1.*x*.

A2199 **.STARTUP does not work with 32-bit segments**

The **.STARTUP** directive cannot be used in a 32-bit segment; it is valid only under MS-DOS and OS/2 1.*x*.

A2200 **ORG directive not allowed in unions**

The **ORG** directive is not valid inside a **UNION** definition.

You can use the **ORG** directive inside **STRUCT** definitions, but it is meaningless inside a **UNION**.

A2201 **scope state cannot be changed**

Both **OPTION SCOPED** and **OPTION NOSCOPED** statements occurred in a module. You cannot switch scoping behavior in a module.

This error may be caused by an **OPTION SCOPED** or **OPTION NOSCOPED** statement in an include file.

A2901 **cannot run ML.EXE**

The MASM driver could not spawn ML.EXE.

One of the following may have occurred:

- ML.EXE was not in the path.
- The READ attribute was not set on ML.EXE.
- There was not enough memory.

F.9.3 ML Warnings

Number	Message

A4000 **cannot modify READONLY segment**

An attempt was made to modify an operand in a segment marked with the READ-ONLY attribute.

A4002 **non-unique STRUCT/UNION field used without qualification**

A **STRUCT** or **UNION** field can be referenced without qualification only if it has a unique identifier.

This conflict can be resolved either by renaming one of the structure fields to make it unique or by fully specifying both field references.

The **NONUNIQUE** keyword requires that all references to the elements of a **STRUCT** or **UNION** be fully specified.

A4003 **start address on END directive ignored with .STARTUP**

Both **.STARTUP** and a program load address (optional with the **END** directive) were specified. The address specification with the **END** directive was ignored.

A4004 **cannot ASSUME CS**

An attempt was made to assume a value for the CS register. CS is always set to the current segment or group.

A4006 **too many arguments in macro call**

There were more arguments given in the macro call than there were parameters in the macro definition.

A4007 **option untranslated, directive required :** *option*

There is no ML command-line equivalent for the given MASM option. The desired behavior can be obtained by using a directive in the source file.

Option	Directive
/A	.ALPHA
/P	OPTION READONLY
/S	.SEQ

A4008 **invalid command-line option value, default is used :** *option*

The value specified with the given option was not valid. The option was ignored, and the default was assumed.

A4009 **virtual memory not available : /VM ignored**

The assembler was unable to initialize virtual memory.

You may be able to fix this error by freeing memory being used by RAM disks, caches, or TSR programs.

A4010 **insufficent memory for /EP : /EP ignored**

There is not enough memory to generate a first-pass listing.

A4011 **expected '>' on text literal**

A macro was called with a text literal argument that was missing a closing angle bracket.

A4012 **multiple .MODEL directives found : .MODEL ignored**

More than one **.MODEL** directive was found in the current module. Only the first **.MODEL** statement is used.

A4910 **cannot open file:** *filename*

The given filename could not be in the current path.

Make sure that *filename* was copied from the distribution disks and is in the current path.

A5000 **@@: label defined but not referenced**

A jump target was defined with the **@@:** label, but the target was not used by a jump instruction.

One common cause of this error is insertion of an extra **@@:** label between the jump and the **@@:** label that the jump originally referred to.

A5001 **expression expected, assume value 0**

There was an **IF**, **ELSEIF**, **IFE**, **IFNE**, **ELSEIFE**, or **ELSEIFNE** directive without an expression to evaluate. The assembler assumes a 0 for the comparison expression.

A5002 **externdef previously assumed to be external**

The **OPATTR** or **.TYPE** operator was applied to a symbol after the symbol was used in an **EXTERNDEF** statement but before it was declared. These operators were used on a line where the assembler assumed that the symbol was external.

A5003 **length of symbol previously assumed to be different**

The **LENGTHOF**, **LENGTH**, **SIZEOF**, or **SIZE** operator was applied to a symbol after the symbol was used in an **EXTERNDEF** statement but before it was declared. These operators were used on a line where the assembler assumed that the symbol had a different length and size.

A5004 **symbol previously assumed to not be in a group**

A symbol was used in an **EXTERNDEF** statement outside of a segment and then was declared inside a segment.

A5005 **types are different**

The type given by an **INVOKE** statement differed from that given in the procedure prototype. The assembler performed the appropriate type conversion.

A6001 **no return from procedure**

A **PROC** statement generated a prologue, but there was no **RET** or **IRET** instruction found inside the procedure block.

A6003 **conditional jump lengthened**

A conditional jump was encoded as a reverse conditional jump around a near unconditional jump.

You may be able to rearrange code to avoid the longer form.

F.10 NMAKE Error Messages

This section lists error messages generated by the Microsoft Program Maintenance Utility (NMAKE):

- Fatal errors (U1*0xx*) cause NMAKE to stop execution.
- Fatal errors (U1*4xx*) cause NMK to stop execution.
- Errors (U2*xxx*) do not stop execution but prevent NMAKE from completing the make process.
- Warnings (U4*xxx*) indicate possible problems in the make process.

F.10.1 NMAKE Fatal Errors

Number	NMAKE Error Message
U1000	**syntax error : ')' missing in macro invocation**

A left parenthesis (( ) appeared without a matching right parenthesis ( )) in a macro invocation. The correct form is $(*name*), or $*n* for one-character names.

U1001 **syntax error : illegal character** *character* **in macro**

A nonalphanumeric character other than an underscore (_) appeared in a macro.

U1002 **syntax error : invalid macro invocation '$'**

A single dollar sign ($) appeared without a macro name associated with it.

The correct form is $(*name*). To use a dollar sign in the file, type it twice ($$) or precede it with a caret (^).

U1003 **syntax error : '=' missing in macro**

The equal sign (=) was missing in a macro definition.

The correct form is

macroname=string

U1004 **syntax error : macro name missing**

A macro invocation appeared without a name.

The correct form is

$(*name*)

U1005 **syntax error : text must follow ':' in macro**

A string substitution was specified for a macro, but the string to be changed in the macro was not specified.

U1006 **syntax error : missing closing double quotation mark**

An opening double quotation mark (") appeared without a closing double quotation mark.

U1007 **double quotation mark not allowed in name**

The specified target name or filename contained a double quotation mark (").

Double quotation marks can surround a filename but cannot be contained within it.

U1017 **unknown directive !***directive*

The directive specified is not one of the recognized directives.

U1018 **directive and/or expression part missing**

The directive was incompletely specified.

The expression part of the directive is required.

U1019 **too many nested !IF blocks**

Note the circumstances of the error and notify Microsoft Corporation by following the instructions on the Microsoft Product Assistance Request form at the back of one of your manuals.

U1020 **end-of-file found before next directive**

A directive, such as **!ENDIF**, was missing.

U1021 **syntax error : !ELSE unexpected**

An **!ELSE** directive was found that was not preceded by **!IF**, **!IFDEF**, or **!IFNDEF**, or the directive was placed in a syntactically incorrect place.

U1022 **missing terminating character for string/program invocation :** *char*

The closing double quotation mark (") in a string comparison in a directive was missing, or the closing bracket ( ]) in a program invocation in a directive was missing.

U1023 **syntax error in expression**

An expression was invalid.

Check the allowed operators and operator precedence.

U1024 **illegal argument to !CMDSWITCHES**

An unrecognized command switch was specified.

U1031 **filename missing (or macro is null)**

An include directive was found, but the name of the file to be included was missing, or the macro expanded to nothing.

U1033 **syntax error : *string* unexpected**

The given string is not part of the valid syntax for a description file.

U1034 **syntax error : separator missing**

The colon (:) that separates targets and dependents is missing.

