MASM Documentation

- Getting Started
- Reference Guide
- Programmer's Guide
- Environment and Tools

Microsoft MASM 6.1 Documentation

Getting Started		
Table of Contents	Ch. 1 - MASM Overview	Ch. 2 - Installing and Using MASM
Ch 3. - Configuring Your System		

Back to Top

Reference Guide		
Title Page	Copyright Page	Table of Contents
Introduction	Ch. 1 - Tools	Ch. 2 - Directives
Ch. 3 - Symbols and Operators	Ch. 4 - Processor	Ch. 5 - Coprocessor
Ch. 6 - Macros	Ch. 7 - Tables	

Back to Top

Programmer's Guide		
Title Page	Copyright Page	Table of Contents
Introduction	Ch. 1 - Understanding Global Concepts	Ch. 2 - Organizing Segments
Ch. 3 - Using Addresses and Pointers	Ch 4. - Defining and Using Simple Data Types	Ch 5. - Understanding and Using Complex Data Types
Ch. 6 - Using Floating-Point and Binary Coded Decimal Numbers	Ch 7. - Controlling Program Flow	Ch. 8 - Sharing Data and Procedures Among Modules and Libraries
Ch. 9. - Using Macros	Ch. 10 - Writing a Dynamic-Link Library for Windows	Ch. 11 - Writing Memory-Resident Software
Ch. 12 - Mixed-Language Programming	Ch 13. - Writing 32-Bit Applications	App. A - Differences Between MASM 6.1 and 5.1
App. B - BNF Grammar	App. C - Generating and Reading Assembly Listings	App. D - MASM Reserved Words
App. E - Default Segment Names	Glossary	Index
Documentation Feedback		

Back to Top

Environment and Tools		
Title Page	Copyright Page	Table of Contents
Introduction		
Part 1 - The Programmer's WorkBench		
Ch. 1 - Introducing the Programmer's WorkBench	Ch. 2 - Quick Start Ch. 2 continued	Ch. 3 - Managing Multimodule Programs
Ch 4. - User Interface Details	Ch. 5 - Advanced PWB Techniques Ch. 5 continued	Ch. 6 - Customizing PWB
Ch. 7 - Programmer's WorkBench Reference		
Part 2 - The CodeView Debugger		
Ch. 8 - Getting Started with CodeView	Ch. 9 - The CodeView Environment	Ch. 10 - Special Topics
Ch. 11 - Using Expressions in CodeView	Ch. 12 - CodeView Reference	
Part 3 - Compiling and Linking		
Ch. 13 - Linking Object Files with LINK	Ch. 14 - Creating Module-Definition Files	Ch. 15 - Using EXEHDR
Part 4 - Utilities		
Ch. 16 - Managing Projects with NMAKE	Ch. 17 - Managing Libraries with LIB	Ch. 18 - Creating Help Files with HELPMAKE
Ch. 19 - Browser Utilities	Ch. 20 - Using Other Utilities	
Part 5 - Using Help		
Ch. 21 - Using Help		
Appendixes		
App. A - Error Messages	App. B - Regular Expressions	Glossary
Index		

Back to Top

Getting Started

Microsoft Corporation

Microsoft® MASM

Assembly-Language Development System
Version 6.1

For MS-DOS® and Windows™ Operating Systems

Information in this document is subject to change without notice. Companies, names, and data used in examples herein are fictitious unless otherwise noted. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Microsoft Corporation.

©1992 Microsoft Corporation. All rights reserved.

Microsoft, MS, MS-DOS, XENIX, CodeView, and QuickC are registered trademarks and Windows and Windows NT are trademarks of Microsoft Corporation in the USA and other countries.

U.S. Patent No. 4955066

IBM is a registered trademark of International Business Machines Corporation.
Intel is a registered trademark of Intel Corporation.
Printed in the United States of America.

Document No. DB35753-1292

Contents

Chapter 1 Microsoft Macro Assembler (MASM) Overview 1
System Requirements 2
Package Contents 2
Product Components 2
New MASM Features 3
Document Conventions 4

Chapter 2 Installing and Using MASM 5
Using Setup 5
Reviewing Installation Settings 7
System Files 9
Installing MASM for Use With Other Programming Languages 11
Running MASM 13
Running MASM from the MS-DOS Command Line 13
Running MASM Within the Windows Operating System 13
Getting More Information 14

Chapter 3 Configuring Your System 15
Understanding System Configuration Terminology 15
Choosing a Development Environment 19
Revising System Files 19
Modifying Your AUTOEXEC.BAT File 20
Modifying Your CONFIG.SYS File 23
Modifying Your .PIF Files 24
Modifying Your SYSTEM.INI File 25
Modifying Your TOOLS.INI FILE 25
Using Your DOSXNT.EXE File 26
Increasing System Speed 26
Optimizing Disk Access Time 26
Using SMARTDRV.EXE 27
Using RAMDRIVE.SYS 31
Optimizing Available Memory 31
Understanding Memory Requirements 32
Determining Memory Availability 32

Freeing Conventional Memory 33
Enabling Extended Memory with HIMEM.SYS 33
Freeing Extended Memory 34
Freeing Expanded Memory 35
Using EMM386.EXE as an Expanded Memory Emulator 35
Using EMM386.EXE to Manage Upper Memory 36
Other DPMI Servers 39
Optimization Summary 39

CHAPTER 1

Microsoft Macro Assembler (MASM) Overview

This chapter describes the features of MASM version 6.1. The following topics are included:

- System Requirements
- Package Contents
- Product Components
- New MASM features
- Documentation Conventions

System Requirements

MASM version 6.1 requires the following system configuration:

- An IBM Personal Computer or 100 percent compatible, running MS-DOS version 3.3 or later
- An 80386 or later processor
- 4 megabytes of available memory (RAM)
- One hard-disk drive with a minimum of 5 megabytes of free space (Depending on the options you select, you may need up to 9 megabytes of disk space. The SETUP program will ask what components you want installed and then check to see if your system has enough disk space to install all the components you selected.)
- One 1.2 megabyte, 5.25-inch floppy disk drive, or one 1.44 megabyte, 3.5-inch floppy disk drive. (For information on the 720K MASM disk set, see “Package Contents.”)

Package Contents

Your MASM version 6.1 package should include the items listed below. If any pieces are missing, contact the retailer from whom you purchased the product.

- *Registration card*. There are many advantages to being a registered owner of MASM, including notification of future software releases and easy access to customer assistance. Please take the time to fill out and mail the registration card now. If you are already a registered owner (from an earlier version of MASM) and have upgraded, your upgrade kit will not include a registration card.
- *Disks*. Disk 1 of the MASM disk set contains a file named PACKING.TXT that lists the name, location and a brief description of each disk file in the MASM package. Most files on the disks are compressed; the SETUP program decompresses files as they are installed. MASM is distributed on five 5.25-inch high-density, or four 3.5-inch high-density disks. If you need 3.5-inch 720K disks to install MASM, please send the media order card contained in the MASM package, or call Microsoft Customer Service (1-800-426-9400).
- *Books*. Your package should contain following books:
  - *Getting Started (this book)* *Getting Started* includes information on system requirements, tells you how to set up the software, and provides instructions on optimizing your system for use with MASM.
  - *Environment and Tools* This book describes how to use the Programmer's WorkBench (PWB), the CodeView (CV) Debugger, and all the other utilities included with your MASM package.
  - *Programmer's Guide* This advanced programming text describes the enhanced features and technical details of MASM version 6.1.
  - *Macro Assembler Reference* This quick-reference book lists the utilities along with a brief description of their command-line options, directives, symbols and operators and include-file macro names. Complete information on processor and coprocessor instructions is also included.

Product Components

MASM version 6.1 includes all the components you need to develop Assembly Language programs for the MS-DOS and Windows operating systems. The following components are included:

- ML Assembler version 6.1

- Programmer’s WorkBench version 2

- CodeView version 4 Debugger
- The latest versions of LINK, LIB, IMPLIB, NMAKE, BSCMAKE, CREF, H2INC, EXEHDR, CVPACK, SBRPACK, HELPMAKE, RM, UNDEL, and EXP utilities
- On-line Help for the assembler and all utilities
- Sample code
- Readme documentation for information unavailable at the time of printing.

New MASM Features

MASM version 6.1 includes the following new features:

- MASM can run as a 32-bit application under MS-DOS version 3.3 and later, and the Windows operating system version 3.*x* (see *Programmer's Guide*).
- The new command-line option sequence /Fl /Sc causes the assembler to show instruction timings in the listing file (see MASM *Reference*).
- The assembler accepts '*@file*' in the command line to specify a response file to extend the command line (see MASM *Reference*).
- The new /COFF command-line option causes the assembler to convert .OBJ output to COFF (see MASM *Reference*).
- Updated versions of the programming utilities are included (see *Environment and Tools*).
- Improved compatibility with MASM version 5.10 (see Appendix A in *Programmer's Guide*).
- Allows creation of Windows-based DLLs without the Microsoft Windows Software Development Kit (see *Programmer's Guide*).
- Includes sample code for writing DLLs for calling from the Windows operating system version 3.*x* (see *Programmer's Guide).*

Document Conventions

The MASM document set uses the following conventions:

Example	Description
COPY TEST.ASM C:	Uppercase letters represent MS-DOS commands and filenames.
INVOKE	Boldface letters indicate standard features of the MASM language: keywords, operators, and standard library functions.
expression	Words in italics indicate place holders for information you must supply, such as a filename. Italics are also occasionally used for emphasis in the text.
ML /Zi HELLO.ASM	This typeface is used for example programs, program fragments, and the names of user-defined functions and variables. It also indicates user input and screen output.
SHIFT	Small capital letters denote names of keys on the keyboard. A plus sign (+) indicates a combination of keys. For example, SHIFT+F5 tells you to hold down the SHIFT key while pressing the F5 key.
"bookmark"	The first time a new term is defined, it is enclosed in quotation marks. Since some knowledge of programming is assumed, common terms such as *memory* or *branch* are not defined.

CHAPTER 2

Installing and Using MASM

This chapter describes the MASM version 6.1 installation. It includes detailed information about:

- Using SETUP
- Reviewing Installation Settings
- Running MASM
- Getting More Information

Before running SETUP, back up the distribution disks and make sure you have enough disk space (see "System Requirements," page 1). For information on configuring your system after you have installed MASM, see Chapter 3 of this book.

Using Setup

To install Microsoft MASM version 6.1, run the SETUP.EXE program (located on Disk 1 of your installation set). The SETUP program performs all tasks necessary for installing the MASM components. You must run SETUP to install MASM, as the files on the distribution disks are compressed. SETUP decompresses the files and copies them to your hard disk. SETUP runs under MS-DOS and under the Windows operating system version 3.*x*.

You can use SETUP to perform the following:

- View the documentation notes, README.TXT, packing list, and information for users of MASM version 5.10.
- Preview the installation prompts and their defaults before installing any files.

- Install the Macro Assembler using the defaults or while modifying the loading options.
- Copy individual files from the distribution disks.

You can use the interactive installation, which is the default, to set the following options:

- Load utilities for use with the Windows operating system (default = yes).
- Load the Programmer's WorkBench (default = yes).
- Configure PWB with BRIEF-compatible environment commands (default = no).
- Load MASM.EXE for MASM version 5.10 compatibility (default = yes).
- Copy Help files, README.TXT, and other documentation files (default = yes).
- Copy the sample programs (default = no).

Note If you plan on using the Tutorials in the *Programmer's Guide*, you should load the sample programs.

- Copy a mouse driver (default = yes).
- Select the drive where you want the MASM files to reside (default = highest drive letter).
- Select the directories for the MASM component files you choose to install. These include:
  - Executable files (default = C:\MASM61\BIN)
  - Library files (default = C:\MASM61\LIB)
  - Include files (default = C:\MASM61\INCLUDE)
  - Initialization files (default = C:\MASM61\INIT)
  - Help files (default = C:\MASM61\HELP)
  - Sample files (default = C:\MASM61\SAMPLES)

Note If you are using more than one Microsoft programming language, you may want to direct the files to other than the default directories to avoid duplicating utilities. For more information on installing MASM in a multi-language environment, see "Installing MASM for Use With Other Programming Languages" on page 11.

- Check your TMP environment variable and available disk space installing any files. If the TMP environment variable is not set when you run SETUP, SETUP will propose C:\MASM61\TMP as the temporary directory on your hard disk to use during installation. This directory will not be deleted when installation is complete. SETUP will place the line

  ```
  SET TMP=C:\MASM61\TMP
  ```

  in the NEW-VARS.BAT file if a TMP environment variable is not defined.

  If there is a problem during installation, SETUP will report the error and terminate without loading MASM. For information on how SETUP uses your system's TMP environment variable, see page 21.

You can run SETUP from either the MS-DOS command line or within the Windows operating system version 3.*x*.

Running SETUP from the MS-DOS Command Line

1. Insert the disk labeled Disk 1 in the appropriate disk drive.
2. At the MS-DOS command prompt, type

   DRIVE`:\ENTER`

   where *DRIVE* is the disk drive into which you just put Disk 1.
3. Type SETUP and press ENTER to begin installation.
4. Press ENTER again to display the Main Menu screen in Figure 2.1.

Running SETUP from the Windows operating system

1. Insert the disk labeled Disk 1 in the appropriate disk drive.
2. Open the File Manager and view the contents of Disk 1.
3. Run the SETUP.EXE file by double-clicking on it with the mouse, or by selecting it and pressing ENTER.
4. Press ENTER to display the main menu.

Reviewing Installation Settings

After SETUP is complete, the SETUP Main Menu screen, shown in Figure 2.1, appears.

To review the default settings for installing MASM, choose Run SETUP Without Installing Any Files from the SETUP Main Menu. SETUP will go through the prompting screens and then return to the Main Menu screen.

Figure 2.1 SETUP Main Menu screen.

Follow the instructions on the screen. Press ENTER to proceed with a selection you have made. Use ARROW KEYS to make a selection. Press F1 for information on a selection. Press CTRL+C to quit SETUP.

If you are running SETUP interactively, the Confirm Your Choices screen shown in Figure 2.2 appears when you have viewed all the prompts. This screen allows you to change any of the installation selections you made.

Figure 2.2 SETUP Confirm Your Choices Screen

To change a setting or to access more information on each menu item, use the ARROW KEYS to make the selection, then press ENTER. To accept all the settings, select No Changes and press ENTER.

After SETUP is complete, the Environment Settings screen shown in Figure 2.3 appears.

Figure 2.3 SETUP Environment Settings screen

System Files

SETUP does not modify your system files. Instead, SETUP copies recommended updates that are named NEW-VARS.BAT, NEW-CONF.SYS, and NEW-SYS.INI to your \MASM61\BIN subdirectory. Depending on your system configuration, some of the settings in these files are necessary for MASM to
run on your system.

NEW-CONF.SYS is a sample CONFIG.SYS file containing system commands for your CONFIG.SYS file. If your CONFIG.SYS file has the same commands, then make sure they are set to the same or greater values as in NEW-CONF.SYS.

NEW-VARS.BAT is a batch file that sets the MASM environment variables. You can run NEW-VARS.BAT from the command line or merge it with your existing AUTOEXEC.BAT file.

If you chose to install the Windows operating system utilities during SETUP, the SYSTEM.INI Setting screen shown in Figure 2.4 will be displayed.

Figure 2.4 EM.INI Settings screen

NEW-SYS.INI contains commands that should exist in the [386.enh] section of your Microsoft Windows SYSTEM.INI file. It also lists any [386.enh] section lines you should delete to complete the update.

The Sample PWB Settings screen shown in Figure 2.5 is displayed next. It tells you the location of the TOOLS.PRE file that contains the default PWB setting. You can rename this file TOOLS.INI, or merge the information in it with your existing TOOLS.INI file.

Figure 2.5 Sample PWB Settings screen

The Memory Utilities screen shown in Figure 2.6 is displayed next. It tells you the location of HIMEM.SYS, RAMDRIVE.SYS and SMARTDRV.EXE. For more information about these memory utilities, see Chapter 3 of this book.

Figure 2.6 Memory Utilities screen

Pressing ENTER displays SETUP's Main Menu screen again. From here you may exit SETUP, view documentation files, or run the installation process again. You may run SETUP again at any time to load any files you chose to exclude during this installation.

Installing MASM for Use With Other Programming Languages

If you will be using MASM with other Microsoft programming languages, such as Microsoft C/C++, you may already have versions of PWB, CV and the other programming utilities loaded. You can install MASM in one of three ways, depending on your working requirements and available hard disk space:

- Install MASM in your existing MASM tree structure, if one exists. This will update all identically named and placed files in the MASM tree. The SETUP program defaults to this option. If a copy of MASM version 5.10 is located in the \MASM directory, it will be renamed to OLDMASM.EXE.
- Install MASM in your high-level language tree structure. This will also update identically named and placed files in the high-level language tree. For example,

if you have Microsoft C/C++ version 7.0 on drive D:, the root of your C/C++ tree is \C700. If you wanted to install MASM in your \C700 tree, you would:

- Select Install the Microsoft Macro Assembler from the Main Menu.
- Specify C700 instead of MASM in each of the target directory prompts.

- Install MASM in an independent tree. MASM will have its own complete tree structure, so any identically named utilities or files that exist between the new MASM tree and your high-level language (or previous MASM version) trees will be preserved.

  If you select this option and want to use the newest versions of the Programmer's Work Bench, CodeView, and other programming utilities loaded during MASM setup, you will need to make sure the directory that contains your executable MASM files (default \MASM61\BIN) comes before other language \BIN directories in your MS-DOS PATH statement.

If SETUP detects files that are named and located identically to files it is about to install, it checks the time/date stamps of those files. If the files are newer than the files SETUP is about to install, and the file is supplied by another Microsoft language product (other than MASM), you are warned and given three options:

- Copy new files over old files. This deletes the old files from your hard disk.
- Do not copy new files, and keep the older versions. The new versions are not copied to your hard disk.
- Exit SETUP. If you exit SETUP you can save your old files in another location.

There are two exceptions:

- If you choose to install the new MASM.EXE utility, your old MASM.EXE is renamed OLDMASM.EXE and left in the same directory.
- You are warned if SETUP is about to overwrite any version of LINK.EXE that is different from the one SETUP is about to install.

If you will be using Microsoft FORTRAN, BASIC, or C/C++ with MASM, you need to activate the appropriate PWB extensions (PWBFORT.XXT for FORTRAN, PWBBASIC.XXT for BASIC, and PWBC.XXT for C/C++). These are located in the C:\MASM61\BIN directory. To activate an extension, change the .XXT extension to .MXT. Language extensions provided with earlier versions of PWB are not compatible with PWB version 2.0, so you must use the new .XXT files if you want to use a language extension.

Note Any extensions you wrote for a previous version of PWB must be rebuilt for PWB version 2.0. Building custom PWB extensions for MASM 6.1 requires the Microsoft C/C++ Version 7 programming set. For more information on PWB extensions, see *Programmer's Guide*.

Running MASM

You can run MASM from the MS-DOS command line, or in an MS-DOS application window within the Windows operating system version 3.*x*. The configuration procedure for MASM to run under the two platforms is slightly different. (For information on using MASM after it is running, see *Programmer's Guide*.)

Running MASM from the MS-DOS Command Line

If you plan to run MASM from the MS-DOS command line, make sure of three things:

- Your computer has booted with a CONFIG.SYS file that includes the commands listed in NEW-CONF.SYS.
- The environment variables listed in NEW-VARS.BAT are set.
- The MS-DOS extender file DOSXNT.EXE is in the path or current directory.

During SETUP, NEW-CONF.SYS, NEW-VARS.BAT, and DOSXNT.EXE are copied in the directory you specify for executable files (default \MASM61\BIN).

Running MASM Within the Windows Operating System

If you plan to run MASM within the Windows operating system version 3.*x*, you may want to add some of the MASM utilities to a program group in the Program Manager. MASM.GRP is copied to the \BIN directory you specify during SETUP for your executable files.

Note Make sure the MASM61\BIN directory is in the current path before you

add MASM.GRP to your Program Manager. You may need to exit the Windows operating system to verify the current path. If the directory MASM61\BIN is not part
of the current path, you will have to add the MASM.GRP program items and icons individually.

Adding the MASM Program Group

1. Open the Program Manager.
2. From the File menu, choose New.
3. Select Program Group.
4. Choose the OK button.
5. Type *DRIVE*`:\MASM61\BIN\MASM.GRP` and press ENTER

DRIVE is the MASM-resident drive. (If you specified a different directory for your executable files during the SETUP program, type that as the path for MASM.GRP instead.)

The Program Manager adds a new MASM Program Group. It has five program items: Programmer's WorkBench, MASM 6.1 Reference, CodeView, MS-DOS CodeView, and WXServer.

Once you have added your MASM program group and included the statements from NEW-SYS.INI to your SYSTEM.INI file, you must exit the Windows operating system to save your changes. Restart the Windows operating system to use the items in your new MASM program group.

For more information on adding program items and groups to your Windows operating system, see your Windows operating system *User's Guide*. For information on using PWB, CV or WX Server, see *Environment and Tools*.

Getting More Information

While SETUP is running, press F1 during any screen to access more information on the highlighted option.

For information on particular components, commonly asked questions, or information not available at the time of printing, use SETUP's Main Menu to view the README.TXT file.

If you have checked these sources as well as information found elsewhere in the documentation set, and you need to contact Microsoft Product Support

Services, see the information on contacting Microsoft located in the front of *Environment and Tools*.

CHAPTER 3

Configuring Your System

This chapter describes how to configure your system for optimal use of MASM and explains the recommended modifications to your system files (CONFIG.SYS, AUTOEXEC.BAT, and SYSTEM.INI).

This chapter also provides information on conventional memory, extended memory, expanded memory, and memory managers. This will help you:

- make more memory available for MASM and other programs
- optimize the speed at which your programs run
- use the memory in your system more efficiently

You may need to experiment with the described techniques to find the right optimization for your system.

Note MS-DOS version 5.0 or later provides many new features that make memory configuration easier. Many of the recommendations made in this chapter require your system to have these new MS-DOS features. If you have not upgraded to
MS-DOS version 5.0 or later, you may want to do so before configuring your system files for MASM.

Understanding System Configuration Terminology

This section defines terms that can help you understand the configuration information in this chapter. Figure 3.1 shows the relationship between the different memory areas.

Figure 3.1 Memory Locations

Conventional (Real) Memory
: The first 640K of memory in a computer using an Intel-compatible processor. All MS-DOS systems have conventional memory. All application programs can use conventional memory without additional memory-management programs.

Extended Memory
: Memory above the first 1 MB of memory on systems with 80286 or higher processors. Most 80386 computers come with some extended memory. Extended memory requires an extended memory manager, such as HIMEM.SYS, to prevent programs from using the same area of extended memory at the same time. You need nearly 3.5 megabytes of extended memory to run MASM on your system (4 megabytes of total system memory).

High Memory Area (HMA)
: The first 64K of extended memory. Systems using MS-DOS version 5.0 or later can load MS-DOS into the HMA. This will free about 50K of conventional memory.

Expanded Memory

An area of memory accessible to programs that can access memory above 640K. Expanded memory is divided into 16K segments called “pages.” When a program requests information from expanded memory, an expanded memory manager “maps” or copies the appropriate page to an area called a “page frame” in upper memory. Since an expanded memory manager allows programs access to a limited amount of information at one time, expanded memory can be slower for programs to use than extended memory. Expanded memory requires special drivers such as EMM386.EXE. EMM386.EXE can also use extended memory to emulate expanded memory on 80386 and 80486 systems.

Upper Memory

Also referred to as High MS-DOS Memory. The 384K of memory above the 640K of conventional memory in most systems. Parts of this area not used by your system are called “upper memory blocks” (UMB). If your system has an 80386 or 80486 processor and extended memory, MS-DOS can be loaded into UMBs, so more conventional memory is free for programs. MS-DOS version 5.0 or later has commands that enable you to store certain device drivers and programs in upper memory. This memory cannot be accessed by user programs.

Device Driver

A program that MS-DOS uses to control devices such as the keyboard, mouse, monitor, disk drives, and physical memory. Memory managers are device drivers. Device drivers are loaded into memory by statements in your CONFIG.SYS file.

Memory Manager

A program that provides access to a particular type of memory. For programs
to use extended memory, expanded memory, or upper memory, your system must have a memory manager. MASM provides two memory managers, HIMEM.SYS and EMM386.EXE, that can be installed on your system. Although memory managers take up some space in conventional memory, they provide access to extended memory, expanded memory, and upper memory.

HIMEM.SYS

A memory manager that provides access to extended memory. HIMEM.SYS is required for MASM.

EMM386.EXE

A device driver provided by MASM to control expanded memory and provide access to upper memory. The EMM386.EXE memory manager can also use extended memory to emulate expanded memory (see page 35).

SMARTDRV.EXE
: A device driver provided by MASM for systems running MS-DOS (version 4.*x* and later) that enables faster disk access. SMARTDRV.EXE creates a disk cache in extended or expanded memory. (SMARTDRV.EXE replaces the earlier version of this program, SMARTDRV.SYS.)

Double Buffering
: A SMARTDRV option that provides compatibility for hard disk controllers that cannot work with virtual memory.

Disk Cache
: An area in extended or expanded memory that SMARTDRV.EXE uses to store information it reads from the hard disk. This speeds up disk access because the next information the application requests may already be available in memory.

RAMDRIVE.SYS
: A device driver provided by MASM for systems running MS-DOS (version 4.*x* and later) that reduces disk access. RAMDRIVE.SYS creates a virtual disk drive in RAM to emulate a physical disk drive.

DOS Protected Mode Interface (DPMI)
: A published specification for handling MS-DOS calls in protected mode programs. The Microsoft Windows operating system version 3.*x* provides DPMI services and is therefore called a “DPMI server.”

MS-DOS Extensions to the DPMI
: These extensions provide additional functionality not required by the DPMI specification. The Windows operating system version 3.*x* provides this additional functionality.

Virtual Control Program Interface (VCPI)
: Defines how multiple programs can run in protected mode on MS-DOS.

MS-DOS-Extended Programs
: Programs that have a protected mode MS-DOS extender bound into the executable file. This allows the program to use extended memory and to use real-mode interrupt services in protected mode.

WX Server
: WXSRVR.EXE works with WX.EXE, a real-mode MS-DOS program, to allow Windows-based programs to be invoked from within a Windows operating system MS-DOS application window.

Choosing a Development Environment

The MASM assembler components require extended memory and an XMS memory manager, such as HIMEM.SYS. MASM supports the DPMI and VCPI specifications, and will use any available DPMI and VCPI allocated memory. However, neither DPMI or VCPI is required to run MASM.

You may run MASM within MS-DOS, or within an MS-DOS application window under the Windows operating system. If you are using the Windows operating system, version 3.*x*, you can have multiple MS-DOS application windows operating simultaneously. For example, you can use one MS-DOS window for editing and another for compiling.

Revising System Files

You will need to modify your CONFIG.SYS and AUTOEXEC.BAT files to use some of the MASM features. Perform the following procedures to update these system files safely:

- Make a backup copy of these files before modifying them.
- To disable a statement without deleting it, insert the word REM in front of an AUTOEXEC.BAT or CONFIG.SYS statement. (Note that use of REM in a CONFIG.SYS file generates a warning message in versions of MS-DOS prior to 4.0.)
- Make a system disk in case a change to your CONFIG.SYS file makes it impossible to restart from your hard disk.

Note When you finish making changes to the files, you must restart your system to enable the changes.

The information in this section applies to a system running MS-DOS version 5.0 or later and the Windows operating system version 3.*x*. If you are not using a Microsoft memory manager, follow the manufacturer's instructions. If your system is running MS-DOS version 3.*x* or version 4.*x*, you cannot load MS-DOS into high memory, because the DEVICEHIGH command is not available for CONFIG.SYS files, and the LOADHIGH command is not available for the AUTOEXEC.BAT file in these versions of MS-DOS. Since upper memory cannot be accessed, less memory is available for the MASM components.

Modifying Your AUTOEXEC.BAT File

The AUTOEXEC.BAT file contains settings or definitions for environment variables. Several environment variables need to be defined so the MASM components can work together optimally. The NEW-VARS.BAT file includes most of the SET statements you need to run MASM.

Required Environment Variables

The following environment variables are required by MASM:

INIT
: Specifies the directory where TOOLS.INI and CURRENT.STS initialization files are located. If INIT is set, the Programmer's WorkBench (PWB) looks for TOOLS.INI and CURRENT.STS in the directory specified by INIT. If you do not set INIT, PWB and the CodeView debugger create a CURRENT.STS file in every directory from which PWB or CodeView are invoked, making it unlikely that the correct status file will be loaded the next time PWB or CodeView is run. Only one directory should be specified for the INIT variable.

PATH
: Specifies the search path for finding executable files.

TMP
: Operating-system environment variable that specifies the directory for temporary files. Only one directory can be specified in the TMP variable. Utilities that use the TMP environment variable are NMAKE, LINK, and PWB.

HELPFILES
: Required for using Help with PWB, CodeView, and QuickHelp. Specifies the list of directories where Help files (.HLP) are located, and the filenames of specific .HLP files. Wildcard characters are allowed in the HELPFILES variable to indicate more than one .HLP file.

Note Do not place Microsoft Windows-based Help files in the HELPFILES directory. They are not compatible with MASM help.

Optional Environment Variables

The following environment variables are optional for MASM:

LIB
: Specifies the list of directories, separated by semicolons, where library files (*.LIB) are located.

INCLUDE
: Specifies the list of directories, separated by semicolons, where INCLUDE files (.INC files for MASM, .H files for C/C++) are located.

MASM
: Additional options for the MASM 5.10 compatibility driver.

ML
: Specifies additional options for the assembler.

LINK
: Specifies additional options for the linker.

You can use environment variables in makefiles. NMAKE uses a set of macro definitions equivalent to the setting of each environment variable when it runs. The macro definitions can be redefined without changing the value of the environment variable. An environment variable can also be defined in a makefile if it has not already been defined. You can view the current settings for macros by specifying the /p option for NMAKE. For more information on NMAKE and its options, see the *Environment and Tools*.

Using the TMP and TEMP Environment Variables

Microsoft programming languages and the utilities included with them use the environment variable TMP. This operating system environment variable is typically set in the AUTOEXEC.BAT file and is assigned to the drive and directory used for temporary file storage.

For example, PWB needs at least 1 MB free in the directory specified by the TMP environment variable because that directory is the location for PWB's virtual memory file.

While Microsoft development tools use the TMP environment variable, Microsoft applications, such as Microsoft Word, use an environment variable called TEMP. Both variables are set to a drive and directory for temporary file storage. Since compilers and other development tools use more temporary disk space than applications, you may want to assign TMP to your hard disk. Applications generally use less temporary disk space, allowing you to set TEMP equal to a small RAM drive.

Always set TMP to an existing subdirectory, as in the following example:

```
SET TMP=C:\TMP
```

Remember the following when using the TMP environment variables:

- If the TMP environment variable is not set when you run SETUP, SETUP will prompt you for the path to store temporary files (default=\MASM61\TMP). If the path you give it doesn't exist, SETUP will create the directory. If you specify a drive in the path that doesn't exist, SETUP will create and use \MASM61\TMP. This temporary directory is not deleted when SETUP is finished installing MASM. After installation, SETUP will place the line

  ```
  SET TMP=\MASM61\TMP
  ```

  in the NEW-VARS.BAT file if a TMP environment variable is not set when it is run.

- If your TMP environment variable points to a location on a network drive, make sure that the directory is not write protected.
- Make sure the directory pointed to by your TMP environment variable exists. If you set the TMP environment variable to a non-existing directory, SETUP will use the root directory of the current drive for storing temporary files. This can cause a problem, since MS-DOS limits the number of files that can be created in a root directory.

For more information on how MS-DOS and the Windows operating system use temporary files and how to change file settings, see your MS-DOS or Windows operating system documentation.

Setting Environment Space

The memory area created by the operating system to store environment variables and their values is called the "environment space." If the assembler can't find files like include files, or if you receive this error message when rebooting:

```
Out of environment space
```

the available environment space is insufficient to hold the definitions of the environment variables.

The default environment size is 256 bytes. The size of the current environment is 256 bytes or the amount of actual memory used by the environment variables rounded up to the next 16 bytes. The limits are 160 to 32,768 bytes. The "current" environment is the actual memory being used for the environment variables, not the size specified with the /e option of the SHELL command in

the CONFIG.SYS file. (The size of the current environment may be less than the size specified with the

SHELL command.) For correct operation, set the value for the environment size to at least 1024 with this statement:

```
SHELL = C:\DOS\COMMAND.COM /e:1024 /p
```

Note Use the SHELL options carefully. It's a good idea to have a system disk with working CONFIG.SYS and AUTOEXEC.BAT files when experimenting with your system.

The SHELL command specifies the name and location of the command interpreter you want MS-DOS to use and sets the environment space to 1024 bytes. The default is 256 bytes. The /p parameter tells MS-DOS to make its associated command interpreter permanent so that you cannot type EXIT to stop the command interpreter. It also tells MS-DOS to run your AUTOEXEC.BAT file when it carries out the SHELL command. See your MS-DOS documentation for more information on the SHELL command.

Modifying Your CONFIG.SYS File

The amount of memory available and the configuration for MS-DOS and Windows operating system are controlled by your CONFIG.SYS file, your SYSTEM.INI file, and your .PIF files. If you are not familiar with these files, see your MS-DOS and Windows operating system, version 3.*x* documentation.

SETUP does not modify your current system files. It writes all suggested changes
to the NEW-CONF.SYS, NEW-VARS.BAT, and NEW-SYS.INI files in the C:\MASM61\BIN subdirectory. For more information about these files, see page 9.

Note Depending on your existing system configuration, some of these changes are necessary for MASM to run on your system.

SETUP adds the BUFFERS, FILES, and DEVICE commands to your NEW-CONF.SYS file or modifies the values set for these commands.

BUFFERS

The BUFFERS command in your CONFIG.SYS file specifies the number of buffers that MS-DOS reserves for file transfers. The greater the number of buffers (up to about 50), the faster your system runs. However increasing the number of buffers past a certain value will cause your system to use more memory without increasing speed. Each buffer requires 532 bytes of memory.

Table 3.1 shows the recommended number of buffers in relation to hard disk size.

Table 3.1 Recommended Number of Buffers

Hard-Disk Size	Number of Buffers
Less than 40 MB	20
40–79 MB	30
80–119 MB	40
More than 120 MB	50

If you have SMARTDRV.EXE installed, set BUFFERS to 10. SMARTDRV.EXE provides much of the increase in system speed that setting BUFFERS to a higher value would accomplish.

FILES

The FILES command sets the number of files MS-DOS can access at the same time. Each file uses 48 bytes of conventional memory. If the number of FILES is set too low, compilations may fail because include files cannot be opened. The default value is 8. If you will be running MASM in a Windows operating system MS-DOS application window, use the setting recommended in your Windows operating system *User's Guide*. If you will be running MASM from the MS-DOS command line, this value will depend on the size of your programs. Start with a minimum of 20, and increase it to a higher value if necessary.

DEVICE, DEVICEHIGH

The DEVICE command loads a device driver. The DEVICEHIGH command loads a device driver into upper memory on MS-DOS version 5.0 or later. DEVICEHIGH also requires adding the DOS=UMB command and a DEVICE statement to HIMEM.SYS and EMM386.EXE. Setting DOS=UMB is necessary if you want to load programs and device drivers into upper memory. This command tells MS-DOS to maintain a link to upper memory. HIMEM.SYS must be installed before you specify the DOS command.

Modifying Your .PIF Files

The .PIF file sets the amount of expanded memory and extended memory used by a program that is not Windows-based and enables background execution by default. None of the MASM programs depend on expanded memory, and the

Windows operating system uses only extended memory. Therefore, the PWB .PIF file provided with MASM sets expanded memory to 0 and extended memory to –1. Setting extended memory to –1 makes available as much extended memory as possible. The .PIF files for an MS-DOS session should also use these settings, unless you have utilities that require expanded memory.

If you have a memory card that can be configured as expanded or extended memory, set it to extended memory and let the software emulate expanded memory if needed. If you have a memory card that provides expanded memory only, set extended memory to –1. Expanded memory should be set to the amount of expanded memory actually available, either from a memory card or from EMM386.EXE.

If you develop programs under the Windows operating system version 3.*x*, you can allow background execution with certain exceptions. When profiling or doing timing dependent tasks, you will want exclusive execution. If you find that background compiles interfere with your foreground work, increase the priority of foreground scheduling by changing the value for Background Priority for Multitasking Options in the PWB .PIF file.

You can also minimize the number of extensions that PWB loads by editing the .PIF file. You can specify the /DA option to suppress automatic loading of all PWB extensions, or you can put a question mark (**?**) in the Optional Parameter section of the PWB .PIF file. This specifies PWB to prompt you for command-line options each time it loads. For more information, see “About Projects,” and “About PWB Extensions” from the PWB Table of Contents in the Microsoft Advisor Help system.