U1035 **syntax error : expected ':' or '=' separator**

Either a colon (:), implying a dependency line, or an equal sign (=), implying a macro definition, was expected.

U1036 **syntax error : too many names to left of '='**

Only one string is allowed to the left of a macro definition.

U1037 **syntax error : target name missing**

A colon (:) was found before a target name was found.

At least one target is required.

U1038 **internal error : lexer**

Note the circumstances of the error and notify Microsoft Corporation by following the instructions on the Microsoft Product Assistance Request form at the back of one of your manuals.

U1039 **internal error : parser**

Note the circumstances of the error and notify Microsoft Corporation by following the instructions on the Microsoft Product Assistance Request form at the back of one of your manuals.

U1040 **internal error : macro expansion**

Note the circumstances of the error and notify Microsoft Corporation by following the instructions on the Microsoft Product Assistance Request form at the back of one of your manuals.

U1041 **internal error : target building**

Note the circumstances of the error and notify Microsoft Corporation by following the instructions on the Microsoft Product Assistance Request form at the back of one of your manuals.

U1042 **internal error : expression stack overflow**

Note the circumstances of the error and notify Microsoft Corporation by following the instructions on the Microsoft Product Assistance Request form at the back of one of your manuals.

U1043 **internal error : temp file limit exceeded**

Note the circumstances of the error and notify Microsoft Corporation by following the instructions on the Microsoft Product Assistance Request form at the back of one of your manuals.

U1044 **internal error : too many levels of recursion building a target**

Note the circumstances of the error and notify Microsoft Corporation by following the instructions on the Microsoft Product Assistance Request form at the back of one of your manuals.

U1045 *internal error message*

Note the circumstances of the error and notify Microsoft Corporation by following the instructions on the Microsoft Product Assistance Request form at the back of one of your manuals.

U1046 **internal error : out of search handles**

This error occurs under OS/2 when there are not enough search handles for NMAKE to run.

U1049 **macro or inline file too long (maximum 64K)**

An inline file or a macro exceeded the limit of 64K.

U1050 *user-specified text*

The message specified with the **!ERROR** directive was displayed.

U1051 **out of memory**

The program ran out of space in the far heap.

Split the description file into smaller and simpler pieces.

U1052 **file** *filename* **not found**

The file was not found.

Check the specification of the filename in the description file.

U1053 **file *filename* unreadable**

The file cannot be read.

The following are possible causes of this error:

- The file does not have appropriate attributes for reading.
- A bad area exists on disk.
- A bad file-allocation table exists.
- The file is locked.

U1054 **cannot create inline file *filename***

NMAKE failed in its attempt to create the given file.

The following are possible causes of this error:

- The file already exists with a read-only attribute.
- There is insufficient disk space to create the file.

U1055 **out of environment space**

The environment space limit was reached.

Restart the program with a larger environment space or with fewer environment variables.

U1056 **cannot find command processor**

The command processor was not found.

NMAKE uses COMMAND.COM or CMD.EXE as a command processor to execute commands. It looks for the command processor first by the full pathname given by the COMSPEC environment variable. If COMSPEC does not exist, NMAKE searches the directories specified by the PATH environment variable.

U1057 **cannot delete temporary file *filename***

NMAKE failed to delete the temporary inline file.

U1058 **terminated by user**

Execution of NMAKE was aborted by CTRL+C or CTRL+BREAK.

U1060 **unable to close file :** *filename*

NMAKE encountered an error while closing a file.

One of the following may have occurred:

- The file is a read-only file.
- There is a locking or sharing violation.
- The disk is full.

U1061 **/F option requires a filename**

The /F command-line option requires the name of the description file to be specified.

To use standard input, specify '–' as the filename.

One cause of this error is omitting the space between /F and the filename.

U1062 **missing filename with /X option**

The /X command-line option requires the name of the file to which diagnostic error output should be redirected.

To use standard output, specify '–' as the output filename.

U1063 **missing macro name before '='**

NMAKE detected an equal sign (=) without a preceding name.

This error can occur in a recursive call when the macro corresponding to the macro name expands to nothing.

U1064 **MAKEFILE not found and no target specified**

No description file was found, and no target was specified.

A description file can be specified either with the /F option or in a file named MAKEFILE. Note that NMAKE can create a target using an inference rule even if no description file is specified.

U1065 **invalid option** *option*

The option specified is not a valid option for NMAKE.

U1066 **option /N not supported; use NMAKE /N**

NMK does not support the /N option. Run NMAKE with the /N option.

U1070 **cycle in macro definition** *macroname*

A circular definition was detected in the given macro definition.

Circular definitions are invalid.

U1071 **cycle in dependency tree for target** *targetname*

A circular dependency was detected in the dependency tree for the given target.

Circular dependencies are invalid.

U1072 **cycle in include files :** *filename*

A circular inclusion was detected in the given include file. This file includes a file which eventually includes this file.

U1073 **don't know how to make** *targetname*

The specified target does not exist, and there are no commands to execute or inference rules given for it.

U1074 **macro definition too long**

The value of a macro definition overflowed an internal buffer.

U1075 **string too long**

The text string overflowed an internal buffer.

U1076 **name too long**

The macro name, target name, or build-command name overflowed an internal buffer.

Macro names cannot exceed 128 characters.

U1077 *program* **: return code** *value*

The given program invoked from NMAKE failed, returning the given exit code.

U1078 **constant overflow at** *directive*

A constant in the directive's expression was too big.

U1079 **illegal expression : divide by zero**

An expression tried to divide by zero.

U1080 **operator and/or operand usage illegal**

The expression incorrectly used an operator or operand.

Check the allowed set of operators and their order of precedence.

U1081 *program* **: program not found**

NMAKE could not find the given program in order to run it.

Make sure that the program is in the current path and has the correct extension.

U1082 *command* **: cannot execute command; out of memory**

NMAKE cannot execute the given command because there is not enough memory.

Free some memory and run NMAKE again.

U1083 **target macro $(*macroname*) expands to nothing**

A target was specified as a macro name that has not been defined or has null value.

NMAKE cannot process a null target.

U1084 **cannot create temporary file** *filename*

NMAKE was unable to create a temporary file it needed for processing the description file.

The following are possible causes of this error:

- The file already exists with a read-only attribute.
- There is insufficient disk space to create the file.
- The TMP environment variable was set to an invalid directory or path.

U1085 **cannot mix implicit and explicit rules**

A regular target was specified along with the target for a rule.

A rule has the form

.fromext.toext

U1086 **inference rule cannot have dependents**

Dependents are not allowed when an inference rule is being defined.

U1087 **cannot have : and :: dependents for same target**

A target cannot have both a single-colon (**:**) and a double-colon (**::**) dependency.

U1088 **invalid separator '::' on inference rule**

Inference rules can use only a single-colon (**:**) separator.

U1089 **cannot have build commands for directive** *targetname*

Directives (for example, **.PRECIOUS** or **.SUFFIXES**) cannot have build commands specified.

U1090 **cannot have dependents for directive** *targetname*

The specified directive (for example, **.SILENT** or **.IGNORE**) cannot have a dependent.

U1091 **invalid suffixes in inference rule**

The suffixes being used in the inference rule are not part of the **.SUFFIXES** list.

U1092 **too many names in rule**

An inference rule cannot have more than one pair of extensions.

U1093 **cannot mix special pseudotargets**

It is illegal to list two or more pseudotargets together.

U1094 **syntax error : only (NO)KEEP allowed here**

Something other than **KEEP** or **NOKEEP** appeared at the end of the syntax for creating an inline file.