Modifying Your SYSTEM.INI File

During installation, SETUP copies the following three lines into the file C:\MASM61\BIN\NEW-SYS.INI. These statements must be added to the [386enh] section of your SYSTEM.INI file:

```
DEVICE=C:\MASM61\BIN\DOSXNT.386
DEVICE=C:\MASM61\BIN\CVW1.386
DEVICE=C:\MASM61\BIN\VMB.386
```

The VMB.386 device line is added only if you choose to have SETUP install PWB. This line is used only by WX.EXE.

Modifying Your TOOLS.INI FILE

PWB and CodeView both use the TOOLS.INI file. Customization settings for PWB are located in TOOLS.INI, and CodeView looks for information in the [CV] and [CVW] sections of the TOOLS.INI file to do remote debugging. TOOLS.INI needs to be located in the directory pointed to by your INIT environment variable. If you don't have a TOOLS.INI file, copy (do not rename) TOOLS.PRE to TOOLS.INI. SETUP loads TOOLS.PRE to the C:\MASM61\INIT directory.

To load only a subset of the PWB extensions, move the extensions that are not in the load set to a directory that is not on the path. Then you can load these extensions only when necessary. You can modify your TOOLS.INI to load an individual PWB extension selectively. For more information, see the "About TOOLS.INI" entry from the PWB Table of Contents in the Microsoft Advisor Help system, or see Chapter 6, "Customizing PWB," in *Environment and Tools* . A TOOLS.INI file for the system that is the target side of a remote debugging session is also useful. For more information on remote debugging, see Chapter 10, "Special Topics," in *Environment and Tools* .

Using Your DOSXNT.EXE File

DOSXNT.EXE is the MS-DOS extender that allows you to run the MASM assembler. You must make sure that DOSXNT.EXE is located somewhere in the path or in the current working directory (default \MASM61\BIN).

Increasing System Speed

There are several ways to improve system speed and the speed of the programs you use. The following sections explain how to increase system speed by optimizing disk access time and by using the SMARTDRV.EXE disk cache program.

Optimizing Disk Access Time

To decrease disk access time, arrange the directories in the PATH statement of your AUTOEXEC.BAT file so that file searches are as efficient as possible. Executable files that you use often should be in directories at the beginning of your PATH statement. Removing unnecessary files from your hard disk can also decrease disk access time.

To optimize disk access time, perform the following:

- Use the CHKDSK /f command to recover lost disk space and then delete the files CHKDSK creates. Do not use CHKDSK when the Windows operating system is running. See your MS-DOS documentation before using CHKDSK.
- Delete obsolete Help files from previous installations of other Microsoft language products, especially UTILERR.HLP and CVW.HLP. See Chapter 23, "Using Help," in *Environment and Tools* for more information.
- Have SETUP install only the MASM options you need.

Using SMARTDRV.EXE

MASM includes SMARTDRV.EXE version 4.0, a sophisticated block-oriented disk cache program that significantly improves compilation and link times. SMARTDRV.EXE is not required by MASM, but can reduce the amount of time your computer spends reading data from your hard disk. SMARTDRV.EXE replaces the older version, SMARTDRV.SYS, and is compatible with all versions of the Windows operating system version 3.*x*.

SMARTDRV.EXE sets aside expanded or extended memory as a cache. SMARTDRV.EXE uses this disk cache to store the information read from the hard disk. When an application attempts to read additional information from the hard disk, the SMARTDRV.EXE program supplies the information directly from its cache instead.

If you are using RAMDRIVE.SYS to create one or more RAM drives, and are limiting the memory assigned to SMARTDRV.EXE as a result, you can increase system speed by reassigning some or all of the memory from the RAM drive and adding it to the memory available to SMARTDRV.EXE.

Install SMARTDRV.EXE by placing the following line in your AUTOEXEC.BAT file:

```
C:\MASM61\BIN\SMARTDRV.EXE
```

SMARTDRV.EXE automatically loads itself into high memory under MS-DOS version 5.0 if EMM386.EXE is loaded and upper memory blocks are available as
a result of a DOS=UMB or DOS=HIGH, UMB command in your CONFIG.SYS file. SMARTDRV.EXE can also be loaded into HMA with third-party memory managers such as 386-Max.

SETUP also checks your CONFIG.SYS file for a DEVICE statement for SMARTDRV.SYS. If SETUP finds a DEVICE statement for SMARTDRV.SYS,

it places the following DEVICE statement for SMARTDRV.EXE into your NEW-CONF.SYS file.

```
DEVICE=C:\MASM61\BIN\SMARTDRV.EXE /DOUBLE_BUFFER
```

Using Double Buffering

If you have a SCSI (Small Computer System Interface) hard disk controller, you may need to use the double buffering feature of SMARTDRV.EXE. Double buffering provides compatibility for hard disk controllers that cannot work with virtual memory.

For double buffering to be enabled, SMARTDRV.EXE must be specified in both your AUTOEXEC.BAT file and CONFIG.SYS file. The double buffer driver is installed in CONFIG.SYS, and the cache component of SMARTDRV.EXE is installed during execution of AUTOEXEC.BAT.

To enable SMARTDRV's double-buffering option, add the following line to the end of your CONFIG.SYS file:

```
DEVICE=C:\MASM61\BIN\SMARTDRV.EXE /DOUBLE_BUFFER
```

Some disk controllers do not need double buffering, so using this option when you do not need it results in some penalty in performance. SETUP does not determine if your system needs double buffering. Therefore, once your system is running with SMARTDRV.EXE, type:

```
SMARTDRV ENTER
```

at the command-line prompt. SMARTDRV.EXE displays disk cache information as illustrated in the following example:

```
Microsoft SMARTDrive Disk Cache version 4.0
Copyright 1991,1992 Microsoft Corp.

Cache size: 1,048,576 bytes
Cache size while running the Windows operating system: 1,048,576 bytes

                Disk Caching Status
    drive   read cache   write cache   buffering
    -------------------------------------------
      A:       yes           no           no
      B:       yes           no           no
      C:       yes           yes          yes
      D:       yes           yes          -

    For help, type Smartdrv /?.
```

Note If every line in the Buffering column is No, you do not need the DEVICE statement for SMARTDRV.EXE in your CONFIG.SYS file.

SMARTDRV.EXE always copies data to your hard disk when an application calls the MS-DOS reset disk function. If you want to force data to be written to disk, use

the /C command-line option. If you use a non-Microsoft utility to reboot your machine from a batch file, you should make sure you have

```
SMARTDRV /C
```

in the batch file prior to the reboot command. Failure to include this line can result in loss of data.

You can use command-line options to control the size of the cache element (/E) and the size of the read-ahead buffer (/B). The read-ahead buffer is additional information that SMARTDRV.EXE reads when the application reads information from the hard disk. The size must be specified in bytes, and the element size must be one of the following: 1024, 2048, 4096, or 8192. The read-ahead buffer must be a multiple of the element size, cannot be less than the element size, and cannot exceed 32768. The defaults are 8192 for the element size and 16384 for the read-ahead buffer. Because these will occupy conventional or upper memory, making them larger reduces the available memory for MS-DOS applications.

You can start SMARTDRV.EXE program by typing SMARTDRV at the MS-DOS prompt before you start the Windows operating system, or by placing a command line in your AUTOEXEC.BAT file. The syntax is:

[[*drive*:]] [[*path*]] S MARTDRV.EXE[[[[*drive*[[+|–]]]]…]] [[/E:*ElementSize*]] [[*InitCacheSize*]] [[*WinCacheSize*]]]] [[/B:*BufferSize*]] [[/C]] [[/R]] [[/L]] [[/Q]] [[/S]] [[/?]]

The following list describes the command-line options available for SMARTDRV.EXE:

Option	Description
drive	Specify the letter of the disk drive you want to control disk caching. If you don't specify a drive letter, floppy disk drives read operations are cached but write operations are not, hard disk drive read and write operations are cached, and CD-ROM and network drives are ignored. You can specify multiple disk drives.
path	Specify the location of the SMARTDRV.EXE file.
+ | –	Enable (+) or disable (–) disk caching. Use the plus (+) and minus (–) signs to override the default settings. If you specify a drive letter without a plus or minus sign, read operations are cached and write operations are not. If you specify a drive letter followed by a plus sign (+), read and write operations are both cached. If you specify a drive letter followed by a minus sign(–), neither read nor write operations are cached.

Option	Description
/E:*ElementSize*	Specify in bytes the amount of the disk cache SMARTDRV.EXE moves at a time. This must be greater than or equal to 1, and a power of 2. The default value is 8K.
InitCacheSize	Specify the size in kilobytes of the disk cache when SMARTDRV.EXE starts (before the Windows operating system is running). The size of the disk cache affects SMARTDRV.EXE's efficiency. In general, the larger the disk cache, the less often SMARTDRV.EXE needs to read information from the disk, which speeds up your system's performance. If you do not specify an *InitCacheSize* value, SMARTDRV.EXE sets the value according to how much memory your system has (see Table 3.2).
WinCacheSize	Limit the amount (in kilobytes) the Windows operating system can reduce the disk cache size. The Windows operating system reduces the size of the disk cache to recover memory for its own use. The Windows operating system and SMARTDRV.EXE cooperate to provide optimum use of your system memory. When you exit the Windows operating system, it restores the disk cache to its normal size. The default value depends on how much available memory your system has (see Table 3.2). If you specify a value for *InitCacheSize* that is smaller than the value specified for WinCacheSize, *InitCacheSize* is set to the same size as *WinCacheSize*
/B:*BufferSize*	Specify the size of the read-ahead buffer. The next time the application is to read information from that file, it can read it from memory instead. The default size of the buffer is 16K. Its value can be any multiple of *ElementSize*.
/C	Write all cached information from memory to the hard disk. SMARTDRV.EXE writes information from memory to the hard disk when other disk activity has slowed. You might use this option if you are going to turn off your computer, and you want to make sure all information has been written to the hard disk.
/R	Clear the contents of the existing disk cache and restart SMARTDRV.EXE.
/L	Prevent SMARTDRV.EXE from loading into upper memory blocks (UMBs), even if there are UMBs available. You can use this option if you are using MS-DOS version 5.0 or later and UMBs are enabled.
/Q	Prevent the display of SMARTDRV.EXE information on your screen.
/S	Display additional information about the status of SMARTDRV.EXE.
/?	Display online Help about the SMARTDRV.EXE command and options.

Table 3.2 shows the default values for *InitCacheSize* and *WinCacheSize*, depending on the amount of available extended memory on your computer.

Table 3.2 Default Values for InitCacheSize and WinCacheSize

Extended Memory	InitCacheSize	WinCache Size
Up to 1 MB	All extended memory	Zero (no caching)
Up to 2 MB	1 MB	256K
Up to 4 MB	1 MB	512K
Up to 6 MB	2 MB	1 MB
6 MB or more	2 MB	2 MB

Note Do not put the SMARTDRV.EXE disk cache in the expanded memory provided by EMM386.EXE. EMM386.EXE uses extended memory to emulate expanded memory that other programs can use. Although SMARTDRV.EXE can use this emulated expanded memory for its cache, it may not make your program run as quickly as it would using extended memory.

Also, earlier versions of SMARTDRV.EXE allowed you to use the /a switch to direct SMARTDRV to use expanded memory. This function is no longer valid with the current version.

The optimal disk cache size for SMARTDRV.EXE depends on the programs you run, and your system configuration. You should experiment to find the best disk cache size for your system, after you have saved a copy of your CONFIG.SYS file. For more information, see your Windows operating system *User's Guide*.

Using RAMDRIVE.SYS

If you don't want to use SMARTDRV.EXE, or if you have a large amount of memory, use RAMDRIVE.SYS to create a disk partition in RAM. To use it for assembler temporary files, set the TMP environment variable to the drive and directory of the RAM disk. The minimum recommended size for a RAM disk is 1 MB.

Optimizing Available Memory

You may need to experiment to find the right memory layout to ensure that all your MS-DOS applications have enough memory to run. The optimal memory layout may change according to the task to be performed. Therefore, you may need to change your CONFIG.SYS file, depending on what you need to

optimize. This section provides general information for making more memory available on your system when necessary.

Understanding Memory Requirements

Your CONFIG.SYS file includes statements that define how your system uses memory. This section describes the memory requirements for MASM components.

To run CodeView, LINK, ML, or CVPACK, you need an XMS, DPMI, or VCPI, server. HIMEM.SYS provides XMS services; EMM386.EXE provides VCPI services; the Windows operating system version 3.*x* in enhanced mode provides DPMI services. CodeView requires expanded memory when running as a VCPI application. If neither DPMI or VCPI service is available, CodeView uses extended memory provided by HIMEM.SYS.

If PWB cannot allocate enough memory, it may be unable to load all the extensions you selected, list boxes may come up empty, and performance may be slower
than usual. Try to make as much conventional and extended memory available as possible when running PWB. PWB will use expanded memory if it is available, but will run faster using extended memory.

Determining Memory Availability

To determine the size and type of memory in your system, use the MS-DOS MEM command to display the amount of used and free memory, list allocated and free memory areas, and list programs that are loaded. (The MEM command is available in MS-DOS version 4.*x* or later.)

Type MEM at the command-line prompt when the Windows operating system is not running:

```
MEM /c | more
```

The MEM command does not report the contents of upper memory if you have the Windows operating system loaded. Use the /c option to tell MS-DOS to display the status of programs loaded into conventional memory and upper memory. (The /c option is only available in MS-DOS version 5.0 or later.)

MS-DOS displays three columns of information about the programs currently using system memory: Name, Size in Decimal, and Size in Hex. See your MS-DOS documentation for more information about the MEM command.

You can use the CHKDSK command to check the amount of free conventional memory if you are using an MS-DOS version earlier to version 4.0. You can also examine system configuration with the MSD.EXE utility provided with MASM. See MSD.TXT in C:\MASM61\BIN for information about running MSD.EXE.

Freeing Conventional Memory

If you have MS-DOS version 5.0 or later on your system and have followed recommendations in the section on HIMEM.SYS, you should have MS-DOS running in extended memory. The MS-DOS (version 5.0 or later only) SETUP program installs MS-DOS so that it runs in the first 64K of extended memory, called the "high memory area" (HMA). HIMEM.SYS or another XMS driver must be loaded before you can load MS-DOS into the high memory area.

There are several other ways to free conventional memory:

- Run device drivers and other memory-resident programs in upper memory (see pages 17 and 24).
- Don't start unnecessary memory-resident programs. Learn the purpose of each statement in your CONFIG.SYS and AUTOEXEC.BAT files so you know what programs are loaded (see your MS-DOS documentation).
- Include DEVICE commands in CONFIG.SYS only for device drivers you really need. If your system has expanded-memory hardware, include a DEVICE command for the expanded-memory manager that comes with the memory board. Configure memory as extended, not expanded, if possible.

In some situations involving the /Zi option, MASM requires 500Kb of free conventional memory for a successful assembly. This can be resolved by assembling and linking in separate steps (that is, by using NMAKE). For more information on the /Zi option, see *Programmer's Guide*. For more information on using NMAKE, see *Environment and Tools*.

Enabling Extended Memory with HIMEM.SYS

HIMEM.SYS, a memory manager provided in the MASM package, is required by MASM version 6.1. This program provides access to extended memory and ensures that two programs do not use the same part of extended memory at the same time. HIMEM.SYS conforms to the Lotus/Intel/Microsoft (LIM) Extended Memory Standard (XMS), version 2.0.

If you do not have a memory manager installed, SETUP adds the following statement to your NEW-CONF.SYS file and copies HIMEM.SYS to your C:\MASM61\BIN subdirectory:

```
DEVICE=C:\MASM61\BIN\HIMEM.SYS
```

The DEVICE command for HIMEM.SYS enables the use of extended memory. This command must be included in your CONFIG.SYS file before any commands that start device drivers, or any programs that use extended memory such as RAMDRIVE.SYS and EMM386.EXE.

Note Some systems require the use of the HIMEM.SYS /m switch. Check your computer's operations guide and MS-DOS manual for details.

To access the maximum amount of conventional memory, use high memory and upper memory as much as possible. To move MS-DOS out of conventional memory and to enable access to upper memory (if you have MS-DOS version 5.0 or later), your CONFIG.SYS file needs the following statements**:**

```
DEVICE=C:\MASM61\BIN\HIMEM.SYS
DOS=HIGH,UMB
```

This loads MS-DOS into high memory (HMA), enables the use of upper memory, and lets you load device drivers and TSRs into HMA, thus keeping a large amount of conventional memory available.

Note If you are not using a Microsoft memory manager, you may need to remove HIMEM.SYS from your system. Follow the manufacturer's instructions.

Freeing Extended Memory

If you are having difficulty running a program that requires additional extended memory, use the MEM command or MSD.EXE to make sure your system has enough extended memory to run the program. See MSD.TXT for a description of MSD.EXE. To minimize use of extended memory:

- Make sure your CONFIG.SYS and AUTOEXEC.BAT files are not loading unnecessary programs or device drivers that are using extended memory. To verify what a particular device driver does, see your MS-DOS or Windows operating system documentation.

- Reduce the amount of extended memory you allocate for each device driver by modifying the DEVICE command for each.

Freeing Expanded Memory

If a program does not run because there is not enough expanded memory, first make sure that your system contains as much physical expanded memory as the program needs. Also, make sure that it has enough extended memory to emulate expanded memory by checking your memory layout with the MEM command. Then check that your CONFIG.SYS and AUTOEXEC.BAT files aren't starting unnecessary programs that use expanded memory. If you want to free more expanded memory, try the following:

- Use EMM386.EXE to provide more expanded memory.
- Check that the devices loaded with the DEVICE command in your CONFIG.SYS file aren't using all of your expanded memory. Then reduce the amount of expanded memory being allocated, or disable unnecessary DEVICE commands.

Using EMM386.EXE as an Expanded Memory Emulator

The EMM386.EXE device driver included with MASM can use extended memory to emulate expanded memory on 80386 and 80486 systems. EMM386.EXE can also function as an upper memory manager (see page 36). EMM386.EXE can be run under MS-DOS version 3.*x* or later. Expanded memory boards and managers conform to the Lotus/Intel/Microsoft (LIM) Expanded Memory Standard (EMS) version 3.2 or 4.0.

Note Do not use EMM386.EXE with an expanded memory manager from another manufacturer.

EMM386.EXE provides expanded memory for systems that have only extended memory. EMM386.EXE requires about 80K of extended memory, in addition to the memory used to emulate expanded memory. Thus expanded memory is provided at the cost of extended memory.

The typical NEW-CONF.SYS statement for EMM386.EXE is:

```
DEVICE=C:\MASM61\BIN\EMM386.EXE NOEMS
```

If you specify the NOEMS option, CodeView and CVPACK will not run outside of the Windows operating system. If you want to run MS-DOS-extended programs

both within and outside of the Windows operating system, change the DEVICE statement in your CONFIG.SYS file as follows:

```
DEVICE=C:\MASM61\BIN\EMM386.EXE 2048 RAM
```

This statement installs EMM386.EXE with an allocation of 2048K memory. Make sure this command comes after the DEVICE command for HIMEM.SYS and before any commands for device drivers that use the high memory area (device drivers loaded with the DEVICEHIGH command).

If you are currently using EMM386.SYS, an older version of this memory manager, replace it with EMM386.EXE and adjust the filename in your CONFIG.SYS file.

You can exclude certain blocks or ranges of memory from the memory you allocate to EMM386.EXE. This prevents EMM386.EXE from using ranges of memory reserved for device drivers (such as hard disk cards, net cards, or video display) that use memory above 1 MB.

Use the `x=` option for this. The following example excludes the memory for a hard card or a net card. (The exact memory locations will vary, depending on your computer and the memory cards you have installed.)

```
DEVICE=C:\MASM61\BIN\EMM386.EXE RAM X=C000-CDFF X=D800-DFFF
```

Make sure that the memory area you specify is the correct range of addresses for your system before enabling this statement in your CONFIG.SYS file.

Note Use EMM386.EXE options carefully. Always save a copy of your CONFIG.SYS file when experimenting with the `x=` option.

You can use the MEM command to determine the hex locations and sizes of the device drivers you have loaded. This will enable you to fine tune the *X*= setting for EMM386.EXE. (For information on the MEM command, see page 32 or your
MS-DOS Reference.)

Using EMM386.EXE to Manage Upper Memory

Upper memory is the region of your 80386 or 80486 computer's memory that is used by the system. Parts of upper memory that are not used are called "upper memory blocks" (UMB). You can use UMBs for running device drivers and other memory-resident programs that make more conventional memory available for running programs. The MS-DOS upper memory manager is

EMM386.EXE. You need to use MS-DOS 4.*x* or higher for EMM386.EXE to access upper memory.

Some programs cannot be moved to upper memory. These include HIMEM.SYS, EMM386.EXE, and MS-DOS system data.

Programs such as DOSKEY, SHARE, FASTOPEN, RAMDRIVE.SYS, console and other device drivers are good choices for loading into upper memory. You can also load your TSRs into high memory if your MS-DOS version supports it.

Note The only way to find out if a program can run in upper memory is to try it. Some programs do not run properly in upper memory. If the program does not execute correctly, or if the system locks up, run it in conventional memory.

To run programs in upper memory, you must include the following commands in your CONFIG.SYS file for loading HIMEM.SYS and EMM386.EXE:

```
DEVICE=C:\MASM61\BIN\HIMEM.SYS
DEVICE=C:\MASM61\BIN\EMM386.EXE NOEMS
```

The DEVICE command for EMM386.EXE installs EMM386.EXE as an upper memory manager. The NOEMS option tells MS-DOS to run EMM386.EXE to manage upper memory only. Since NOEMS prevents EMM386.EXE from emulating expanded memory, use NOEMS only if your programs do not require expanded memory.

The NOEMS option for EMM386.EXE is the most efficient setting for the Windows operating system, but MS-DOS-extended programs in MASM (CodeView, CVPACK, and LINK) fail if you run them from MS-DOS without the Windows operating system.

Use this command:

```
DEVICE=C:\DOS\EMM386.EXE RAM
```

if you want to use EMM386.EXE both for the upper memory area manager and to emulate expanded memory.

Note The Microsoft Windows operating system will be unable to allocate expanded memory to programs that need it if you specify the NOEMS option when installing EMM386.EXE. If you use such programs, use the RAM option (or no options) instead.

Put the DEVICE command for HIMEM.SYS before the DEVICE command for EMM386.EXE. The DEVICE commands for HIMEM.SYS and EMM386.EXE must appear before any other DEVICE commands.

If MS-DOS runs in upper memory, your CONFIG.SYS file will have DOS=HIGH,UMB instead of DOS=UMB.

To load programs into upper memory, check your memory layout by executing the MEM command. At the end of the output from the MEM /c command, note the size in the line Largest available upper memory block. Then look in the "Conventional Memory" section of the output and find the largest device driver or program that will fit into that upper memory block (UMB). Change the command in the CONFIG.SYS file for that device driver from DEVICE to DEVICEHIGH. For memory-resident programs, change the command in the AUTOEXEC.BAT file from LOAD to LOADHIGH. Do this for one program at a time. You must restart your system each time.

If you get an error with one of the programs you have loaded into upper memory,
or the program or device driver is still running in conventional memory after you restart your system, it may be that the largest UMB is not large enough. Some programs require more memory when they are loaded than when they are running. Try using the SIZE= option with the DEVICEHIGH command (see the information in your MS-DOS documentation on the DEVICE command). Modify the DEVICEHIGH command in your CONFIG.SYS file to specify the hexadecimal size of the driver from the Size in Hex column of the MEM output, and restart your computer. For example, if the information in the Size in Hex column from the MEM command output for MOUSE.SYS is 39E0, you would put this statement in your CONFIG.SYS:

```
DEVICEHIGH SIZE=39E0 C:\WIN3\MOUSE.SYS
```

The SIZE= option takes effect only if needed. If using the SIZE= option doesn't allow your program to run, or if your system locks up during startup or when running the program, it is likely that the program cannot run in upper memory. Change the DEVICEHIGH command to DEVICE and remove LOADHIGH commands one at a time until the program works correctly.

Some hardware programs might attempt to use upper memory after EMM386.EXE has determined this memory is available for running device drivers and programs. To avoid this, you can use the *x*= option when you load EMM386.EXE. This option prevents EMM386.EXE from allocating a specified range of upper memory for its use. For example, to prevent EMM386.EXE

from using the addresses D800h through DFFFh for UMB, you can include the following command in your CONFIG.SYS file:

```
DEVICE=C:\DOS\EMM386.EXE NOEMS x=D800-DFFF
```

If you think your computer is set up correctly to run device drivers and programs in upper memory, but nothing appears there when you use the MEM /c command, check the following:

- Make sure you are not running the Windows operating system version 3.*x* in 386-Enhanced mode when you execute the MEM command. The MEM command does not report the contents of upper memory when you are running the Windows operating system.
- Your CONFIG.SYS file must contain the DOS=UMB or DOS=HIGH,UMB command.
- The DEVICE command for EMM386.EXE in your CONFIG.SYS file must contain the NOEMS or RAM option. RAM is the default.
- Your CONFIG.SYS file must contain a DEVICEHIGH command, or your AUTOEXEC.BAT file must contain the LOADHIGH command for each program you want to run in upper memory.
- The DEVICE command for HIMEM.SYS must appear before the DEVICE command for EMM386.EXE; the DEVICE command for EMM386.EXE must appear before any DEVICEHIGH command in your CONFIG.SYS file.

Once programs are working successfully in upper memory, you can experiment to find the most efficient way to use available memory.

In general, load device drivers and programs in order of size, from largest to smallest. Do this because MS-DOS uses the largest remaining UMB, even if that program would fit into a smaller UMB. The optimal loading order depends on the sizes of programs you are loading and the sizes of available UMB.

Other DPMI Servers

Though MASM version 6.1 will make use of any memory allocated by a DPMI or VCPI server. it does not require the presence of a DPMI server. MASM does require an XMS server, such as HIMEM.SYS.

Optimization Summary

Table 3.3, Summary of Optimization Methods, lists the different optimization methods and how they are used.

Table 3.3 Summary of Optimizing Methods

Method	When to Use	Memory Used
Use HIMEM.SYS.	Required for MASM version 6.1.	Conventional
Run MS-DOS in extended memory.	If your system has extended memory.	High memory (HMA)
Use EMM386.EXE as an expanded memory emula-tor.	If your system has extended memory and your programs need expanded memory.	Extended and conventional
Make sure CONFIG.SYS and AUTOEXEC.BAT are not loading unnecessary programs or device drivers.	To free memory.	Depends on the programs you remove
Run device drivers and programs in upper memory	To free memory.	Upper memory
Use SMARTDRV.EXE. (Don't use with a secondary buffer cache.)	If your system has extended or expanded memory that isn't needed by programs.	All programs
Use RAMDRIVE.SYS.	If your system has extended or expanded memory and you use programs that RAMDRIVE.SYS optimize.	Programs that use tempo-rary files or programs that you run often

Reference

Microsoft® MASM

Assembly-Language Development System
Version 6.1

For MS-DOS® and Windows™ Operating System

Microsoft Corporation

Information in this document is subject to change without notice. Companies, names, and data used in examples herein are fictitious unless otherwise noted. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Microsoft Corporation.

© 1987, 1991, 1992 Microsoft Corporation. All rights reserved.

Microsoft, MS, MS-DOS, XENIX, CodeView, and QuickC are registered trademarks and Windows and Windows NT are trademarks of Microsoft Corporation in the USA and other countries.

U.S. Patent No. 4955066

IBM is a registered trademark of International Business Machines Corporation.
Intel is a registered trademark and 386, 387, 486 are trademarks of Intel Corporation.
Timings and encodings in this manual are used with permission of Intel and come from the following publications:
Intel Corporation, iAPX 86, 88, 186, and 188 *User's Manual*, *Programmer's Reference*. Santa Clara, Calif. 1985.
Intel Corporation, *iAPX 286 Programmer's Reference Manual* including the *iAPX 286 Numeric Supplement*. Santa Clara, Calif. 1985.
Intel Corporation. *80386 Programmer's Reference Manual*. Santa Clara, Calif. 1986.
Intel Corporation. *80387 80-bit CHMOS III Numeric Processor Extension*. Santa Clara, Calif. 1987.
Intel Corporation. *i486 Microprocessor Data Sheet*. Santa Clara, Calif. 1989.

Document No. DB35749-1292
Printed in the United States of America.

Contents

Introduction ix

Document Conventions x

Chapter 1 Tools 1
Microsoft® CodeView® Debugger 2
CVPACK 3
EXEHDR 3
EXP 4
HELPMAKE 4
H2INC 6
IMPLIB 7
LIB 7
LINK 8
MASM 11
ML 12
NMAKE 14
PWB (Programmer's WorkBench) 16
PWBRMAKE 17
QuickHelp 18
RM 19
UNDEL 20

Chapter 2 Directives 21
Topical Cross-reference for Directives 22
Directives 25

Chapter 3 Symbols and Operators 39
Topical Cross-reference for Symbols 40
Topical Cross-reference for Operators 41
Predefined Symbols 43
Operators 44
Run-Time Operators 48

Chapter 4 Processor 49
Topical Cross-reference for Processor Instructions 50
Interpreting Processor Instructions 53
Flags 53

Clock Speeds 54
Timings on the 8088 and 8086 Processors 55
Timings on the 80286–80486 Processors 56
Interpreting Encodings 56
Interpreting 80386/486 Encoding Extensions 59
16-Bit Encoding 60
32-Bit Encoding 60
Address-Size Prefix 60
Operand-Size Prefix 60
Encoding Differences for 32-Bit Operations 60
Scaled Index Base Byte 61
Instructions 64
AAA ASCII Adjust After Addition 64
AAD ASCII Adjust Before Division 64
AAM ASCII Adjust After Multiply 65
AAS ASCII Adjust After Subtraction 65
ADC Add With Carry 66
ADD Add 67
AND Logical AND 68
ARPL Adjust Requested Privilege Level 69
BOUND Check Array Bounds 69
BSF/BSR Bit Scan 70
BSWAP Byte Swap 71
BT/BTC/BTR/BTS Bit Tests 72
CALL Call Procedure 73
CBW Convert Byte to Word 74
CDQ Convert Double to Quad 75
CLC Clear Carry Flag 75
CLD Clear Direction Flag 76
CLI Clear Interrupt Flag 76
CLTS Clear Task Switched Flag 76
CMC Complement Carry Flag 77
CMP Compare Two Operands 77
CMPS/CMPSB/CMPSW/CMPSD Compare String 79
CMPXCHG Compare and Exchange 80
CWD Convert Word to Double 80
CWDE Convert Word to Extended Double 81
DAA Decimal Adjust After Addition 81
DAS Decimal Adjust After Subtraction 82
DEC Decrement 82

DIV Unsigned Divide 83
ENTER Make Stack Frame 84
HLT Halt 84
IDIV Signed Divide 85
IMUL Signed Multiply 85
IN Input from Port 87
INC Increment 88
INS/INSB/INSW/INSD Input from Port to String 89
INT Interrupt 89
INTO Interrupt on Overflow 90
INVD Invalidate Data Cache 91
INVLPG Invalidate TLB Entry 91
IRET/IRETD Interrupt Return 92
Jcondition Jump Conditionally 92
JCXZ/JECXZ Jump if CX is Zero 94
JMP Jump Unconditionally 94
LAHF Load Flags into AH Register 96
LAR Load Access Rights 96
LDS/LES/LFS/LGS/LSS Load Far Pointer 97
LEA Load Effective Address 98
LEAVE High Level Procedure Exit 99
LES/LFS/LGS Load Far Pointer to Extra Segment 99
LGDT/LIDT/LLDT Load Descriptor Table 99
LMSW Load Machine Status Word 100
LOCK Lock the Bus 101
LODS/LODSB/LODSW/LODSD Load Accumulator from String 101
LOOP/LOOPW/LOOPD Loop 102
LOOPcondition/LOOPconditionW/LOOPconditionD Loop Conditionally . 102
LSL Load Segment Limit 103
LSS Load Far Pointer to Stack Segment 104
LTR Load Task Register 104
MOV Move Data 105
MOV Move to/from Special Registers 106
MOVS/MOVSB/MOVSW/MOVSD Move String Data 108
MOVSX Move with Sign-Extend 108
MOVZX Move with Zero-Extend 109
MUL Unsigned Multiply 109
NEG Two’s Complement Negation 110
NOP No Operation 111
NOT One’s Complement Negation 111

OR Inclusive OR 112
OUT Output to Port 113
OUTS/OUTSB/OUTSW/OUTSD Output String to Port 113
POP Pop 114
POPA/POPAD Pop All 115
POPF/POPFD Pop Flags 116
PUSH/PUSHW/PUSHD Push 116
PUSHA/PUSHAD Push All 117
PUSHF/PUSHFD Push Flags 118
RCL/RCR/ROL/ROR Rotate 118
REP Repeat String 120
REPcondition Repeat String Conditionally 122
RET/RETN/RETF Return from Procedure 123
ROL/ROR Rotate 124
SAHF Store AH into Flags 124
SAL/SAR Shift 125
SBB Subtract with Borrow 125
SCAS/SCASB/SCASW/SCASD Scan String Flags 126
SETcondition Set Conditionally 127
SGDT/SIDT/SLDT Store Descriptor Table 128
SHL/SHR/SAL/SAR Shift 129
SHLD/SHRD Double Precision Shift 131
SMSW Store Machine Status Word 133
STC Set Carry Flag 134
STD Set Direction Flag 134
STI Set Interrupt Flag 134
STOS/STOSB/STOSW/STOSD Store String Data 135
STR Store Task Register 136
SUB Subtract 136
TEST Logical Compare 137
VERR/VERW Verify Read or Write 138
WAIT Wait 139
WBINVD Write Back and Invalidate Data Cache 140
XADD Exchange and Add 140
XCHG Exchange 141
XLAT/XLATB Translate 141
XOR Exclusive OR 142

Chapter 5 Coprocessor 145
Topical Cross-reference for Coprocessor Instructions 146

Interpreting Coprocessor Instructions 148
F2XM1 2X–1 150
FABS Absolute Value 150
FADD/FADDP/FIADD Add 151
FBLD Load BCD 151
FBSTP Store BCD and Pop 151
FCHS Change Sign 152
FCLEX/FNCLEX Clear Exceptions 152
FCOM/FCOMP/FCOMPP/FICOM/FICOMP Compare 152
FCOS Cosine 154
FDECSTP Decrement Stack Pointer 154
FDISI/FNDISI Disable Interrupts 154
FDIV/FDIVP/FIDIV Divide 155
FDIVR/FDIVRP/FIDIVR Divide Reversed 156
FENI/FNENI Enable Interrupts 156
FFREE Free Register 157
FIADD/FISUB/FISUBR/FIMUL/FIDIV/FIDIVR Integer Arithmetic 157
FICOM/FICOMP Compare Integer 157
FILD Load Integer 157
FINCSTP Increment Stack Pointer 158
FINIT/FNINIT Initialize Coprocessor 158
FIST/FISTP Store Integer 158
FLD/FILD/FBLD Load 159
FLD1/FLDZ/FLDPI/FLDL2E/FLDL2T/FLDLG2/FLDLN2 Load Constant159
FLDCW Load Control Word 161
FLDENV/FLDENVW/FLDENVD Load Environment State 161
FMUL/FMULP/FIMUL Multiply 161
FNinstruction No-Wait Instructions 162
FNOP No Operation 163
FPATAN Partial Arctangent 163
FPREM Partial Remainder 163
FPREM1 Partial Remainder (IEEE Compatible) 164
FPTAN Partial Tangent 165
FRNDINT Round to Integer 165
FRSTOR/FRSTORW/FRSTORD Restore Saved State 166
FSAVE/FSAVEW/FSAVED/FNSAVE/FNSAVEW/FNSAVED
Save Coprocessor State 166
FSCALE Scale 167
FSETPM Set Protected Mode 167
FSIN Sine 168

FSINCOS Sine and Cosine 168
FSQRT Square Root. 169
FST/FSTP/FIST/FISTP/FBSTP Store. 169
FSTCW/FNSTCW Store Control Word. 170
FSTENV/FSTENVW/FSTENVD/FNSTENV/FNSTENVW/FNSTENVD
Store Environment State 170
FSTSW/FNSTSW Store Status Word 171
FSUB/FSUBP/FISUB Subtract 171
FSUBR/FSUBRP/FISUBR Subtract Reversed. 172
FTST Test for Zero 173
FUCOM/FUCOMP/FUCOMPP Unordered Compare 173
FWAIT Wait. 174
FXAM Examine. 175
FXCH Exchange Registers. 176
FXTRACT Extract Exponent and Significand. 176
FYL2X Y log2(X) 176
FYL2XP1 Y log2(X+1) 177

Chapter 6 Macros. 179
Introduction. 180
BIOS.INC 180
CMACROS.INC, CMACROS.NEW 180
MS-DOS.INC. 183
MACROS.INC. 184
PROLOGUE.INC 185
WIN.INC. 185

Chapter 7 Tables 187
ASCII Codes. 188
Key Codes. 190
MS-DOS Program Segment Prefix (PSP) 192
Color Display Attributes. 193
Hexadecimal-Binary-Decimal Conversion 194

Introduction

This Microsoft ® Macro Assembler *Reference* lists all MASM instructions, directives, statements, and operators. It also serves as a quick reference to the Programmer's WorkBench commands, and the commands for Microsoft utilities such as LINK and LIB. This book documents features of MASM version 6.1, and
is part of a complete MASM documentation set. Other titles in the set are:

Getting Started—Explains how to perform all the tasks necessary to install and begin running MASM 6.1 on your system.