The syntax for generating an inline file allows an action to be specified after the second pair of angle brackets. Valid actions are **KEEP and NOKEEP**. Any other specification is invalid.

The **KEEP** option specifies that NMAKE should leave the inline file on disk. The **NOKEEP** option causes NMAKE to delete the file before exiting. The default is **NOKEEP**.

U1095 **expanded command line** *commandline* **too long**

After macro expansion, the command line shown exceeded the length limit for command lines for the operating system.

DOS permits up to 128 characters on a command line.

If the command is for a program that can accept command-line input from a file, change the command and supply input from either a file on disk or an inline file. For example, LINK and LIB accept input from a response file.

U1096 **cannot open file** *filename*

The given file could not be opened, either because the disk was full or because the file has been set to be read-only.

U1097 **extmake syntax usage error, no dependent**

No dependent was given.

In extmake syntax, the target under consideration must have either an implicit dependent or an explicit dependent.

U1098 **extmake syntax in *string* incorrect**

The part of the string shown contains an extmake syntax error.

U1099 **stack overflow**

The description file being processed was too complex for the current stack allocation in NMAKE.

NMAKE has a default allocation of 0x3000 (12K).

To increase NMAKE's stack allocation, run the EXEHDR utility with a larger stack option:

EXEHDR /STACK:*stacksize*

where *stacksize* is a number greater than the current stack allocation in NMAKE.

U1450 **could not execute NMAKE.EXE**

NMK was not able to locate and execute the NMAKE utility. Make sure this file is on your path.

U1451 **out of memory**

There was not enough available memory to complete the operation.

One of the following may be a cause:

- There are too many TSR programs installed. Remove some TSRs.
- A previous command did not release memory when it terminated. This can happen if you attempt to run a TSR from within NMK.
- There are too many active command shells. Close the current shell by entering `EXIT` at the operating-system prompt.

U1452 **COMSPEC not defined**

The COMSPEC environment variable is not defined

NMK requires COMSPEC to be set to the full pathname of the operating-system command processor.

U1453 **error reading script file**

NMK encountered an error while reading the script file, which contains commands to execute during a shell or build operation.

This can be caused by a CTRL+BREAK or a disk error while reading the script file.

U1454 *command*
could not execute

NMK was unable to execute the given command.

One of the following may have occurred:

- There was not enough available memory to execute the command. A previous command may not have released memory when it ended. This can happen if you attempt to run a TSR from within NMK.
- The operating system denied access to the file: it is in use by another program.
- The executable file is corrupt.

U1455 **bad command or file name**

An operating-system command or executable program could not be executed.

Either the command was spelled incorrectly, or it does not exist on the paths specified in the PATH environment variable.

F.10.2 NMAKE Errors

Number	NMAKE Error Message

U2001 **no more file handles (too many files open)**

NMAKE could not find a free file handle.

One of the following may be a solution:

- Reduce recursion in the build procedures.
- In DOS, increase the number of file handles by changing the FILES setting in CONFIG.SYS to allow a larger number of open files. FILES=20 is the recommended setting.

F.10.3 NMAKE Warnings

Number	NMAKE Warning
U4001	**command file can be invoked only from command line** A command file cannot be invoked from within another command file. The invocation was ignored. The command file must contain the entire remaining command line.
U4002	**resetting value of special macro** *macroname* The value of a macro such as **$(MAKE)** was changed within a description file.
U4003	**no match found for wildcard** *filename* There are no filenames that match the specified target or dependent file with the wildcard characters asterisk (*) and question mark (?).
U4004	**too many rules for target** *targetname* Multiple blocks of build commands were specified for a target using single colons (**:**) as separators. To use multiple dependency blocks for the same target, specify a pair of colons (**::**) as the separator.
U4005	**ignoring rule** *rule* **(extension not in .SUFFIXES)** The rule was ignored because the suffix(es) in the rule are not listed in the **.SUFFIXES** list.
U4006	**special macro undefined :** *macroname* The special macro name is undefined and expands to nothing.
U4007	**filename** *filename* **too long; truncating to 8.3** The base name of the given file has more than eight characters, or the extension has more than three characters. NMAKE truncated the name to an eight-character base and a three-character extension. You can use long filenames supported by HPFS under OS/2 by enclosing the name in double quotation marks.
U4008	**removed target** *target* Execution of NMAKE was interrupted while NMAKE was trying to build the given target, and therefore the target was incomplete. Because the target was not specified in the **.PRECIOUS** list, NMAKE has deleted it.

U4009 **duplicate inline file** *filename*

The given filename is the same as the name of an earlier inline file.

Reuse of this name caused the earlier file to be overwritten. This will probably cause unexpected results.

F.11 PWB.COM Error Messages

This section lists fatal error messages generated by the DOS Microsoft Programmer's WorkBench (PWB.COM). PWB errors (U13*xx*) prevent PWB from starting up, or returning from a build or operating-system shell.

Number PWB Error Message

U1350 **Could not execute PWBED.EXE**

PWB.COM could not find or load PWBED.EXE.

Make sure your system has the following configuration:

- Make sure PWBED.EXE can be found on the path specified in the PATH environment variable and that there is sufficient memory to operate PWB.
- Check that your environment contains the recommended settings from the NEW-VARS.BAT file created by the SETUP program when you installed PWB.

PWBED.EXE is the executable file for the PWB editor and environment. PWB.COM processes the command line on start-up and handles all system-level commands when building projects and executing Shell, User, Print, and Compile commands.

U1351 **out of memory**

There is not enough available memory to complete the operation.

Some possible causes for this error are

- You may have too many TSR programs installed. Remove some TSRs.
- A previous command may not have released memory when it terminated. This can happen if you attempt to run a TSR from within PWB.
- You may have too many active command shells. Leave the current shell with the operating-system Exit command.

U1352 **COMSPEC not defined**

The COMSPEC environment variable is not set.

PWB requires COMSPEC to be set to the full pathname of the operating-system command processor.

U1353 **error reading script file**

PWB.COM encountered an error while reading the file that contains a script of commands to execute during a shell or build operation.

This can be caused by a CTRL+BREAK or a disk error while reading the script file.

U1354 **could not execute**

PWB.COM was unable to execute the given command.

Some possible causes for this error are

- The executable file for the command was not found.
- The pathname of the command was not found.
- The operating system denied access to the file: it is in use by another program.
- There is not enough available memory to execute the command.

  A previous command may not have released memory when it terminated. This can happen if you attempt to run a TSR from within PWB.
- The environment is corrupt.
- The executable file is corrupt.

U1355 **Bad Command or Filename**

An operating-system command or executable program could not be executed.

The command may be spelled incorrectly, or it does not exist on the path specified in the PATH environment variable. Make sure that your environment contains the recommended settings from the NEW-VARS.BAT file created by the SETUP program when you installed PWB.

F.12 PWBRMAKE Error Messages

This section lists error messages generated by the Microsoft PWBRMAKE Utility (PWBRMAKE):

- Fatal errors (U15*xx*) cause PWBRMAKE to stop execution.
- Warnings (U45*xx*) indicate possible problems in the operation of PWBRMAKE.

F.12.1 PWBRMAKE Fatal Errors

Number	PWBRMAKE Error Message
U1500	**UNKNOWN ERROR** **Contact Microsoft Product Support Services** PWBRMAKE detected an unknown error condition. Note the circumstances of the error and notify Microsoft Corporation by following the instructions on the Microsoft Product Assistance Request form at the back of one of your manuals.
U1501	**unknown character** *character* **in option** *option* PWBRMAKE did not recognize the given character specified for the given option.
U1502	**incomplete specification for option** *option* The given option did not contain the complete specification that PWBRMAKE expected.
U1503	**cannot write to file** *filename* PWBRMAKE could not write to the given file. One of the following may have occurred: ■ The disk was full. ■ A hardware error occurred.

U1504 **cannot position in file** *filename*

PWBRMAKE could not move to a location in the given file.

One of the following may have occurred:

- The disk was full.
- A hardware error occurred.
- The file was truncated. Truncation can occur if the compiler runs out of disk space or is interrupted when it is creating the .SBR file.

U1505 **cannot read from file** *filename*

PWBRMAKE could not read from the given file.

One of the following may have occurred:

- The file was corrupt.
- The file was truncated. Truncation can occur if the compiler runs out of disk space or is interrupted when it is creating the .SBR file.

U1506 **cannot open file** *filename*

PWBRMAKE could not open the given file.

One of the following may have occurred:

- No more file handles were available. In DOS, increase the number of file handles by changing the FILES setting in CONFIG.SYS to allow a larger number of open files. FILES=20 is the recommended setting.
- The file was locked by another process.
- The disk was full.
- A hardware error occurred.
- The specified output file had the same name as an existing subdirectory.

U1507 **cannot open temporary file** *filename*

PWBRMAKE could not open one of its temporary files.

One of the following may have occurred:

- No more file handles were available. In DOS, increase the number of file handles by changing the FILES setting in CONFIG.SYS to allow a larger number of open files. FILES=20 is the recommended setting.
- The TMP environment variable was not set to a valid drive and directory.
- The disk was full.

U1508 **cannot delete temporary file** *filename*

PWBRMAKE could not delete one of its temporary files.

One of the following may have occurred:

- Another process had the file open.
- A hardware error occurred.

U1509 **out of heap space**

PWBRMAKE ran out of memory.

One of the following may be a solution:

- Reduce the memory that PWBRMAKE will require by using one or more options. Use /Ei or /Es to eliminate some input files. Use /Em to eliminate macro bodies.
- Free some memory by removing terminate-and-stay-resident (TSR) software.
- Reconfigure the EMM driver.
- Change CONFIG.SYS to specify a lower number of buffers (the BUFFERS command) and fewer drives (the LASTDRIVE command).

U1510 **corrupt .SBR file** *filename*

The given .SBR file is corrupt or does not have the expected format.

Recompile to regenerate the .SBR file.

U1511 **invalid response file specification**

PWBRMAKE did not understand the command-line specification for the response file. The specification was probably wrong or incomplete.

For example, the following specification causes this error:

```
pwbrmake @
```

U1512 **database capacity exceeded**

PWBRMAKE could not build a database because the number of definitions, references, modules, or other information exceeded the limit for a database.

One of the following may be a solution:

- Exclude some information using the /Em, /Es, or /Ei option.
- Omit the /Iu option if it was used.
- Divide the list of .SBR files and build multiple databases.

U1513 **nonincremental update requires all .SBR files**

An attempt was made to build a new database, but one or more of the specified .SBR files was truncated. This message is always preceded by warning U4502, which will give the name of the .SBR file that caused the error.

PWBRMAKE can process a truncated, or zero-length, .SBR file only when a database already exists and is being incrementally updated.

One of the following may be a cause:

- The database was deleted.
- The wrong database name was specified.
- The database file was corrupted, requiring a full build.

U1514 **all .SBR files truncated and not in database**

None of the .SBR files specified for an update was a part of the original database. This message is always preceded by warning U4502, which will give the name of the .SBR file that caused the error.

One of the following may be a cause:

- The wrong database name was specified.
- The database file was corrupted, requiring a full build.

F.12.2 PWBRMAKE Warnings

Number	PWBRMAKE Warning
U4500	**UNKNOWN WARNING** **Contact Microsoft Product Support Services** An unknown error condition was detected by PWBRMAKE. Note the circumstances of the error and notify Microsoft Corporation by following the instructions on the Microsoft Product Assistance Request form at the back of one of your manuals.
U4501	**ignoring unknown option** *option* PWBRMAKE did not recognize the given option and ignored it.
U4502	**truncated .SBR file** *filename* **not in database** The given zero-length .SBR file, specified during a database update, was not originally part of the database. If a zero-length file not part of the original build of the database is specified during a rebuild of that database, PWBRMAKE issues this warning. One of the following may be a cause: ■ The wrong database name was specified. ■ The database was deleted. (Error U1513 will result.) ■ The database file was corrupted, requiring a full build.

Glossary

8087, 80287, or 80387 coprocessor Intel chips that perform high-speed floating-point and binary coded decimal number processing. Also called math coprocessors. Floating-point instructions are supported directly by the 80486 processor.

A

address The memory location of a data item or procedure, or an expression that evaluates to an address. The expression can represent just the offset (in which case the default segment is assumed), or it can be in *segment*:*offset* format.

address constant In an assembly-language instruction, an immediate operand derived by applying the **SEG** or **OFFSET** operator to an identifier.

address range A range of memory bounded by two addresses.

addressing modes The various ways a memory address or device I/O address can be generated. See **far address, near address.**

aggregate types Data types containing more than one element, such as arrays, structures, and unions.

animate A debugging feature in which each line in a running program is highlighted as it executes. The Animate command from the CodeView debugger Run menu turns on animation.

API (application program interface) A set of system-level routines that can be used in an application program for tasks such as basic input/output and file management. In a graphics-oriented operating environment like Microsoft Windows, high-level support for video graphics output is part of the Windows graphical API. See **Family application program interface.**

arg In PWB, a function modifier that introduces an argument or an editing function. The argument may be of any type and is passed to the next function as input. For example, the PWB command `Arg textarg Copy` passes the text argument `textarg` to the function `Copy`.

argument A value passed to a procedure or function. See **parameter.**

array An ordered set of continuous elements of the same type.

ASCII (American Standard Code for Information Interchange) A widely used coding scheme where one-byte numeric values represent letters, numbers, symbols, and special characters. There are 256 possible codes. The first 128 codes are standardized; the remaining 128 are special characters defined by the computer manufacturer.

assembler A program that converts a text file containing mnemonically coded microprocessor instructions into the corresponding binary machine code. MASM is an assembler. See **compiler.**

assembly language A programming language in which each line of source code corresponds to a specific microprocessor instruction. Assembly language gives the programmer full access to the computer's hardware and produces the most compact, fastest executing code. See **high-level language.**

assembly mode The mode in which the CodeView debugger displays the assembly-language equivalent of the high-level code being executed. CodeView obtains the assembly-language code by disassembling the executable file. See **source mode.**

B

base address The starting address of a stack frame. Base addresses are usually stored in the BP register.

base name The portion of the filename that precedes the extension. For example, SAMPLE is the base name of the file SAMPLE.ASM.

BCD (binary coded decimal) A way of representing decimal digits where four bits of one byte are a decimal digit, coded as the equivalent binary number.

binary Referring to the base-2 counting system, whose digits are 0 and 1.

binary expression A Boolean expression consisting of two operands joined by a binary operator and resolving to a binary number.

binary file A file that contains numbers in binary form (as opposed to ASCII characters representing the same numbers). For example, a program file is a binary file.

binary operator A Boolean operator that takes two arguments. The **AND** and **OR** operators in assembly language are examples of binary operators.