Environment and Tools—Describes the development tools that are included with MASM 6.1: the Programmer's WorkBench, CodeView debugger, LINK, EXEHDR, NMAKE, LIB, and other tools and utilities. A detailed tutorial on the Programmer's WorkBench teaches the basics of creating and debugging MASM code in this full-featured programming environment. A complete list of utilities and error messages generated by ML is also included.

Programmer's Guide—Provides information for experienced assembly-language programmers on the features of the MASM 6.1 language. The appendixes cover the differences between MASM 5.1, MASM 6.0, and MASM 6.1, and the Backus-Naur Form for grammar notation to use in determining the syntax for any MASM language component.

Document Conventions

The following document conventions are used throughout this book:

Example	Description
SAMPLE 2ASM	Uppercase letters indicate filenames, segment names, registers and terms used at the command line.
KEY TERMS	Bold type indicates text that must be typed exactly as shown. This includes assembly-language instructions, directives, symbols, operators, and keywords in other languages.
placeholders	Italics indicate variable information supplied by the user.
`Examples`	This typeface indicates example programs, user input, and screen output.
⟦*optional items*⟧	Double brackets indicate that the enclosed item is optional.
{*choice1* | *choice2*}	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...	Three dots following an item indicate that you may type more items having the same form.
SHIFT+F1	Small capital letters indicate key names.

C H A P T E R 1

Tools

CodeView 2
CVPACK 3
EXEHDR 3
EXP 4
HELPMAKE 4
H2INC 6
IMPLIB 7
LIB 7
LINK 8
MASM 11
ML. 12
NMAKE 14
PWB 16
PWBRMAKE 17
QuickHelp. 18
RM 19
UNDEL 20

Microsoft® CodeView® Debugger

The Microsoft® CodeView® debugger runs the assembled or compiled program while simultaneously displaying the program source code, program variables, memory locations, processor registers, and other pertinent information.

Syntax

CV [[*options*]] *executablefile* [[*arguments*]]

CVW [[*options*]] *executablefile* [[*arguments*]]

Options

Option	Action
/2	Permits the use of two monitors.
/8	Uses 8514/a as Windows display, and VGA as debugger display (CVW only).
/25	Starts in 25-line mode.
/43	Starts in 43-line mode.
/50	Starts in 50-line mode.
/B	Starts in black-and-white mode.
/C*commands*	Executes *commands* on startup.
/F	Exchanges screens by flipping between video pages (CV only).
/G	Eliminates refresh snow on CGA monitors.
/I[[0 | 1]]	Turns nonmaskable-interrupt and 8259-interrupt trapping on (/I1) or off (/I0).
/L*dllfile*	Loads DLL *dllfile* for debugging (CVW only).
/K	Disables installation of keyboard monitors for the program being debugged (CV only).
/M	Disables CodeView use of the mouse. Use this option when debugging an application that supports the mouse.
/N[[0 | 1]]	/N0 tells CodeView to trap nonmaskable interrupts; /N1 tells it not to trap.
/R	Enables 80386/486 debug registers (CV only).
/S	Exchanges screens by changing buffers (primarily for use with graphics programs) (CV only).
/TSF	Toggles TOOLS.INI entry to read/not read the CURRENT.STS file.

Environment Variables

Variable	Description
HELPFILES	Specifies path of help files or list of help filenames.
INIT	Specifies path for TOOLS.INI and CURRENT.STS files.

CVPACK

The CVPACK utility reduces the size of an executable file that contains CodeView debugging information.

Syntax

CVPACK [[*options*]] *exefile*

Options

Option	Action
/HELP	Calls QuickHelp for help on CVPACK.
/P	Packs the file to the smallest possible size.
/?	Displays a summary of CVPACK command-line syntax.

EXEHDR

The EXEHDR utility displays and modifies the contents of an executable-file header.

Syntax

EXEHDR [[*options*]] *filenames*

Options

Option	Action
/HEA:*number*	Option name: /HEA[[P]]. Sets the heap allocation field to *number* bytes for segmented-executable files.
/HEL	Option name: /HEL[[P]]. Calls QuickHelp for help on EXEHDR.
/MA:*number*	Option name: /MA[[X]]. Sets the maximum memory allocation to *number* paragraphs for DOS executable files.
/MI:*number*	Option name: /MI[[N]]. Sets the minimum memory allocation to *number* paragraphs for DOS executable files.
/NE	Option name: /NE[[WFILES]]. Enables support for HPFS.
/NO	Option name: /NO[[LOGO]]. Suppresses the EXEHDR copyright message.
/PM:*type*	Option name: /PM[[TYPE]]. Sets the application type for Microsoft Windows®, where *type* is one of the following: **PM** (or **WINDOWAPI**), **VIO** (or **WINDOWCOMPAT**), or **NOVIO** (or **NOTWINDOWCOMPAT**).
/R	Option name: /R[[ESETERROR]]. Clears the error bit in the header of a Windows executable file.
/S:*number*	Option name: /S[[TACK]]. Sets the stack allocation to *number* bytes.

Option	Action
/V	Option name: /V[[ERBOSE]]. Provides more information about segmented-executable files, including the default flags in the segment table, all run-time relocations, and additional fields from the header.
/?	Option name: /?. Displays a summary of EXEHDR command-line syntax.

EXP

The EXP utility deletes all files in the hidden DELETED subdirectory of the current or specified directory. EXP is used with RM and UNDEL to manage backup files.

Syntax

EXP [[*options*]] [[*directories*]]

Options

Option	Action
/HELP	Calls QuickHelp for help on EXP.
/Q	Suppresses display of deleted files.
/R	Recurses into subdirectories of the current or specified directory.
/?	Displays a summary of EXP command-line syntax.

HELPMAKE

The HELPMAKE utility creates help files and customizes the help files supplied with Microsoft language products.

Syntax

HELPMAKE {/E[[*n*]] | /D[[*c*]] | /H | /?} [[*options*]] *sourcefiles*

Options

Option	Action
/A*c*	Specifies *c* as an application-specific control character for the help database, marking a line that contains special information for internal use by the application.
/C	Indicates that the context strings are case sensitive so that at run time all searches for help topics are case sensitive.
/D	Fully decodes the help database.

Option	**Action**
/DS	Splits the concatenated, compressed help database into its components, using their original names. No decompression occurs.
/DU	Decompresses the database and removes all screen formatting and cross-references.
/E[*n*]	Creates ("encodes") a help database from a specified text file (or files). The optional *n* indicates the amount of compression to take place. The value of *n* can range from 0 to 15.
/H[ELP]	Calls the QuickHelp utility. If HELPMAKE cannot find QuickHelp or the help file, it displays a summary of HELPMAKE command-line syntax.
/K*filename*	Specifies a file containing word-separator characters. This file must contain a single line of characters that separate words. ASCII characters from 0 to 32 (including the space) and character 127 are always separators. If the /K option is not specified, the following characters are also considered separators: !"#&'()*+-,/:;<=>?@[\]^_`{\}~
/L	Locks the generated file so that it cannot be decoded by HELPMAKE at a later time.
/NOLOGO	Suppresses the HELPMAKE copyright message.
/O*outfile*	Specifies *outfile* as the name of the help database. The name *outfile* is optional with the /D option.
/S*n*	Specifies the type of input file, according to the following values for *n*: /S1 Rich Text Format /S2 QuickHelp Format /S3 Minimally Formatted ASCII
/T	During encoding, translates dot commands to application-specific commands. During decoding, translates application commands to dot commands. The /T option forces /A:.
/V[*n*]	Sets the verbosity of the diagnostic and informational output, depending on the value of *n*. The value of *n* can range from 0 to 6.
/W*width*	Sets the fixed width of the resulting help text in number of characters. The value of *width* can range from 11 to 255.
/?	Displays a summary of HELPMAKE command-line syntax.

H2INC

The H2INC utility converts C header (.H) files into MASM-compatible include (.INC) files. It translates declarations and prototypes, but does not translate code.

Syntax

H2INC [[*options*]] *filename*.H

Options

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.
/Fc[[*filename*]]	Specifies that the output file contain equivalent MASM statements plus original C statements converted to comment lines.
/HELP	Calls QuickHelp for help on H2INC.
/Ht	Enables generation of text equates. By default, text items are not translated.
/Mn	Instructs H2INC to explicitly declare the distances for all pointers and functions.
/Ni	Suppresses the expansion of nested include files.
/Zn *string*	Adds *string* to all names generated by H2INC. Used to eliminate name conflicts with other H2INC-generated include files.
/Zu	Makes all structure and union tag names unique.
/?	Displays a summary of H2INC command-line syntax.

*H2INC also supports the following options from Microsoft C, version 6.0 and higher: /AC, /AH, /AL, /AM, /AS, /AT, /D, /F, /Fi, /G0, /G1, /G2, /G3, /G4, /Gc, /Gd, /Gr, /I, /J, /Tc, /U, /u, /W0, /W1, /W2, /W3, /W4, /X, /Za, /Zc, /Ze, /Zp1, /Zp2, /Zp4.

Environment Variables

Variable	Description
CL	Specifies default command-line options.
H2INC	Specifies default command-line options. Appended after the CL environment variable.
INCLUDE	Specifies search path for include files.

IMPLIB

The IMPLIB utility creates import libraries used by LINK to link dynamic-link libraries with applications.

Syntax

IMPLIB [[*options*]] *implibname* {*dllfile*... | *deffile*...}

Options

Option	Action
/H	Option name: /H[[ELP]]. Calls QuickHelp for help on IMPLIB.
/NOI	Option name: /NOI[[GNORECASE]]. Preserves case for entry names in DLLs.
/NOL	Option name: /NOL[[OGO]]. Suppresses the IMPLIB copyright message.
/?	Option name: /?. Displays a summary of IMPLIB command-line syntax.

LIB

The LIB utility helps create, organize, and maintain run-time libraries.

Syntax

LIB *inlibrary* [[*options*]] [[*commands*]] [[**,** [[*listfile*]] [[**,** [[*outlibrary*]]]]]] [[**;**]]

Options

Option	Action
/H	Option name: /H[[ELP]]. Calls QuickHelp for help on LIB.
/I	Option name: /I[[GNORECASE]]. Tells LIB to ignore case when comparing symbols (the default). Use to combine a library marked /NOI with an unmarked library to create a new case-insensitive library.
/NOE	Option name: NOE[[XTDICTIONARY]]. Prevents LIB from creating an extended dictionary.
/NOI	Option name: /NOI[[GNORECASE]]. Tells LIB to preserve case when comparing symbols. When combining libraries, if any library is marked /NOI, the output library is case sensitive, unless /IGN is specified.
/NOL	Option name: /NOL[[OGO]]. Suppresses the LIB copyright message.

Option	Action
/P:*number*	Option name: /P[[AGESIZE]]. Specifies the page size (in bytes) of a new library or changes the page size of an existing library. The default for a new library is 16.

/?	Option name: /?. Displays a summary of LIB command-line syntax.

Commands

Operator	Action
+*name*	Appends an object file or library file.
–*name*	Deletes a module.
–+*name*	Replaces a module by deleting it and appending an object file with the same name.
*****name*	Copies a module to a new object file.
–*****name*	Moves a module out of the library by copying it to a new object file and then deleting it.

LINK

The LINK utility combines object files into a single executable file or dynamic-link library.

Syntax

LINK *objfiles* [[, [[*exefile*]] [[, [[*mapfile*]] [[, [[*libraries*]] [[, [[*deffile*]]]]]]]]]] [[;]]

Options

Option	Action
/A:*size*	Option name: /A[[LIGNMENT]]. Directs LINK to align segment data in a segmented-executable file along the boundaries specified by *size* bytes, where *size* must be a power of two.
/B	Option name: /B[[ATCH]]. Suppresses prompts for library or object files not found.
/CO	Option name: /CO[[DEVIEW]]. Adds symbolic data and line numbers needed by the Microsoft CodeView debugger. This option is incompatible with the /EXEPACK option.
/CP:*number*	Option name: /CP[[ARMAXALLOC]]. Sets the program's maximum memory allocation to *number* of 16-byte paragraphs.
/DO	Option name: /DO[[SSEG]]. Orders segments in the default order used by Microsoft high-level languages.

Option	Action
/DS	Option name: /DS[ALLOCATE]. Directs LINK to load all data starting at the high end of the data segment. The /DSALLOC option is for assembly-language programs that create MS-DOS .EXE files.
/E	Option name: /E[XEPACK]. Packs the executable file. The /EXEPACK option is incompatible with /INCR and /CO. Do not use /EXEPACK on a Windows-based application.
/F	Option name: /F[ARCALLTRANSLATION]. Optimizes far calls. The /FARCALL option is automatically on when using /TINY. The /PACKC option is not recommended with /FARCALL when linking a Windows-based program.
/HE	Option name: /HE[LP]. Calls QuickHelp for help on LINK.
/HI	Option name: /HI[GH]. Places the executable file as high in memory as possible. Use /HIGH with the /DSALLOC option. This option is for assembly-language programs that create MS-DOS .EXE files.
/INC	Option name: /INC[REMENTAL]. Prepares for incremental linking with ILINK. This option is incompatible with /EXEPACK and /TINY.
/INF	Option name: /INF[ORMATION]. Displays to the standard output the phase of linking and names of object files being linked.
/LI	Option name: /LI[NENUMBERS]. Adds source file line numbers and associated addresses to the map file. The object file must be created with line numbers. This option creates a map file even if *mapfile* is not specified.
/M	Option name: /M[AP]. Adds public symbols to the map file.
/NOD[:*libraryname*]	Option name: /NOD[EFAULTLIBRARYSEARCH]. Ignores the specified default library. Specify without *libraryname* to ignore all default libraries.
/NOE	Option name: /NOE[XTDICTIONARY]. Prevents LINK from searching extended dictionaries in libraries. Use /NOE when redefinition of a symbol causes error L2044.
/NOF	Option name: /NOF[ARCALLTRANSLATION]. Turns off far-call optimization.
/NOI	Option name: /NOI[GNORECASE]. Preserves case in identifiers.
/NOL	Option name: /NOL[OGO]. Suppresses the LINK copyright message.

Option	Action
/NON	Option name: /NON[ULLSDOSSEG]. Orders segments as with the /DOSSEG option, but with no additional bytes at the beginning of the _TEXT segment (if defined). This option overrides /DOSSEG.
/NOP	Option name: /NOP[ACKCODE]. Turns off code segment packing.
/PACKC[:*number*]	Option name: /PACKC[ODE]. Packs neighboring code segments together. Specify *number* bytes to set the maximum size for physical segments formed by /PACKC.
/PACKD[:*number*]	Option name: /PACKD[ATA]. Packs neighboring data segments together. Specify *number* bytes to set the maximum size for physical segments formed by /PACKD. This option is for Windows only.
/PAU	Option name: /PAU[SE]. Pauses during the link session for disk changes.
/PM:*type*	Option name: /PM[TYPE]. Specifies the type of Windows-based application where *type* is one of the following: **PM** (or **WINDOWAPI**), **VIO** (or **WINDOWCOMPAT**), or **NOVIO** (or **NOTWINDOWCOMPAT**).
/ST:*number*	Option name: /ST[ACK]. Sets the stack size to *number* bytes, from 1 byte to 64K.
/T	Option name: /T[INY]. Creates a tiny-model MS-DOS program with a .COM extension instead of .EXE. Incompatible with /INCR.
/?	Option name: /?. Displays a summary of LINK command-line syntax.

Note Several rarely used options not listed here are described in Help.

Environment Variables

Variable	Description
INIT	Specifies path for the TOOLS.INI file.
LIB	Specifies search path for library files.
LINK	Specifies default command-line options.
TMP	Specifies path for the VM.TMP file.

MASM

The MASM program converts command-line options from MASM style to ML style, adds options to maximize compatibility, and calls ML.EXE.

Note MASM.EXE is provided to maintain compatibility with old makefiles. For new makefiles, use the more powerful ML driver.

Syntax

MASM [[*options*]] *sourcefile* [[, [[*objectfile*]] [[, [[*listingfile*]] [[, [[*crossreferencefile*]]]]]]]] [[;]]

Options

Option	Action
/A	Orders segments alphabetically. Results in a warning. Ignored.
/B	Sets internal buffer size. Ignored.
/C	Creates a cross-reference file. Translated to /FR.
/D	Creates a Pass 1 listing.Translated to F1/ST.
/D*symbol*[[=*value*]]	Defines a symbol. Unchanged.
/E	Emulates floating-point instructions. Translated to /FPi.
/H	Lists command-line arguments. Translated to /help.
/HELP	Calls QuickHelp for help on the MASM driver.
/I *pathname*	Specifies an include path. Unchanged.
/L	Creates a normal listing. Translated to /Fl.
/LA	Lists all. Translated to /Fl and /Sa.
/ML	Treats names as case sensitive. Translated to /Cp.
/MU	Converts names to uppercase. Translated to /Cu.
/MX	Preserves case on nonlocal names. Translated to /Cx.
/N	Suppresses table in listing file. Translated to /Sn.
/P	Checks for impure code. Use **OPTION READONLY**. Ignored.
/S	Orders segments sequentially. Results in a warning. Ignored.
/T	Enables terse assembly. Translated to /NOLOGO.
/V	Enables verbose assembly. Ignored.

Option	Action
/W*level*	Sets warning level, where *level* = 0, 1, or 2.
/X	Lists false conditionals. Translated to /Sx.
/Z	Displays error lines on screen. Ignored.
/ZD	Generates line numbers for CodeView. Translated to /Zd.
/ZI	Generates symbols for CodeView. Translated to /Zi.

Environment Variables

Variable	Description
INCLUDE	Specifies default path for .INC files.
MASM	Specifies default command-line options.
TMP	Specifies path for temporary files.

ML

The ML program assembles and links one or more assembly-language source files. The command-line options are case sensitive.

Syntax

ML [[*options*]] *filename* [[[[*options*]] *filename*]]... [[/link *linkoptions*]]

Options

Option	Action
/AT	Enables tiny-memory-model support. Enables error messages for code constructs that violate the requirements for .COM format files. Note that this is not equivalent to the **.MODEL TINY** directive.
/Bl *filename*	Selects an alternate linker.
/c	Assembles only. Does not link.
/Cp	Preserves case of all user identifiers.
/Cu	Maps all identifiers to uppercase (default).
/Cx	Preserves case in public and extern symbols.
/D*symbol*[[=*value*]]	Defines a text macro with the given name. If *value* is missing, it is blank. Multiple tokens separated by spaces must be enclosed in quotation marks.
/EP	Generates a preprocessed source listing (sent to STDOUT). See /Sf.
/F*hexnum*	Sets stack size to *hexnum* bytes (this is the same as /link /STACK:*number*). The value must be expressed in hexadecimal notation. There must be a space between /F and *hexnum*.

Option	**Action**
/Fe*filename*	Names the executable file.
/F[[*filename*]]	Generates an assembled code listing. See /Sf.
/Fm[[*filename*]]	Creates a linker map file.
/Fo*filename*	Names an object file.
/FPi	Generates emulator fixups for floating-point arithmetic (mixed-language only).
/Fr[[*filename*]]	Generates a Source Browser .SBR file.
/FR[[*filename*]]	Generates an extended form of a Source Browser .SBR file.
/Gc	Specifies use of FORTRAN- or Pascal-style function calling and naming conventions. Same as **OPTION LANGUAGE:PASCAL**.
/Gd	Specifies use of C-style function calling and naming conventions. Same as **OPTION LANGUAGE:C**.
/H *number*	Restricts external names to *number* significant characters. The default is 31 characters.
/help	Calls QuickHelp for help on ML.
/I *pathname*	Sets path for include file. A maximum of 10 /I options is allowed.
/nologo	Suppresses messages for successful assembly.
/Sa	Turns on listing of all available information.
/Sc	Adds instruction timings to listing file.
/Sf	Adds first-pass listing to listing file.
/Sg	Turns on listing of assembly-generated code.
/Sl *width*	Sets the line width of source listing in characters per line. Range is 60 to 255 or 0. Default is 0. Same as **PAGE** *width.*
/Sn	Turns off symbol table when producing a listing.
/Sp *length*	Sets the page length of source listing in lines per page. Range is 10 to 255 or 0. Default is 0. Same as **PAGE** *length.*
/Ss *text*	Specifies *text* for source listing. Same as **SUBTITLE** *text*.
/St *text*	Specifies title for source listing. Same as **TITLE** *text*.
/Sx	Turns on false conditionals in listing.
/Ta *filename*	Assembles source file whose name does not end with the .ASM extension.
/w	Same as /W0.
/W*level*	Sets the warning level, where *level* = 0, 1, 2, or 3.

Option	Action
/WX	Returns an error code if warnings are generated.
/Zd	Generates line-number information in object file.
/Zf	Makes all symbols public.
/Zi	Generates CodeView information in object file.
/Zm	Enables **M510** option for maximum compatibility with MASM 5.1.
/Zp[*alignment*]	Packs structures on the specified byte boundary. The *alignment* may be 1, 2, or 4.
/Zs	Performs a syntax check only.
/?	Displays a summary of ML command-line syntax.

QuickAssembler Support

For compatibility with QuickAssembler makefiles, ML recognizes these options:

Option	Action
/a	Orders segments alphabetically in QuickAssembler. MASM 6.1 uses the **.ALPHA** directive for alphabetical ordering and ignores /a.
/Cl	Equivalent to /Cp.
/Ez	Prints the source for error lines to the screen. MASM 6.1 ignores this option.
/P1	Performs one-pass assembly. MASM 6.1 ignores this option.
/P2	Performs two-pass assembly. MASM 6.1 ignores this option.
/s	Orders segments sequentially. MASM 6.1 uses the **.SEQ** directive for sequential ordering and ignores /s.
/Sq	Equivalent to /Sl0 /Sp0.

Environment Variables

Variable	Description
INCLUDE	Specifies search path for include files.
ML	Specifies default command-line options.
TMP	Specifies path for temporary files.

NMAKE

The NMAKE utility automates the process of compiling and linking project files.

Syntax

NMAKE [[*options*]] [[*macros*]] [[*targets*]]

Options

Option	Action
/A	Executes all commands even if targets are not out-of-date.
/C	Suppresses the NMAKE copyright message and prevents nonfatal error or warning messages from being displayed.
/D	Displays the modification time of each file when the times of targets and dependents are checked.
/E	Causes environment variables to override macro definitions within description files.
/F *filename*	Specifies *filename* as the name of the description file to use. If a dash (–) is entered instead of a filename, NMAKE reads the description file from the standard input device. If /F is not specified, NMAKE uses MAKEFILE as the description file. If MAKEFILE does not exist, NMAKE builds command-line targets using inference rules.
/HELP	Calls QuickHelp for help on NMAKE.
/I	Ignores exit codes from commands in the description file. NMAKE continues executing the rest of the description file despite the errors.
/N	Displays but does not execute commands from the description file.
/NOLOGO	Suppresses the NMAKE copyright message.
/P	Displays all macro definitions, inference rules, target descriptions, and the .**SUFFIXES** list.
/Q	Checks modification times of command-line targets (or first target in the 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.
/R	Ignores inference rules and macros that are predefined or defined in the TOOLS.INI file.
/S	Suppresses display of commands as they are executed.
/T	Changes modification times of command-line targets (or first target in the description file if no command-line targets are specified) to the current time. Only preprocessing commands in the description file are executed. The contents of target files are not modified.
/X *filename*	Sends all error output to *filename*, which can be either a file or a device. If a dash (–) is entered instead of a filename, the error output is sent to the standard output device.
/Z	Internal option for use by the Microsoft Programmer's WorkBench (PWB).
/?	Displays a summary of NMAKE command-line syntax.

Environment Variable

Variable	Description
INIT	Specifies path for TOOLS.INI file, which may contain macros, inference rules, and description blocks.

PWB (Programmer's WorkBench)

The Microsoft Programmer's WorkBench (PWB) provides an integrated environment for developing programs in assembly language. The command-line options are case sensitive.

Syntax

PWB [[*options*]] [[*files*]]

Options

Option	Action
/D[[*init*]]	Prevents PWB from examining initialization files, where *init* is one or more of the following characters: A Disable autoload extensions (including language-specific extensions and Help). S Ignore CURRENT.STS. T Ignore TOOLS.INI. If the /D option does not include an *init* character, it is equivalent to specifying /DAST (all files and extensions ignored).
/e *cmdstr*	Executes the command or sequence of commands at start-up. The entire *cmdstr* argument must be placed in double quotation marks if it contains a space. If *cmdstr* contains literal double quotation marks, place a backslash (\) in front of each double quotation mark. To include a literal backslash in the command string, use double backslashes (\\).
/m *mark*	Moves the cursor to the specified *mark* instead of moving it to the last known position. The mark can be a line number.
/P[[*init*]]	Specifies a program list for PWB to read, where *init* can be: Ffile Read a foreign program list (one not created using PWB). L Read the last program list. Use this option to start PWB in the same state you left it. Pfile Read a PWB program list.
/r	Starts PWB in no-edit mode. Functions that modify files are disallowed.

Option	Action
[[/t]] *file...*	Loads the specified file at startup. The *file* specification can contain wildcards. If multiple *files* are specified, PWB loads only the first file. When the *Exit* function is invoked, PWB saves the current file and loads the next file in the list. Files specified with /t are temporary; PWB does not add them to the file history on the File menu. No other options can follow /t on the command line. Each temporary file must be specified in a separate /t option.
/?	Displays a summary of PWB command-line syntax.

Environment Variables

Variable	Description
HELPFILES	Specifies path of help files or list of help filenames.
INIT	Specifies path for TOOLS.INI and CURRENT.STS files.
TMP	Specifies path for temporary files.

PWBRMAKE

PWBRMAKE converts the .SBR files created by the assembler into database .BSC files that can be read by the Microsoft Programmer's WorkBench (PWB) Source Browser. The command-line options are case sensitive.

Syntax

PWBRMAKE [[*options*]] *sbrfiles*

Options

Option	Action
/Ei *filename* /Ei (*filename...*)	Excludes the contents of the specified include files from the database. To specify multiple filenames, separate them with spaces and enclose the list in parentheses.
/Em	Excludes symbols in the body of macros. Use /Em to include only macro names.
/Es	Excludes from the database every include file specified with an absolute path or found in an absolute path specified in the INCLUDE environment variable.
/HELP	Calls QuickHelp for help on PWBRMAKE.
/Iu	Includes unreferenced symbols.
/n	Forces a nonincremental build and prevents truncation of .SBR files.
/o filename	Specifies a name for the database file.
/v	Displays verbose output.
/?	Displays a summary of PWBRMAKE command-line syntax.

QuickHelp

The QuickHelp utility displays Help files. All MASM reserved words and error messages can be used for *topic*.

Syntax

QH [[*options*]] [[*topic*]]

Options

Option	Action
/d *filename*	Specifies either a specific database name or a path where the databases are found.
/l*number*	Specifies the number of lines the QuickHelp window should occupy.
/m*number*	Changes the screen mode to display the specified number of lines, where *number* is in the range 25 to 60.
/p *filename*	Sets the name of the paste file.
/pa[[*filename*]]	Specifies that pasting operations are appended to the current paste file (rather than overwriting the file).
/q	Prevents the version box from being displayed when QuickHelp is installed as a keyboard monitor.
/r *command*	Specifies the command that QuickHelp should execute when the right mouse button is pressed. The *command* can be one of the following letters: l — Display last topic i — Display history of help topics w — Hide window b — Display previous topic e — Find next topic t — Display contents
/s	Specifies that clicking the mouse above or below the scroll box causes QuickHelp to scroll by lines rather than pages.

Option	Action
/t *name*	Directs QuickHelp to copy the specified section of the given topic to the current paste file and exit. The *name* may be: All — Paste the entire topic Syntax — Paste the syntax only Example — Paste the example only If the topic is not found, QuickHelp returns an exit code of 1.
/u	Specifies that QuickHelp is being run by a utility. If the topic specified on the command line is not found, QuickHelp immediately exits with an exit code of 3.

Environment Variables

Variable	Description
HELPFILES	Specifies path of help files or list of help filenames.
QH	Specifies default command-line options.
TMP	Specifies directory of default paste file.

RM

The RM utility moves a file to a hidden DELETED subdirectory of the directory containing the file. Use the UNDEL utility to recover the file and the EXP utility to mark the hidden file for deletion.

Syntax

RM [[*options*]] [[*files*]]

Options

Option	Action
/F	Deletes read-only files without prompting.
/HELP	Calls QuickHelp for help on RM.
/I	Inquires for permission before removing each file.
/K	Keeps read-only files without prompting.
/R *directory*	Recurses into subdirectories of the specified directory.
/?	Displays a summary of RM command-line syntax.

UNDEL

The UNDEL utility moves a file from a hidden DELETED subdirectory to the parent directory. UNDEL is used along with EXP and RM to manage backup files.

Syntax

UNDEL [[{*option* | *files*}]]

Options

Option	Action
/HELP	Calls QuickHelp for help on UNDEL.
/?	Displays a summary of UNDEL command-line syntax.

C H A P T E R 2

Directives

Topical Cross-reference for Directives. 22
Directives 25

Topical Cross-reference for Directives

Code Labels

ALIGN	EVEN
LABEL	ORG

Conditional Assembly

ELSE	ELSEIF	ELSEIF2
ENDIF	IF	IF2
IFB/IFNB	IFDEF/IFNDEF	IFDIF/IFDIFI
IFE	IFIDN/IFIDNI	

Conditional Control Flow

.BREAK	.CONTINUE	.ELSE
.ELSEIF	.ENDIF	.ENDW
.IF	.REPEAT	.UNTIL/
.UNTILCXZ	.WHILE	

Conditional Error

.ERR	.ERR2	.ERRB
.ERRDEF	.ERRDIF/.ERRDIFI	.ERRE
.ERRIDN/.ERRIDNI	.ERRNB	.ERRNDEF
.ERRNZ		

Data Allocation

ALIGN	BYTE/SBYTE	DWORD/SDWORD
EVEN	FWORD	LABEL
ORG	QWORD	REAL4
REAL8	REAL10	TBYTE
WORD/SWORD		

Equates

=

EQU

TEXTEQU

Listing Control

.CREF	.LIST	.LISTALL
.LISTIF	.LISTMACRO	.LISTMACROALL
.NOCREF	.NOLIST	.NOLISTIF
.NOLISTMACRO	PAGE	SUBTITLE
.TFCOND	TITLE	

Macros

ENDM	EXITM	GOTO
LOCAL	MACRO	PURGE

Miscellaneous

ASSUME	COMMENT	ECHO
END	INCLUDE	INCLUDELIB
OPTION	POPCONTEXT	PUSHCONTEXT
.RADIX		

Procedures

ENDP	INVOKE	PROC
PROTO	USES	

Processor

.186	.286	.286P
.287	.386	.386P
.387	.486	.486P
.8086	.8087	.NO87

Repeat Blocks

ENDM	FOR	FORC
GOTO	REPEAT	WHILE

Scope

COMM	EXTERN	EXTERNDEF
INCLUDELIB	PUBLIC	

Segment

.ALPHA	ASSUME	.DOSSEG
END	ENDS	GROUP
SEGMENT	.SEQ	

Simplified Segment

.CODE	.CONST	.DATA
.DATA?	.DOSSEG	.EXIT
.FARDATA	.FARDATA?	.MODEL
.STACK	.STARTUP	

String

CATSTR	INSTR
SIZESTR	SUBSTR

Structure and Record

ENDS	RECORD	STRUCT
TYPEDEF	UNION	

Directives

name = expression
Assigns the numeric value of *expression* to *name*. The symbol may be redefined later.

.186
Enables assembly of instructions for the 80186 processor; disables assembly of instructions introduced with later processors. Also enables 8087 instructions.

.286
Enables assembly of nonprivileged instructions for the 80286 processor; disables assembly of instructions introduced with later processors. Also enables 80287 instructions.

.286P
Enables assembly of all instructions (including privileged) for the 80286 processor; disables assembly of instructions introduced with later processors. Also enables 80287 instructions.

.287
Enables assembly of instructions for the 80287 coprocessor; disables assembly of instructions introduced with later coprocessors.

.386
Enables assembly of nonprivileged instructions for the 80386 processor; disables assembly of instructions introduced with later processors. Also enables 80387 instructions.

.386P
Enables assembly of all instructions (including privileged) for the 80386 processor; disables assembly of instructions introduced with later processors. Also enables 80387 instructions.

.387
Enables assembly of instructions for the 80387 coprocessor.

.486
Enables assembly of nonprivileged instructions for the 80486 processor.

.486P
Enables assembly of all instructions (including privileged) for the 80486 processor.

.8086
Enables assembly of 8086 instructions (and the identical 8088 instructions); disables assembly of instructions introduced with later processors. Also enables 8087 instructions. This is the default mode for processors.

.8087
Enables assembly of 8087 instructions; disables assembly of instructions introduced with later coprocessors. This is the default mode for coprocessors.

ALIGN [[*number*]]
Aligns the next variable or instruction on a byte that is a multiple of *number*.

.ALPHA
Orders segments alphabetically.

ASSUME *segregister*:*name* [[**,** *segregister*:*name*]]...
ASSUME *dataregister*:*type* [[**,** *dataregister*:*type*]]...
ASSUME *register*:**ERROR** [[**,** *register*:**ERROR**]]...
ASSUME [[*register*:]] **NOTHING** [[**,** *register*:**NOTHING**]]...
Enables error-checking for register values. After an **ASSUME** is put into effect, the assembler watches for changes to the values of the given registers. **ERROR** generates an error if the register is used. **NOTHING** removes register error-checking. You can combine different kinds of assumptions in one statement.

.BREAK [[**.IF** *condition*]]
Generates code to terminate a **.WHILE** or **.REPEAT** block if *condition* is true.

[[*name*]] **BYTE** *initializer* [[**,** *initializer*]]...
Allocates and optionally initializes a byte of storage for each *initializer*. Can also be used as a type specifier anywhere a type is legal.

name **CATSTR** [[*textitem1* [[**,** *textitem2*]]...]]
Concatenates text items. Each text item can be a literal string, a constant preceded by a **%**, or the string returned by a macro function.

.CODE [[*name*]]
When used with **.MODEL**, indicates the start of a code segment called *name* (the default segment name is _TEXT for tiny, small, compact, and flat models, or *module*_TEXT for other models).

COMM *definition* [[**,** *definition*]]...
Creates a communal variable with the attributes specified in *definition*. Each *definition* has the following form:

[[*langtype*]] [[**NEAR** | **FAR**]] *label*:*type*[[:*count*]]

The *label* is the name of the variable. The *type* can be any type specifier (**BYTE**, **WORD**, and so on) or an integer specifying the number of bytes. The *count* specifies the number of data objects (one is the default).

COMMENT *delimiter* [[*text*]]
[[*text*]]

[[*text*]] *delimiter* [[*text*]]

Treats all *text* between or on the same line as the delimiters as a comment.

.CONST
When used with **.MODEL**, starts a constant data segment (with segment name CONST). This segment has the read-only attribute.

.CONTINUE [[**.IF** *condition*]]
Generates code to jump to the top of a **.WHILE** or **.REPEAT** block if *condition* is true.

.CREF
Enables listing of symbols in the symbol portion of the symbol table and browser file.

.DATA
When used with **.MODEL**, starts a near data segment for initialized data (segment name _DATA).

.DATA?
When used with **.MODEL**, starts a near data segment for uninitialized data (segment name _BSS).

.DOSSEG
Orders the segments according to the MS-DOS segment convention: CODE first, then segments not in DGROUP, and then segments in DGROUP. The segments in DGROUP follow this order: segments not in BSS or STACK, then BSS segments, and finally STACK segments. Primarily used for ensuring CodeView support in MASM stand-alone programs. Same as **DOSSEG**.

DOSSEG
Identical to .**DOSSEG**, which is the preferred form.

DB
Can be used to define data like **BYTE**.

DD
Can be used to define data like **DWORD**.

DF
Can be used to define data like **FWORD**.

DQ
Can be used to define data like **QWORD**.

DT
Can be used to define data like **TBYTE**.

DW
Can be used to define data like **WORD**.

[[*name*]] **DWORD** *initializer* [[**,** *initializer*]]...
Allocates and optionally initializes a doubleword (4 bytes) of storage for each *initializer*. Can also be used as a type specifier anywhere a type is legal.

ECHO *message*
Displays *message* to the standard output device (by default, the screen). Same as **%OUT**.

.ELSE
See **.IF**.

ELSE
Marks the beginning of an alternate block within a conditional block. See **IF**.

ELSEIF
Combines **ELSE** and **IF** into one statement. See **IF**.

ELSEIF2
ELSEIF block evaluated on every assembly pass if **OPTION:SETIF2** is **TRUE**.

END [[*address*]]
Marks the end of a module and, optionally, sets the program entry point to *address*.