BIOS (Basic Input/Output System) The software in a computer's ROM which forms a hardware-independent interface between the CPU and its peripherals (for example, keyboard, disk drives, video display, I/O ports).

bit Short for **bi**nary digi**t**. The basic unit of binary counting. Logically equivalent to decimal digits, except that bits can have a value of 0 or 1, whereas decimal digits can range from 0 through 9.

breakpoint A user-defined condition that pauses program execution while debugging. CodeView can set breakpoints at a specific line of code, for a specific value of a variable, or for a combination of these two conditions.

buffer A reserved section of memory that holds data temporarily, most often during input/output operations.

byte The smallest unit of measure for computer memory and data storage. One byte consists of eight bits and can store one 8-bit character (a letter, number, punctuation mark, or other symbol). It can represent unsigned values from 0 to 255 or signed values between –128 and +127.

C

C calling convention The convention that follows the order used by C—that is, pushing arguments onto the stack from right to left, or in reverse order from the way they are declared in the MASM procedure. The C calling convention permits a variable number of arguments to be passed.

chaining (to an interrupt) Installing an interrupt handler that shares control of an interrupt with other handlers. Control passes from one handler to the next until a handler breaks the chain by terminating through an **IRET** instruction. See **interrupt handler**.

character string A group of characters enclosed in single quotation marks (' ') or double quotation marks (" ").

child process In protected mode, a new process created by a currently executing process (its parent process).

clipboard In PWB, a section of memory that holds text deleted with the Copy, Ldelete, or Sdelete functions. Any text attached to the clipboard deletes text already there. The Paste function inserts text from the clipboard at the current cursor position.

.COM The filename extension for executable files that have a single segment containing both code and data. Tiny model produces .COM files.

combine type The segment-declaration specifier (**AT**, **COMMON**, **MEMORY**, **PUBLIC**, or **STACK**)

which tells the linker to combine all segments of the same type. Segments without a combine type are private and are placed in separate physical segments.

compact A memory model with multiple data segments but only one code segment.

compiler A program that translates source code into machine language. Usually applied only to high-level languages, such as Basic, Pascal, FORTRAN, or C. See **assembler**.

constant A value that does not change during program execution. A variable, on the other hand, is a value that can—and usually does—change. See **symbolic constant**.

constant expression Any expression that evaluates to a constant. It may include integer constants, character constants, floating-point constants, or other constant expressions.

D

debugger A utility program that allows the programmer to execute a program one line at a time and view the contents of registers and memory in order to help locate the source of bugs or other problems. Examples are CodeView and Symdeb.

declaration A construct that associates the name and the attributes of a variable, function, or type. See **variable declaration**.

default A setting or value that is assumed unless specified otherwise.

definition A construct that initializes and allocates storage for a variable, or that specifies either code labels or the name, formal parameters, body, and return type of a procedure. See **type definition**.

device driver A program that transforms I/O requests into the operations necessary to make a specific piece of hardware fulfill that request.

Dialog Command window The window at the bottom of the CodeView screen where dialog commands can be entered, and previously entered dialog commands can be reviewed.

direct memory operand In an assembly-language instruction, a memory operand that refers to the contents of an explicitly specified memory location.

directive An instruction that controls the assembler's state.

displacement In an assembly-language instruction, a constant value added to an effective address. This value often specifies the starting address of a variable, such as an array or multidimensional table.

DLL See **dynamic-link library**.

double-click To rapidly press and release a mouse button twice while pointing the mouse cursor at an object on the screen.

double precision A real (floating-point) value that occupies eight bytes of memory (MASM type **REAL8**). Double-precision values are accurate to 15 or 16 digits.

doubleword A four-byte word (MASM type DWORD).

drag To move the mouse while pointing at an object and holding down one of the mouse buttons.

dump To display the contents of memory at a specified memory range.

dynamic linking The resolution of external references at load time or run time (rather than link time). Dynamic linking allows the called subroutines to be packaged, distributed, and maintained independently of their callers. OS/2 extends the dynamic-link mechanism to serve as the primary method by which all system and nonsystem services are obtained.

dynamic-link library (DLL) A file, in a special format, that contains the binary code for a group of dynamically linked routines.

dynamic-link routine A routine that can be linked at load time or run time.

E

environment block The section of memory containing the DOS environment variables.

errorlevel code See **exit code**.

.EXE The filename extension for a program that can be loaded and executed by the computer. The small, compact, medium, large, huge, and flat models generate .EXE files. See **.COM, tiny**.

exit code A code returned by a program to the operating system. This usually indicates whether the program ran successfully.

expanded memory Increased memory available after adding an EMS (Expanded Memory Specification) board to an 8086 or 80286 machine. Expanded memory can be simulated in software. The EMS board can increase memory from 1 megabyte to 8 megabytes by swapping segments of high-end memory into lower memory. Applications must be written to the EMS standard in order to make use of expanded memory. See **extended memory**.

expression Any valid combination of mathematical or logical variables, constants, strings, and operators that yields a single value.

extended memory Physical memory above 1 megabyte that can be addressed by 80286–80486 machines in protected mode. Adding a memory card adds extended memory. On 80386-based machines, extended memory can be made to simulate expanded memory by using a memory-management program.

extension The part of a filename (of up to three characters) that follows the period (.). An extension is not required but is usually added to differentiate similar files. For example, the source-code file MYPROG.ASM is assembled into the object file MYPROG.OBJ, which is linked to produce the executable file MYPROG.EXE.

external variable A variable declared in one module and referenced in another module.

F

Family Application Program Interface (Family API) A standard execution environment under MS-DOS® (versions 2.*x* and later) and OS/2. The programmer can use the Family API to create an application that uses a subset of OS/2 functions (but a superset of MS-DOS 3.*x* functions).

far address A memory location specified with a segment value plus an offset from the start of that segment. Far addresses require four bytes—two for the segment and two for the offset. See **near address**.

field One of the components of a structure, union, or record variable.

fixup The linking process that supplies addresses for procedure calls and variable references.

flags register A register containing information about the status of the CPU and the results of the last arithmetic operation performed by the CPU.

flat A nonsegmented linear address space. Selectors in flat model can address the entire four gigabytes of addressable memory space. See **segment, selector**.

formal parameters The variables that receive values passed to a function when the function is called.

forward declaration A function declaration that establishes the attributes of a symbol so that it can be referenced before it is defined, or called from a different source file.

frame The segment, group, or segment register that specifies the segment portion of an address.

G

General-Protection (GP) fault An error that occurs in protected mode when a program accesses invalid memory locations or accesses valid locations in an invalid way (such as writing into ROM areas).

gigabyte 1,024 megabytes, or 1,073,741,824 bytes.

global See **visibility**.

global constant A constant available throughout a module. Symbolic constants defined in the module-level code are global constants.

global data segment A data segment that is shared among all instances of a dynamic-link routine; in other words, a single segment that is accessible to all processes that call a particular dynamic-link routine.

global variable A variable that is available (visible) across multiple modules.

granularity The degree to which library procedures can be linked as individual blocks of code. In Microsoft libraries, granularity is at the object-file level. If a single object file containing three procedures is added to a library, all three procedures will be linked with the main program even if only one of them is actually called.

group A collection of individually defined segments that have the same segment base address.

H

handle An arbitrary value that an operating system supplies to a program (or vice versa) so that the program can access system resources, files, peripherals, and so forth, in a controlled fashion.

hexadecimal The base-16 numbering system whose digits are 0 through F (the letters A through F represent the decimal numbers 10 through 15). This is often used in computer programming because it is easily converted to and from the binary (base-2) numbering system the computer itself uses.

high-level language A programming language that expresses operations as mathematical or logical relationships which the language's compiler then converts into machine code. This contrasts with assembly language, in which the program is written directly as a sequence of explicit microprocessor instructions. Basic, C, COBOL, FORTRAN, and Pascal are examples of high-level languages. See **assembly language**, **compiler**.

hooking (an interrupt) Replacing an address in the interrupt vector table with the address of another interrupt handler. See **interrupt handler**, **interrupt vector table**, **unhooking (an interrupt)**.

huge A memory model (similar to large model) with more than one code segment and more than one data segment. However, individual data items can be larger than 64K, spanning more than one segment. See **large**.