.ENDIF
See **.IF**.

ENDIF
See **IF**.

ENDM
Terminates a macro or repeat block. See **MACRO**, **FOR**, **FORC**, **REPEAT**, or **WHILE**.

name **ENDP**
Marks the end of procedure *name* previously begun with **PROC**. See **PROC**.

name **ENDS**
Marks the end of segment, structure, or union *name* previously begun with **SEGMENT**, **STRUCT**, **UNION**, or a simplified segment directive.

.ENDW
See **.WHILE**.

name **EQU** *expression*
Assigns numeric value of *expression* to *name*. The *name* cannot be redefined later.

name **EQU** <*text*>
Assigns specified *text* to *name*. The *name* can be assigned a different *text* later. See **TEXTEQU**.

.ERR [[*message*]]
Generates an error.

.ERR2 ⟦*message*⟧
.ERR block evaluated on every assembly pass if **OPTION:SETIF2** is **TRUE.**

.ERRB <*textitem*> ⟦**,** *message*⟧
Generates an error if *textitem* is blank.

.ERRDEF *name* ⟦**,** *message*⟧
Generates an error if *name* is a previously defined label, variable, or symbol.

.ERRDIF⟦**I**⟧ <*textitem1*>**,** <*textitem2*> ⟦**,** *message*⟧
Generates an error if the text items are different. If **I** is given, the comparison is case insensitive.

.ERRE *expression* ⟦**,** *message*⟧
Generates an error if *expression* is false (0).

.ERRIDN⟦**I**⟧ <*textitem1*>**,** <*textitem2*> ⟦**,** *message*⟧
Generates an error if the text items are identical. If **I** is given, the comparison is case insensitive.

.ERRNB <*textitem*> ⟦**,** *message*⟧
Generates an error if *textitem* is not blank.

.ERRNDEF *name* ⟦**,** *message*⟧
Generates an error if *name* has not been defined.

.ERRNZ *expression* ⟦**,** *message*⟧
Generates an error if *expression* is true (nonzero).

EVEN
Aligns the next variable or instruction on an even byte.

.EXIT ⟦*expression*⟧
Generates termination code. Returns optional *expression* to shell.

EXITM ⟦*textitem*⟧
Terminates expansion of the current repeat or macro block and begins assembly of the next statement outside the block. In a macro function, *textitem* is the value returned.

EXTERN ⟦*langtype*⟧ *name* ⟦**(***altid***)**⟧ **:***type* ⟦**,** ⟦*langtype*⟧ *name* ⟦**(***altid***)**⟧ **:***type*⟧...
Defines one or more external variables, labels, or symbols called *name* whose type is *type*. The *type* can be **ABS**, which imports *name* as a constant. Same as **EXTRN**.

EXTERNDEF ⟦*langtype*⟧ *name***:***type* ⟦**,** ⟦*langtype*⟧ *name***:***type*⟧...
Defines one or more external variables, labels, or symbols called *name* whose type is *type*. If *name* is defined in the module, it is treated as **PUBLIC**. If *name* is referenced in the module, it is treated as **EXTERN**. If *name* is not referenced, it is ignored. The *type* can be **ABS**, which imports *name* as a constant. Normally used in include files.

EXTRN
See **EXTERN**.

.FARDATA [[*name*]]
When used with **.MODEL**, starts a far data segment for initialized data (segment name FAR_DATA or *name*).

.FARDATA? [[*name*]]
When used with **.MODEL**, starts a far data segment for uninitialized data (segment name FAR_BSS or *name*).

FOR parameter [[**:REQ** | **:=**default]] **,** <argument [[**,** argument]]...>
statements
ENDM
Marks a block that will be repeated once for each *argument*, with the current *argument* replacing *parameter* on each repetition. Same as **IRP**.

FORC
*parameter***,** *<string> statements*
ENDM
Marks a block that will be repeated once for each character in *string*, with the current character replacing *parameter* on each repetition. Same as **IRPC**.

[[*name*]] **FWORD** *initializer* [[**,** *initializer*]]...
Allocates and optionally initializes 6 bytes of storage for each *initializer*. Also can be used as a type specifier anywhere a type is legal.

GOTO *macrolabel*
Transfers assembly to the line marked **:***macrolabel*. **GOTO** is permitted only inside **MACRO**, **FOR**, **FORC**, **REPEAT**, and **WHILE** blocks. The label must be the only directive on the line and must be preceded by a leading colon.

name **GROUP** *segment* [[**,** *segment*]]...
Add the specified *segments* to the group called *name*.

.IF *condition1*
statements
[[**.ELSEIF** condition2
statements]]
[[**.ELSE**
statements]]
.ENDIF
Generates code that tests *condition1* (for example, AX > 7) and executes the *statements* if that condition is true. If an **.ELSE** follows, its statements are executed if the original condition was false. Note that the conditions are evaluated at run time.

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

Grants assembly of *ifstatements* if *expression1* is true (nonzero) or *elseifstatements* if *expression1* is false (0) and *expression2* is true. The following directives may be substituted for **ELSEIF**: **ELSEIFB**,

ELSEIFDEF, **ELSEIFDIF**, **ELSEIFDIFI**, **ELSEIFE**, **ELSEIFIDN**, **ELSEIFIDNI**, **ELSEIFNB**, and **ELSEIFNDEF**. Optionally, assembles *elsestatements* if the previous expression is false. Note that the expressions are evaluated at assembly time.

IF2 *expression*
IF block is evaluated on every assembly pass if **OPTION:SETIF2** is **TRUE**. See **IF** for complete syntax.

IFB *textitem*
Grants assembly if *textitem* is blank. See **IF** for complete syntax.

IFDEF *name*
Grants assembly if *name* is a previously defined label, variable, or symbol. See **IF** for complete syntax.

IFDIF[[**I**]] *textitem1***,** *textitem2*
Grants assembly if the text items are different. If **I** is given, the comparison is case insensitive. See **IF** for complete syntax.

IFE *expression*
Grants assembly if *expression* is false (0). See **IF** for complete syntax.

IFIDN[[**I**]] *textitem1***,** *textitem2*
Grants assembly if the text items are identical. If **I** is given, the comparison is case insensitive. See **IF** for complete syntax.

IFNB *textitem*
Grants assembly if *textitem* is not blank. See **IF** for complete syntax.

IFNDEF *name*
Grants assembly if *name* has not been defined. See **IF** for complete syntax.

INCLUDE *filename*
Inserts source code from the source file given by *filename* into the current source file during assembly. The *filename* must be enclosed in angle brackets if it includes a backslash, semicolon, greater-than symbol, less-than symbol, single quotation mark, or double quotation mark.

INCLUDELIB *libraryname*
Informs the linker that the current module should be linked with *libraryname*. The *libraryname* must be enclosed in angle brackets if it includes a backslash, semicolon, greater-than symbol, less-than symbol, single quotation mark, or double quotation mark.

name **INSTR** [[*position***,**]] *textitem1***,** *textitem2*
Finds the first occurrence of *textitem2* in *textitem1*. The starting *position* is optional. Each text item can be a literal string, a constant preceded by a **%**, or the string returned by a macro function.

INVOKE *expression* ⟦**,** *arguments*⟧
Calls the procedure at the address given by *expression*, passing the arguments on the stack or in registers according to the standard calling conventions of the language type. Each argument passed to the procedure may be an expression, a register pair, or an address expression (an expression preceded by **ADDR**).

IRP
See **FOR**.

IRPC
See **FORC**.

name **LABEL** *type*
Creates a new label by assigning the current location-counter value and the given *type* to *name*.

name **LABEL** ⟦**NEAR** | **FAR** | **PROC**⟧ **PTR** ⟦*type*⟧
Creates a new label by assigning the current location-counter value and the given *type* to *name*.

.LALL
See **.LISTMACROALL**.

.LFCOND
See **.LISTIF**.

.LIST
Starts listing of statements. This is the default.

.LISTALL
Starts listing of all statements. Equivalent to the combination of **.LIST**, **.LISTIF**, and **.LISTMACROALL**.

.LISTIF
Starts listing of statements in false conditional blocks. Same as **.LFCOND**.

.LISTMACRO
Starts listing of macro expansion statements that generate code or data. This is the default. Same as **.XALL**.

.LISTMACROALL
Starts listing of all statements in macros. Same as **.LALL**.

LOCAL *localname* ⟦**,** *localname*⟧...
Within a macro, **LOCAL** defines labels that are unique to each instance of the macro.

LOCAL *label* ⟦ **[***count* **]** ⟧⟦**:***type*⟧ ⟦**,** *label* ⟦ **[***count***]** ⟧ ⟦*type*⟧⟧...
Within a procedure definition (**PROC**), **LOCAL** creates stack-based variables that exist for the duration of the procedure. The *label* may be a simple variable or an array containing *count* elements.

name **MACRO** [[*parameter* [[**:REQ** | **:=***default* | **:VARARG**]]]]...
statements
ENDM [[*value*]]
Marks a macro block called *name* and establishes *parameter* placeholders for arguments passed when the macro is called. A macro function returns *value* to the calling statement.

.MODEL *memorymodel* [[**,** *langtype*]] [[**,** *stackoption*]]
Initializes the program memory model. The *memorymodel* can be **TINY**, **SMALL**, **COMPACT**, **MEDIUM**, **LARGE**, **HUGE**, or **FLAT**. The *langtype* can be **C**, **BASIC**, **FORTRAN**, **PASCAL**, **SYSCALL**, or **STDCALL**. The *stackoption* can be **NEARSTACK** or **FARSTACK**.

NAME *modulename*
Ignored.

.NO87
Disallows assembly of all floating-point instructions.

.NOCREF [[*name*[[**,** *name*]]...]]
Suppresses listing of symbols in the symbol table and browser file. If names are specified, only the given names are suppressed. Same as **.XCREF**.

.NOLIST
Suppresses program listing. Same as **.XLIST**.

.NOLISTIF
Suppresses listing of conditional blocks whose condition evaluates to false (0). This is the default. Same as **.SFCOND**.

.NOLISTMACRO
Suppresses listing of macro expansions. Same as **.SALL**.

OPTION *optionlist*
Enables and disables features of the assembler. Available options include **CASEMAP**, **DOTNAME**, **NODOTNAME**, **EMULATOR**, **NOEMULATOR**, **EPILOGUE**, **EXPR16**, **EXPR32**, **LANGUAGE**, **LJMP**, **NOLJMP**, **M510**, **NOM510**, **NOKEYWORD**, **NOSIGNEXTEND**, **OFFSET**, **OLDMACROS**, **NOOLDMACROS**, **OLDSTRUCTS**, **NOOLDSTRUCTS**, **PROC**, **PROLOGUE**, **READONLY**, **NOREADONLY**, **SCOPED**, **NOSCOPED**, **SEGMENT**, and **SETIF2**.

ORG *expression*
Sets the location counter to *expression*.

%OUT
See **ECHO**.

PAGE [[[[*length*]]**,** *width*]]
Sets line *length* and character *width* of the program listing. If no arguments are given, generates a page break.

PAGE +
Increments the section number and resets the page number to 1.

POPCONTEXT *context*
Restores part or all of the current *context* (saved by the **PUSHCONTEXT** directive). The *context* can be **ASSUMES**, **RADIX**, **LISTING**, **CPU**, or **ALL**.

label **PROC** ⟦*distance*⟧ ⟦*langtype*⟧ ⟦*visibility*⟧ ⟦<*prologuearg*>⟧
⟦**USES** *reglist*⟧ ⟦**,** *parameter* ⟦**:***tag*⟧⟧...
statements
label **ENDP**
Marks start and end of a procedure block called *label*. The statements in the block can be called with the **CALL** instruction or **INVOKE** directive.

label **PROTO** ⟦*distance*⟧ ⟦*langtype*⟧ ⟦**,** ⟦*parameter*⟧**:***tag*⟧...
Prototypes a function.

PUBLIC ⟦*langtype*⟧ *name* ⟦**,** ⟦*langtype*⟧ *name*⟧...
Makes each variable, label, or absolute symbol specified as *name* available to all other modules in the program.

PURGE *macroname* ⟦**,** *macroname*⟧...
Deletes the specified macros from memory.

PUSHCONTEXT *context*
Saves part or all of the current *context*: segment register assumes, radix value, listing and cref flags, or processor/coprocessor values. The *context* can be **ASSUMES**, **RADIX**, **LISTING**, **CPU**, or **ALL**.

⟦*name*⟧ **QWORD** *initializer* ⟦**,** *initializer*⟧...
Allocates and optionally initializes 8 bytes of storage for each *initializer*. Also can be used as a type specifier anywhere a type is legal.

.RADIX *expression*
Sets the default radix, in the range 2 to 16, to the value of *expression*.

name **REAL4** *initializer* ⟦**,** *initializer*⟧...
Allocates and optionally initializes a single-precision (4-byte) floating-point number for each *initializer*.

name **REAL8** *initializer* ⟦**,** *initializer*⟧...
Allocates and optionally initializes a double-precision (8-byte) floating-point number for each *initializer*.

name **REAL10** *initializer* ⟦**,** *initializer*⟧...
Allocates and optionally initializes a 10-byte floating-point number for each *initializer*.

recordname **RECORD** *fieldname*:*width* [[= *expression*]]
[[, *fieldname*:*width* [[= *expression*]]]]...
Declares a record type consisting of the specified fields. The *fieldname* names the field, *width* specifies the number of bits, and *expression* gives its initial value.

.REPEAT
statements
.UNTIL *condition*
Generates code that repeats execution of the block of *statements* until *condition* becomes true. **.UNTILCXZ**, which becomes true when CX is zero, may be substituted for **.UNTIL**. The *condition* is optional with **.UNTILCXZ**.

REPEAT *expression*
statements
ENDM
Marks a block that is to be repeated *expression* times. Same as **REPT**.

REPT
See **REPEAT**.

.SALL
See **.NOLISTMACRO**.

name **SBYTE** *initializer* [[, *initializer*]]...
Allocates and optionally initializes a signed byte of storage for each *initializer*. Can also be used as a type specifier anywhere a type is legal.

name **SDWORD** *initializer* [[, *initializer*]]...
Allocates and optionally initializes a signed doubleword (4 bytes) of storage for each *initializer*. Also can be used as a type specifier anywhere a type is legal.

name **SEGMENT** [[**READONLY**]] [[*align*]] [[*combine*]] [[*use*]] [['*class*']]
statements
name **ENDS**
Defines a program segment called *name* having segment attributes *align* (**BYTE**, **WORD**, **DWORD**, **PARA**, **PAGE**), *combine* (**PUBLIC**, **STACK**, **COMMON**, **MEMORY**, **AT** *address*, **PRIVATE**), *use* (**USE16**, **USE32**, **FLAT**), and *class*.

.SEQ
Orders segments sequentially (the default order).

.SFCOND
See **.NOLISTIF**.

name **SIZESTR** *textitem*
Finds the size of a text item.

.STACK [[*size*]]
When used with **.MODEL**, defines a stack segment (with segment name STACK). The optional *size* specifies the number of bytes for the stack (default 1,024). The **.STACK** directive automatically closes the stack statement.

.STARTUP
Generates program start-up code.

STRUC
See **STRUCT**.

name **STRUCT** [[*alignment*]] [[**, NONUNIQUE**]]
fielddeclarations
name **ENDS**
Declares a structure type having the specified *fielddeclarations*. Each field must be a valid data definition. Same as **STRUC**.

name **SUBSTR** *textitem***,** *position* [[**,** *length*]]
Returns a substring of *textitem*, starting at *position*. The *textitem* can be a literal string, a constant preceded by a **%**, or the string returned by a macro function.

SUBTITLE *text*
Defines the listing subtitle. Same as **SUBTTL**.

SUBTTL
See **SUBTITLE**.

name **SWORD** *initializer* [[**,** *initializer*]]...
Allocates and optionally initializes a signed word (2 bytes) of storage for each *initializer*. Can also be used as a type specifier anywhere a type is legal.

[[*name*]] **TBYTE** *initializer* [[**,** *initializer*]]...
Allocates and optionally initializes 10 bytes of storage for each *initializer*. Can also be used as a type specifier anywhere a type is legal.

name **TEXTEQU** [[*textitem*]]
Assigns *textitem* to *name*. The *textitem* can be a literal string, a constant preceded by a **%**, or the string returned by a macro function.

.TFCOND
Toggles listing of false conditional blocks.

TITLE *text*
Defines the program listing title.

name **TYPEDEF** *type*
Defines a new type called *name*, which is equivalent to *type*.

name **UNION** [[*alignment*]] [[**, NONUNIQUE**]]
fielddeclarations
[[*name*]] **ENDS**
Declares a union of one or more data types. The *fielddeclarations* must be valid data definitions. Omit the **ENDS** *name* label on nested **UNION** definitions.

.UNTIL
See **.REPEAT**.

.UNTILCXZ
See **.REPEAT**.

.WHILE *condition*
statements
.ENDW
Generates code that executes the block of *statements* while *condition* remains true.

WHILE *expression*
statements
ENDM
Repeats assembly of block *statements* as long as *expression* remains true.

[[*name*]] **WORD** initializer [[**,** *initializer*]]...
Allocates and optionally initializes a word (2 bytes) of storage for each *initializer*. Can also be used as a type specifier anywhere a type is legal.

.XALL
See **.LISTMACRO**.

.XCREF
See **.NOCREF**.

.XLIST
See **.NOLIST**.

C H A P T E R 3

Symbols and Operators

Topical Cross-reference for Symbols 40
Topical Cross-reference for Operators 41
Predefined Symbols 43
Operators 44
Run-Time Operators 48

Topical Cross-reference for Symbols

Date and Time Information

@Date

@Time

Environment Information

@Cpu

@Environ

@Interface

@Version

File Information

@FileCur

@FileName

@Line

Macro Functions

@CatStr

@InStr

@SizeStr

@SubStr

Miscellaneous

$	?	@@:
@B	@F	

Segment Information

@code	@CodeSize	@CurSeg
@data	@DataSize	@fardata
@fardata?	@Model	@stack
@WordSize		

Topical Cross-reference for Operators

Arithmetic

*	+	-
.	/	[]
MOD		

Control Flow

!	!=	&
&&	<	< =
= =	>	> =
‖		

Logical and Shift

AND	NOT	OR
SHL	SHR	XOR

Macro

!	%	&
;;	<>	

Miscellaneous

’’	“ ”	:
::	;	CARRY?
DUP	OVERFLOW?	PARITY?
SIGN?	ZERO?	

Record

MASK

WIDTH

Relational

EQ	GE	GT
LE	LT	NE

Segment

:

LROFFSET

OFFSET

SEG

Type

HIGH	HIGHWORD	LENGTH
LENGTHOF	LOW	LOWWORD
OPATTR	PTR	SHORT
SIZE	SIZEOF	THIS
TYPE		

Predefined Symbols

$
The current value of the location counter.