I

identifier A name that identifies a register or memory location.

IEEE format A standard created by the Institute of Electrical and Electronics Engineers for representing floating-point numbers, performing math with them, and handling underflow/overflow conditions. The 8087 family of coprocessors and the emulator package implement this format.

immediate expression An expression that evaluates to a number that can either be a component of an address or the entire address.

immediate operand In an assembly-language instruction, a constant operand that is specified at assembly time and stored in the program file as part of the instruction opcode.

include file A text file with the .INC extension whose contents are inserted into the source-code file and immediately assembled.

indirect memory operand In an assembly-language instruction, a memory operand whose value is treated as an address that points to the location of the desired data.

instruction The unit of binary information that a CPU decodes and executes. In assembly language, instruction refers to the mnemonic (such as **LDS** or **SHL**) that the assembler converts into machine code.

instruction prefix See **prefix**.

interrupt Instructions that cause a new sequence of actions to take place.

interrupt handler A routine that receives processor control when a specific interrupt occurs.

interrupt service routine See **interrupt handler**.

interrupt vector An address that points to an interrupt handler.

interrupt vector table A table maintained by the operating system. It contains addresses (vectors) of current interrupt handlers. When an interrupt occurs, the CPU branches to the address in the table that corresponds to the interrupt's number. See **interrupt handler**.

K

keyword A word with a special, predefined meaning for the assembler. In MASM 6.0, keywords cannot be used as identifiers.

kilobyte (K) 1,024 bytes.

L

label A symbol (identifier) representing the address of a code label or data objects.

language type The specifier that establishes the naming and calling conventions for a procedure. These are **BASIC**, **C**, **FORTRAN**, **PASCAL**, **STDCALL**, and **SYSCALL**.

large A memory model with more than one code segment and more than one data segment, but with no individual data item larger than 64K (a single segment). See **huge**.

library A file with the .LIB extension that stores modules of compiled code (object files). The linker extracts modules from the library and combines them with other object modules to create executable program files.

linked list A data structure in which each entry includes a pointer to the location of the adjoining entries.

linking The process in which the linker resolves all external references by searching the run-time and user libraries, and then computes absolute offset addresses for these references. The linking process results in a single executable file.

local constant A constant whose scope is limited to a procedure or a module.

local variable A variable whose scope is confined to a particular unit of code, such as module-level code, or a procedure. See **module-level code**.

logical device A symbolic name for a device that can be mapped to a physical (actual) device.

logical line A complete program statement in source code, including the initial line of code and any extension lines.

low-level input and output routines Run-time library routines that perform unbuffered, unformatted input/output operations.

LSB (least-significant bit) The bit lowest in memory in a binary number.

M

machine code The binary numbers that a microprocessor interprets as program instructions. See **instruction**.

macro A block of text or instructions that has been assigned an identifier. When the assembler sees this identifier in the source code, it substitutes the related text or instructions and assembles them.

main module The module containing the point where program execution begins (the program's entry point). See **module**.

math coprocessor See **8087, 80287, or 80387 coprocessor**.

medium A memory model with multiple code segments but only one data segment.

megabyte 1,024 kilobytes or 1,048,576 bytes.

member One of the elements of a structure or union; also called a field.

memory address A number through which a program can reference a location in memory.

memory map A representation of where in memory the computer expects to find certain types of information.

memory model A convention for specifying the number and types of code and data segments in a module. See **tiny**, **small**, **medium**, **compact**, **large**, **huge**, and **flat**.

memory operand An operand that specifies a memory location.

meta A prefix that modifies the subsequent PWB function.

mnemonic A word, abbreviation, or acronym that replaces something too complex to remember or type easily. For example, **ADC** is the mnemonic for the 8086's add-with-carry instruction. The assembler converts it into machine (binary) code, so it is not necessary to remember or calculate the binary form.

module A discrete group of statements. Every program has at least one module (the main module). In most cases, a module is the same as a source file.

module-level code Program statements within any module that are outside procedure definitions.

MSB (most-significant bit) The bit farthest to the left in a binary number. It represents $2^{(n-1)}$, where n is the number of bits in the number.

multitasking operating system An operating system in which two or more programs, processes, or threads can execute simultaneously.

N

naming convention The way the compiler or assembler alters the name of a routine before placing it into an object file.

NAN Acronym for "not a number." The math coprocessors generate NANs when the result of an operation cannot be represented in IEEE format. For example, if two numbers being multiplied have a product larger than the maximum value permitted, the coprocessor returns a NAN instead of the product.

near address A memory location specified by the offset from the start of the value in a segment register. A near address requires only two bytes. See **far address**.

nonreentrant See **reentrant procedure**.

null character The ASCII character encoded as the value 0.

null pointer A pointer to nothing, expressed as the value 0.

O

.OBJ Default filename extension for an object file.

object file A file (normally with the extension .OBJ) produced by assembling source code. It contains relocatable machine code. The linker combines object files with run-time and library code to create an executable file.

offset The number of bytes from the beginning of a segment to a particular byte within that segment.

opcode The binary number that represents a specific microprocessor instruction.

operand A constant or variable value that is manipulated in an expression or instruction.

operator One or more symbols that specify how the operand or operands of an expression are manipulated.

option A variable that modifies the way a program performs. Options can appear on the command line, or they can be part of an initialization file (such as TOOLS.INI). An option is sometimes called a switch.

OS/2 A multitasking operating system for the 80286–80486 family of personal computers.

output screen The CodeView screen that displays program output. Choosing the Output command from the View menu or pressing F4 switches to this screen.

overflow An error that occurs when the value assigned to a numeric variable falls outside the allowable range for that variable's type.

overlay A program component loaded into memory from disk only when needed. This technique reduces the amount of free RAM needed to run the program.

P

parameter The name given in a procedure definition to a variable that is passed to the procedure. See **argument**.

passing by reference Transferring the address of an argument to a procedure. This allows the procedure to modify the argument's value.

passing by value Transferring the value (rather than the address) of an argument to a procedure. This prevents the procedure from changing the argument's original value.

physical memory The hardware addresses of the actual RAM or ROM present in the computer.

physical segment The hardware address of a segment.

pointer A variable containing the address or relative offset of another variable.

precedence The relative position of an operator in the hierarchy that determines the order in which expression elements are evaluated.

preemptive Having the power to take precedence over another event.

prefix A keyword (**LOCK**, **REP**, **REPE**, **REPNE**, **REPNZ**, or **REPZ**) that modifies the behavior of an instruction. MASM 6.0 checks to be sure the prefix is compatible with the instruction.

private Data items and routines local to the module in which they are defined. They cannot be accessed outside that module. See **public**.

privilege level A hardware-supported feature of the 80286–80486 processors which allows the programmer to specify the exclusivity of a program or process. Programs running at low-numbered privilege levels can access data or resources at higher-numbered privilege levels, but the reverse is not true. This feature reduces the possibility that

malfunctioning code will corrupt data or crash the operating system.

privileged mode The term applied to privilege level 0. This privilege level should only be used by the OS/2 kernel and device drivers. Special privileged instructions are enabled by **.286P**, **.386P**, and **.486P**. This feature should not be confused with protected mode.

procedure call An expression that invokes a procedure and passes actual arguments (if any) to the procedure.

procedure definition A definition that specifies a procedure's name, its formal parameters, the declarations and statements that define what it does, and (optionally) its return type and storage class.

procedure prototype A procedure declaration that includes a list of the names and types of formal parameters following the procedure name.

process Generally, any executing program or code unit. This term implies that the program or unit is one of a group of processes executing independently.

Program Segment Prefix (PSP) A 256-byte data structure at the base of the memory block allocated to a transient program. It contains linkages to DOS and data from DOS that the program can use or ignore.

protected mode The 80286–80486 operating mode that permits multiple processes to run and not interfere with each other. This feature should not be confused with privileged mode.

public Data items and procedures that can be accessed outside the module in which they are defined. See **private**.

Q

qualifiedtype A user-defined type consisting of an existing MASM type (intrinsic, structure, union, or record), or a previously defined **TYPEDEF** type, together with its language or distance attributes.

R

radix The base of a number system. The default radix for MASM and CodeView is 10.

RAM (random-access memory) Computer memory that can both be written to and read from. RAM data is volatile; it is usually lost when the computer is turned off. Programs are loaded into and executed from RAM. See **ROM**.

real mode The normal operating mode of the 8086 family of processors. Addresses correspond to physical (not mapped) memory locations, and there is no mechanism to keep one application from accessing or modifying the code or data of another. See **protected mode**.

record A MASM variable that consists of a sequence of bit values.

reentrant procedure A procedure that can be safely interrupted during its execution and restarted from its beginning in response to a call from a preemptive process. After servicing the preemptive call, the procedure continues execution at the point at which it was interrupted.

register operand In an assembly-language instruction, an operand that is stored in the register specified by the instruction.

register window The optional CodeView window in which the CPU registers and the flag register bits are displayed.

registers Memory locations in the processor that temporarily store data, addresses, and logical values.

regular expression A text expression that specifies a pattern of text to be matched (as opposed to matching specific characters).

relocatable Not having an absolute address. The assembler does not know where the label, data, or

code will be located in memory, so it generates a fixup record. The linker provides the address.

return value The value returned by a function.

ROM (read-only memory) Computer memory that can only be read from and cannot be modified. ROM data is permanent; it is not lost when the machine is turned off. A computer's ROM often contains BIOS routines and parts of the operating system. See **RAM**.

routine A generic term for a procedure or function.

run-time dynamic linking The act of establishing a link when a process is started or is running.

run-time error A math or logic error that can be detected only when the program runs. Examples of run-time errors are dividing by a variable whose value is zero or calling a DLL function that doesn't exist.

S

scope The range of statements over which a variable or constant can be referenced by name. See **global constant**, **global variable**, **local constant**, **local variable**.

screen swapping A screen-exchange method that uses buffers to store the debugging and output screens. When you request the other screen, the two buffers are exchanged. This method is slower than flipping (the other screen-exchange method), but it works with most adapters and most types of programs.

scroll bars The bars that appear at the right side and bottom of a window and some list boxes. Dragging the mouse on the scroll bars allows scrolling through the contents of a window or text box.

segment A section of memory, limited to 64K with 16-bit segments or 4 gigabytes with 32-bit segments, containing code or data. Also refers to the starting address of that memory area.

sequential mode The mode in CodeView in which no windows are available. Input and output scroll down the screen, and the old output scrolls off the top of the screen when the screen is full. You cannot examine previous commands after they scroll off the top. This mode is required with computers that are not IBM compatible.

selector An address segment component supplied by a protected-mode operating system (such as OS/2). Programs that attempt to modify or directly manipulate these values may crash or cause system malfunctions.

shared memory A memory segment that can be accessed simultaneously by more than one process.

shell escape A method of gaining access to the operating system without leaving CodeView or losing the current debugging context. It is possible to execute DOS commands, then return to the debugger.

sign extended The process of widening (for example, going from a byte to a word, or a word to a doubleword) a negative integer while retaining its correct value and sign.

signed integer A binary integer that uses the most-significant bit to represent signed quantities. If this bit is one, the number is negative; if zero, the number is non-negative. See **two's complement**, **unsigned integer**.

single precision A real (floating-point) value that occupies four bytes of memory. Single-precision values are accurate to six or seven decimal places.

single-tasking environment An environment in which only one program runs at a time. DOS is a single-tasking environment.

small A memory model with only one code segment and only one data segment.

source file A text file containing symbols that define the program.

source mode The mode in which CodeView displays the assembly-language source code that represents the machine code currently being executed.

stack A dynamically shrinking and expanding area of memory in which data items are stored consecutively and removed on a last-in, first-out basis. A stack can be used to pass parameters to procedures.

stack frame The portion of a stack containing a particular procedure's local variables and parameters.

stack probe A short routine called on entry to a function to verify that there is enough room in the program stack to allocate local variables required by the function and, if so, to allocate those variables.

stack switching Changing pointers (usually the SS:SP register) to point to another stack or stack frame.

stack trace A symbolic representation of the functions that are being executed to reach the current instruction address. As a function is executed, the function address and any function arguments are pushed on the stack. Therefore, tracing the stack shows the active functions and their arguments.

standard error The device to which a program can send error messages. The display is normally standard error.

standard input The device from which a program reads its input. The keyboard is normally standard input.

standard output The device to which a program can send its output. The display is normally standard output.

statement A combination of labels, data declarations, directives, or instructions that the assembler can convert into machine code.

static linking The combining of multiple object and library files into a single executable file with all external references resolved. See **dynamic linking**.

status bar The line at the bottom of the PWB or CodeView screen. The status bar displays text position, keyboard status, current context of execution, and other program information.

STDCALL A calling convention that uses caller stack cleanup if the **VARARG** keyword is specified. Otherwise the called routine must clean up the stack.

string A contiguous sequence of characters identified with a symbolic name.

string literal A string of characters and escape sequences delimited by single quotation marks (' ') or double quotation marks (" ").

structure A set of elements or fields, which may be of different types, grouped under a single name.

structure member One of the elements of a structure. Also called a field.

switch See **option**.

symbol A name that identifies a memory location (usually for data).

symbolic constant A constant represented by a symbol rather than the constant itself. Symbolic constants are defined with **EQU** statements. They make a program easier to read and modify.

SYSCALL A language type for a procedure. Its conventions are identical to C's, except no underscore is prefixed to the name. All OS/2 version 2.0 functions use the **SYSCALL** language type.

T

tag The name assigned to a structure, union, or enumeration type.

task See **process**.

text Ordinary, readable characters, including the uppercase and lowercase letters of the alphabet, the numerals 0 through 9, and punctuation marks.

text box In PWB, a box where you type information needed to carry out a command. A text box appears within a dialog box. The text box may be blank or contain a default entry.

tiny Memory model with a single segment for both code and data. This limits the total program size to 64K. Tiny programs have the filename extension .COM.

toggle A function key or menu selection that turns a feature off if it is on, or on if it is off. Used as a verb, "toggle" means to reverse the status of a feature.

TOOLS.INI A file containing initialization information for many of the Microsoft utilities, including PWB.

two's complement A form of base-2 notation in which negative numbers are formed by inverting the bit values of the equivalent positive number and adding 1 to the result.

type A description of a set of values and a valid set of operations on items of that type. For example, a variable of type **BYTE** can have any of a set of integer values within the range specified for the type on a particular machine.

type checking An operation in which the assembler verifies that the operands of an operator are valid or that the actual arguments in a function call are of the same types as the function definition's parameters.

type definition The storage format and attributes for a data unit.

U

unary expression An expression consisting of a single operand preceded or followed by a unary operator.

unary operator An operator that acts on a single operand, such as **NOT**.

underflow An error condition that occurs when a calculation produces a result too small for the computer to represent.

unhooking (an interrupt) The act of removing your interrupt handler and restoring the original vector. See **hooking (an interrupt)**.

union A set of values (in fields) of different types that occupy the same storage space.

unresolved external See **unresolved reference**.

unresolved reference A reference to a global or external variable or function that cannot be found, either in the modules being linked or in the libraries linked with those modules. An unresolved reference causes a fatal link error.

unsigned integer A positive binary integer; all its bits represent the magnitude of the number. See **signed integer**.

user-defined type A data type defined by the user. It is usually a structure, union, record, or pointer.

V

variable declaration A statement that initializes and allocates storage for a variable of a given type.

virtual disk A portion of the computer's random access memory reserved for use as a simulated disk drive. Also called an electronic disk or RAM disk. Unless saved to a physical disk, the contents of a virtual disk are lost when the computer is turned off.

virtual memory Memory space allocated on a disk, rather than in RAM. Virtual memory allows

large data structures that would not fit in conventional memory, at the expense of slow access.

visibility The characteristic of a variable or function that describes the parts of the program in which it can be accessed. An item has global visibility if it can be referenced in every source file constituting the program. Otherwise, it has local visibility.

W

watch window The window in CodeView that displays watch statements and their values. A variable or expression is watchable only while execution is occurring in the section of the program (context) in which the item is defined.

window A discrete area of the screen in PWB or CodeView used to display part of a file or to enter statements.

window commands Commands that work only in CodeView's window mode. Window commands consist of function keys, mouse selections, CTRL and ALT key combinations, and selections from pop-up menus.

window mode The mode in which CodeView displays separate windows, which can change independently. CodeView has mouse support and a wide variety of window commands in window mode.

word A data unit containing 16 bits (two bytes). It can store values from 0 to 65,535 (or -32,768 to +32,767).

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Microsoft Product Assistance Request — MASM 6.0

Microsoft Product Support Services
Phone (206) 646-5109

Instructions

When you need assistance with a Microsoft product and you are calling from the United States, contact our Product Support Services group at **(206) 646-5109**. If you are calling from another country, please contact the nearest Microsoft subsidiary. (The subsidiaries' phone numbers are on the preaddressed labels included in the package.) So that we can answer your questions as quickly as possible, please gather all information that applies to your problem. Note or print out any on-screen messages you get when the problem occurs. Have your manual and product disks close at hand and have available all the information requested on this form when you call.

So that we can assist you more effectively, please be prepared to answer the following questions regarding your problem, your software, and your hardware.

Diagnosing a Problem

1 Can you reproduce the problem?

❑ yes ❑ no

Steps to duplicate problem:

_____

_____

_____

_____

2 Does the problem occur with another copy of the original disk of your Microsoft software?

❑ yes ❑ no

3 Does the problem occur with another system (if available)?

❑ yes ❑ no

4 If you were running other windowing or memory-resident software at the same time, does the problem also occur when you don't use the other software?

❑ yes ❑ no

_____
Name/Version Number

_____
Name/Version Number

5 Which version of the linker are you using? (To display the version number on your screen, type LINK at the DOS or OS/2 prompt and press ENTER.)

_____
Version Number

Product

_____
Name/Version Number

Operating System

_____
Name/Version Number

Hardware

Computer

_____
Manufacturer/Model CPU (e.g., 8088, 80286)

_____
Capacity (megabyte)

Note: If using DOS, you can run CHKDSK to determine the amount of memory available. If using Apple® Macintosh® Finder™, select "About the Finder..." from the Apple menu to determine the amount of memory available.

▪ Floppy-disk drives

Number: ❑ 1 ❑ 2 ❑ other

Density: ❑ single ❑ double ❑ quad

Capacity 5.25": ❑ 160K ❑ 360K ❑ 1.2 MB

3.5": ❑ 360K ❑ 720K ❑ 1.4 MB

▪ Hard Disks

_____
Manufacturer/Model Capacity (megabyte)

_____
Manufacturer/Model Capacity (megabyte)

Peripherals

▪ Printer/Plotter

Manufacturer/Model ❑ Serial ❑ Parallel

Printer peripherals, such as font cartridges, downloadable fonts, sheet feeders:

▪ Mouse

Microsoft® Mouse: ❑ Bus ❑ Serial ❑ InPort® ❑ PS/2® ❑ Other

Manufacturer/Model

▪ Boards

❑ Add-on RAM board/EMS boards

Manufacturer/Model/Total Memory

❑ Graphics-adapter board

Manufacturer/Model

❑ Other boards installed

Manufacturer/Model

Manufacturer/Model

▪ Modem

Manufacturer/Model

CD-ROM Player

Manufacturer/Model

Version of Microsoft MS-DOS® CD-ROM Extensions:

Network

Is your system part of a network? ❑ yes ❑ no

Manufacturer/Model

What software does your network use?

◆ ◆ ◆

Documentation Feedback – Microsoft® MASM 6.0

Please help us improve our documentation. When you become familiar with this product, complete and return this form. Comments and suggestions become the property of Microsoft Corporation.

Please answer the following questions about your programming background and practice.

Programming experience:
Total years ____ Years using MASM ____
Occupation: _____

Was it easy to set up this product for your programming environment? Yes ____ No ____
Comments: _____

What percentage of your programming is done in the Programmer's WorkBench (PWB)? ____
Outside PWB? ____

What editor(s) other than PWB do you use? _____

Please answer the following questions about the documentation. Then, using a scale of 1–5, rate the overall effectiveness of each piece.
(1 = Poor, 2 = Below average, 3 = Satisfactory, 4 = Very good, 5 = Excellent)

Microsoft Macro Assembler Advisor (online help system)

1. Do you use the Microsoft Macro Assembler Advisor? Yes ____ No ____
   Why or why not? _____
2. How useful are the example programs?
   Not useful ____ Somewhat useful ____
   Very useful ____ Comments: _____
3. Can you find the information you need quickly and easily? Yes ____ No ____
4. List any information you expected to find that was not there. _____
5. Rate the amount of information on each screen:
   Too much ____ Not enough ____ About right ____
   Comments: _____
6. What improvements would you like in future versions of online help? _____
7. Rating (1–5): ____ Comments: _____

Installing and Using the Professional Development System

1. Did you use this book to install MASM 6.0?
   Yes ____ No ____
2. Did the chapter on using the Programmer's WorkBench cover PWB's features adequately?
   Yes ____ No ____ Didn't read ____
3. Did the chapter on using online help explain the help system clearly?
   Yes ____ No ____ Didn't read ____
4. Rating (1–5): ____ Comments: _____

Programmer's Guide

1. Which statement best summarizes your response to the *Programmer's Guide*?
   ____ It's too simple; I want more in-depth information.
   ____ It's about right; I can understand and use it without difficulty.
   ____ It's too technical; I find it hard to read and apply.
2. Which chapters do you find most helpful? _____
   Least helpful? _____
3. What other topics would you like to see covered? _____
4. Rating (1–5): ____ Comments: _____

Reference

1. Which section(s) do you use the most? _____
   The least? _____
2. What other topics or information should be covered? _____
3. Rating (1–5): ____ Comments: _____

Which parts of the documentation do you refer to most frequently? _____

Use the back of this form for additional suggestions and comments. Please note any errors and special strengths or weaknesses in areas such as programming examples, indexes, and overall organization.

Name

Address

City/State/Zip

Phone ( ) — (home) ( ) — (work)

Additional comments:

Please mail this form to:

Microsoft Corporation
One Microsoft Way
Redmond, WA
98052-6399

Attn: Languages—MASM 6.0

Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-6399

Microsoft®
Making it all make sense™

0291 Part No. 06556