?
In data declarations, a value that the assembler allocates but does not initialize.

@@:
Defines a code label recognizable only between *label1* and *label2*, where *label1* is either start of code or the previous **@@:** label, and *label2* is either end of code or the next **@@:** label. See **@B** and **@F**.

@B
The location of the previous **@@:** label.

@CatStr(*string1* [**,** *string2*...] **)**
Macro function that concatenates one or more strings. Returns a string.

@code
The name of the code segment (text macro).

@CodeSize
0 for **TINY**, **SMALL**, **COMPACT**, and **FLAT** models, and 1 for **MEDIUM**, **LARGE**, and **HUGE** models (numeric equate).

@Cpu
A bit mask specifying the processor mode (numeric equate).

@CurSeg
The name of the current segment (text macro).

@data
The name of the default data group. Evaluates to DGROUP for all models except **FLAT**. Evaluates to **FLAT** under the **FLAT** memory model (text macro).

@DataSize
0 for **TINY**, **SMALL**, **MEDIUM**, and **FLAT** models, 1 for **COMPACT** and **LARGE** models, and 2 for **HUGE** model (numeric equate).

@Date
The system date in the format mm/dd/yy (text macro).

@Environ(*envvar* **)**
Value of environment variable *envvar* (macro function).

@F
The location of the next **@@:** label.

@fardata
The name of the segment defined by the **.FARDATA** directive (text macro).

@fardata?
The name of the segment defined by the **.FARDATA?** directive (text macro).

@FileCur
The name of the current file (text macro).

@FileName
The base name of the main file being assembled (text macro).

@InStr([[*position*]]**,** *string1***,** *string2* **)**
Macro function that finds the first occurrence of *string2* in *string1*, beginning at *position* within *string1*. If *position* does not appear, search begins at start of *string1*. Returns a position integer or 0 if *string2* is not found.

@Interface
Information about the language parameters (numeric equate).

@Line
The source line number in the current file (numeric equate).

@Model
1 for **TINY** model, 2 for **SMALL** model, 3 for **COMPACT** model, 4 for **MEDIUM** model, 5 for **LARGE** model, 6 for **HUGE** model, and 7 for **FLAT** model (numeric equate).

@SizeStr(*string* **)**
Macro function that returns the length of the given string. Returns an integer.

@SubStr(*string***,** *position* [[**,** *length*]] **)**
Macro function that returns a substring starting at *position*.

@stack
DGROUP for near stacks or STACK for far stacks (text macro).

@Time
The system time in 24-hour hh:mm:ss format (text macro).

@Version
610 in MASM 6.1 (text macro).

@WordSize
Two for a 16-bit segment or 4 for a 32-bit segment (numeric equate).

Operators

expression1 **+** *expression2*
Returns *expression1* plus *expression2*.

expression1 – *expression2*
Returns *expression1* minus *expression2*.

expression1 ***** *expression2*
Returns *expression1* times *expression2*.

expression1 / *expression2*
Returns *expression1* divided by *expression2*.

–*expression*
Reverses the sign of *expression*.

expression1 **[***expression2***]**
Returns *expression1* plus **[***expression2***]**.

*segment***:** *expression*
Overrides the default segment of *expression* with *segment*. The *segment* can be a segment register, group name, segment name, or segment expression. The *expression* must be a constant.

*expression***.** *field* [[**.** *field*]]...
Returns *expression* plus the offset of *field* within its structure or union.

[*register***].** *field* [[**.** *field*]]...
Returns value at the location pointed to by *register* plus the offset of *field* within its structure or union.

<*text*>
Treats *text* as a single literal element.

"*text*"
Treats "*text*" as a string.

'*text*'
Treats '*text*' as a string.

!*character*
Treats *character* as a literal character rather than as an operator or symbol.

;*text*
Treats *text* as a comment.

;;*text*
Treats *text* as a comment in a macro that appears only in the macro definition. The listing does not show *text* where the macro is expanded.

%*expression*
Treats the value of *expression* in a macro argument as text.

&parameter**&**
Replaces *parameter* with its corresponding argument value.

ABS
See the **EXTERNDEF** directive.

ADDR
See the **INVOKE** directive.

expression1 **AND** *expression2*
Returns the result of a bitwise AND operation for *expression1* and *expression2*.

count **DUP** (*initialvalue* [[**,** *initialvalue*]]...)
Specifies *count* number of declarations of *initialvalue*.

expression1 **EQ** *expression2*
Returns true (–1) if *expression1* equals *expression2*, or returns false (0) if it does not.

expression1 **GE** *expression2*
Returns true (–1) if *expression1* is greater-than-or-equal-to *expression2*, or returns false (0) if it is not.

expression1 **GT** *expression2*
Returns true (–1) if *expression1* is greater than *expression2*, or returns false (0) if it is not.

HIGH *expression*
Returns the high byte of *expression*.

HIGHWORD *expression*
Returns the high word of *expression*.

expression1 **LE** *expression2*
Returns true (–1) if *expression1* is less than or equal to *expression2*, or returns false (0) if it is not.

LENGTH *variable*
Returns the number of data items in *variable* created by the first initializer.

LENGTHOF *variable*
Returns the number of data objects in *variable*.

LOW *expression*
Returns the low byte of *expression*.

LOWWORD *expression*
Returns the low word of *expression*.

LROFFSET *expression*
Returns the offset of *expression*. Same as **OFFSET**, but it generates a loader resolved offset, which allows Windows to relocate code segments.

expression1 **LT** *expression2*
Returns true (–1) if *expression1* is less than *expression2*, or returns false (0) if it is not.

MASK {*recordfieldname* | *record*}
Returns a bit mask in which the bits in *recordfieldname* or *record* are set and all other bits are cleared.

expression1 **MOD** *expression2*
Returns the integer value of the remainder (modulo) when dividing *expression1* by *expression2*.

expression1 **NE** *expression2*
Returns true (–1) if *expression1* does not equal *expression2*, or returns false (0) if it does.

NOT *expression*
Returns *expression* with all bits reversed.

OFFSET *expression*
Returns the offset of *expression*.

OPATTR *expression*
Returns a word defining the mode and scope of *expression*. The low byte is identical to the byte returned by **.TYPE**. The high byte contains additional information.

expression1 **OR** *expression2*
Returns the result of a bitwise OR operation for *expression1* and *expression2*.

type **PTR** *expression*
Forces the *expression* to be treated as having the specified *type*.

[[*distance*]] **PTR** *type*
Specifies a pointer to *type*.

SEG *expression*
Returns the segment of *expression*.

expression **SHL** *count*
Returns the result of shifting the bits of *expression* left *count* number of bits.

SHORT *label*
Sets the type of *label* to short. All jumps to *label* must be short (within the range –128 to +127 bytes from the jump instruction to *label*).

expression **SHR** *count*
Returns the result of shifting the bits of *expression* right *count* number of bits.

SIZE *variable*
Returns the number of bytes in *variable* allocated by the first initializer.

SIZEOF {*variable* | *type*}
Returns the number of bytes in *variable* or *type*.

THIS *type*
Returns an operand of specified *type* whose offset and segment values are equal to the current location-counter value.

.TYPE *expression*
See **OPATTR**.

TYPE *expression*
Returns the type of *expression*.

WIDTH {*recordfieldname* | *record*}
Returns the width in bits of the current *recordfieldname* or *record*.

expression1 **XOR** *expression2*
Returns the result of a bitwise XOR operation for *expression1* and *expression2*.

Run-Time Operators

The following operators are used only within **.IF**, **.WHILE**, or **.REPEAT** blocks and are evaluated at run time, not at assembly time:

expression1 **==** *expression2*
Is equal to.

expression1 **!=** *expression2*
Is not equal to.

expression1 > *expression2*
Is greater than.

expression1 >= *expression2*
Is greater than or equal to.

expression1 < *expression2*
Is less than.

expression1 <= *expression2*
Is less than or equal to.

expression1 **||** *expression2*
Logical OR.

expression1 **&&** *expression2*
Logical AND.

expression1 **&** *expression2*
Bitwise AND.

!*expression*
Logical negation.

CARRY?
Status of carry flag.

OVERFLOW?
Status of overflow flag.

PARITY?
Status of parity flag.

SIGN?
Status of sign flag.

ZERO?
Status of zero flag.

CHAPTER 4

Processor

Topical Cross-reference for Processor Instructions 50
Interpreting Processor Instructions 53
Flags 53
Syntax 54
Examples 54
Clock Speeds 54
Timings on the 8088 and 8086 Processors 55
Timings on the 80286-80486 Processors 56
Interpreting Encodings 56
Interpreting 80386–80486 Encoding Extensions 59
16-bit Encoding. 60
32-bit Encoding. 60
Address-Size Prefix 60
Operand-Size Prefix 60
Encoding Differences for 32-Bit Operations 60
Scaled Index Base Byte 61
Instructions 64

Topical Cross-reference for Processor Instructions

Arithmetic

ADC	ADD	DEC
DIV	IDIV	IMUL
INC	MUL	NEG
SBB	SUB	XADD#

BCD Conversion

AAA	AAD	AAM
AAS	DAA	DAS

Bit Operations

AND	BSF§	BSR§
BT§	BTC§	BTR§
BTS§	NOT	OR
RCL	RCR	ROL
ROR	SAR	SHL/SAL
SHLD§	SHR	SHRD§
XOR		

Compare

BT§	BTC§	BTR§
BTS§	CMP	CMPS
CMPXCHG#	TEST	

Conditional Set

SETA/SETNBE§	SETAE/SETNB§	SETB/SETNAE§
SETBE/SETNA§	SETC§	SETE/SETZ§
SETG/SETNLE§	SETGE/SETNL§	SETL/SETNGE§
SETLE/SETNG§	SETNC§	SETNE/SETNZ§
SETNO§	SETNP/SETPO§	SETNS§
SETO§	SETP/SETPE§	SETS§

* 80186–80486 only. † 80286–80486 only.
§ 80386–80486 only. # 80486 only.

Conditional Transfer

BOUND*	INTO	JA/JNBE
JAE/JNB	JB/JNAE	JBE/JNA
JC	JCXZ/JECXZ	JE/JZ
JG/JNLE	JGE/JNL	JL/JNGE
JLE/JNG	JNC	JNE/JNZ
JNO	JNP/JPO	JNS
JO	JP/JPE	JS

Data Transfer

BSWAP#	CMPXCHG#	LDS/LES
LEA	LFS/LGS/LSS§	LODS
MOV	MOVS	MOVSX§
MOVZX§	STOS	XADD#
XCHG	XLAT/XLATB	

Flag

CLC	CLD	CLI
CMC	LAHF	POPF
PUSHF	SAHF	STC
STD	STI	

Input/Output

IN	INS*
OUT	OUTS*

Loop

JCXZ/JECXZ	LOOP
LOOPE/LOOPZ	LOOPNE/LOOPNZ

* 80186–80486 only. † 80286–80486 only.
§ 80386–80486 only. # 80486 only.

Process Control

ARPL†	CLTS†	LAR†
LGDT/LIDT/LLDT†	LMSW†	LSL†
LTR†	SGDT/SIDT/SLDT†	SMSW†
STR†	VERR†	VERW†
MOV *special*§	INVD#	INVLPG#
WBINVD#		

Processor Control

HLT	LOCK
NOP	WAIT

Stack

PUSH	PUSHF	PUSHA*
PUSHAD*	POP	POPF
POPA*	POPAD*	ENTER*
LEAVE*		

String

MOVS	LODS	STOS
SCAS	CMPS	INS*
OUTS*	REP	REPE/REPZ
REPNE/REPNZ		

Type Conversion

CBW	CWD
CWDE§	CDQ§
BSWAP#	

Unconditional Transfer

CALL	INT	IRET
RET	RETN/RETF	JMP

* 80186–80486 only. † 80286–80486 only.
§ 80386–80486 only. # 80486 only.

Interpreting Processor Instructions

The following sections explain the format of instructions for the 8086, 8088, 80286, 80386, and 80486 processors. Those instructions begin on page 64.

Flags

Only the flags common to all processors are shown. If none of the flags is affected by the instruction, the flag line says No change. If flags can be affected, a two-line entry is shown. The first line shows flag abbreviations as follows:

Abbreviation	Flag
O	Overflow
D	Direction
I	Interrupt
T	Trap
S	Sign
Z	Zero
A	Auxiliary carry
P	Parity
C	Carry

The second line has codes indicating how the flag can be affected:

Code	Effect
1	Sets the flag
0	Clears the flag
?	May change the flag, but the value is not predictable
blank	No effect on the flag
±	Modifies according to the rules associated with the flag

Syntax

Each encoding variation may have different syntaxes corresponding to different addressing modes. The following abbreviations are used:

reg A general-purpose register of any size.

segreg One of the segment registers: DS, ES, SS, or CS (also FS or GS on the 80386–80486).

accum An accumulator register of any size: AL or AX (also EAX on the 80386–80486).

mem A direct or indirect memory operand of any size.

label A labeled memory location in the code segment.

src,dest A source or destination memory operand used in a string operation.

immed A constant operand.

In some cases abbreviations have numeric suffixes to specify that the operand must be a particular size. For example, *reg16* means that only a 16-bit (word) register is accepted.

Examples

One or more examples are shown for each syntax. Their position is not related to the clock speeds in the right column.

Clock Speeds

Column 3 shows the clock speeds for each processor. Sometimes an instruction may have more than one clock speed. Multiple speeds are separated by commas. If several speeds are part of an expression, they are enclosed in parentheses. The following abbreviations are used to specify variations:

EA Effective address. This applies only to the 8088 and 8086 processors, as described in the next section.

b,w,d Byte, word, or doubleword operands.

pm Protected mode.

n Iterations. Repeated instructions may have a base number of clocks plus a number of clocks for each iteration. For example, 8+4*n* means 8 clocks plus 4 clocks for each iteration.

noj No jump. For conditional jump instructions, *noj* indicates the speed if the condition is false and the jump is not taken.

m Next instruction components. Some control transfer instructions take different times depending on the length of the next instruction executed. On the 8088 and 8086, *m* is never a factor. On the 80286, *m* is the number of bytes in the instruction. On the 80386–80486, *m* is the number of components. Each byte of encoding is a component, and the displacement and data are separate components.

W88,88 8088 exceptions. See "Timings on the 8088 and 8086 Processors," following.

Clocks can be converted to nanoseconds by dividing 1 microsecond by the number of megahertz (MHz) at which the processor is running. For example, on a processor running at 8 MHz, 1 clock takes 125 nanoseconds (1000 MHz per nanosecond / 8 MHz).

The clock counts are for best-case timings. Actual timings vary depending on wait states, alignment of the instruction, the status of the prefetch queue, and other factors.

Timings on the 8088 and 8086 Processors

Because of its 8-bit data bus, the 8088 always requires two fetches to get a 16-bit operand. Therefore, instructions that work on 16-bit memory operands take longer on the 8088 than on the 8086. Separate 8088 timings are shown in parentheses following the main timing. For example, 9 (*W88*=13) means that the 8086 with any operands or the 8088 with byte operands take 9 clocks, but the 8088 with word operands takes 13 clocks. Similarly, 16 (*88*=24) means that the 8086 takes 16 clocks, but the 8088 takes 24 clocks.

On the 8088 and 8086, the effective address (*EA*) value must be added for instructions that operate on memory operands. A displacement is any direct memory or constant operand, or any combination of the two. The following shows the number of clocks to add for the effective address:

Components	EA Clocks	Examples
Displacement	6	`mov ax,stuff` `mov ax,stuff+2`
Base or index	5	`mov ax,[bx]` `mov ax,[di]`
Displacement plus base or index	9	`mov ax,[bp+8]` `mov ax,stuff[di]`
Base plus index (BP+DI, BX+SI)	7	`mov ax,[bx+si]` `mov ax,[bp+di]`

Components	EA Clocks	Examples
Base plus index (BP+SI, BX+DI)	8	`mov ax,[bx+di]` `mov ax,[bp+si]`
Base plus index plus displacement (BP+DI+*disp*, BX+SI+*disp*)	11	`mov ax,stuff[bx+si]` `mov ax,[bp+di+8]`
Base plus index plus displacement (BP+SI+*disp*, BX+DI+*disp*)	12	`mov ax,stuff[bx+di]` `mov ax,[bp+si+20]`
Segment override	*EA*+2	`mov ax,es:stuff` `mov ax,ds:[bp+10]`

Timings on the 80286–80486 Processors

On the 80286–80486 processors, the effective address calculation is handled by hardware and is therefore not a factor in clock calculations except in one case. If a memory operand includes all three possible elements—a displacement, a base register, and an index register—then add one clock. On the 80486, the extra clock is not always used. Examples are shown in the following.

```
mov     ax,[bx+di]                  ;No extra
mov     ax,array[bx+di]             ;One extra
mov     ax,[bx+di+6]                ;One extra
```

Note 80186 and 80188 timings are different from 8088, 8086, and 80286 timings. They are not shown in this manual. Timings are also not shown for protected-mode transfers through gates or for the virtual 8086 mode available on the 80386–80486 processors.

Interpreting Encodings

Encodings are shown for each variation of the instruction. This section describes encoding for all processors except the 80386–80486. The encodings take the form of boxes filled with 0s and 1s for bits that are constant for the instruction variation, and abbreviations (in italics) for the following variable bits or bitfields:

d Direction bit. If set, do memory to register; the *reg* field is the destination. If clear, do register to memory or register to register; the *reg* field is the source.

a Accumulator direction bit. If set, move accumulator register to memory. If clear, move memory to accumulator register.

w Word/byte bit. If set, use 16-bit or 32-bit operands. If clear, use 8-bit operands.

s Sign bit. If set, sign-extend 8-bit immediate data to 16 bits.

mod Mode. This 2-bit field gives the register/memory mode with displacement. The possible values are shown below:

mod	**Meaning**
00	This value can have two meanings: If r/m is 110, a direct memory operand is used. If r/m is not 110, the displacement is 0 and an indirect memory operand is used. The operand must be based, indexed, or based indexed.
01	An indirect memory operand is used with an 8-bit displacement.
10	An indirect memory operand is used with a 16-bit displacement.
11	A two-register instruction is used; the *reg* field specifies the destination and the *r/m* field specifies the source.

reg Register. This 3-bit field specifies one of the general-purpose registers:

reg	**16/32-bit if *w*=1**	**8-bit if *w*=0**
000	AX/*EA*X	AL
001	CX/ECX	CL
010	DX/EDX	DL
011	BX/EBX	BL
100	SP/ESP	AH
101	BP/EBP	CH
110	SI/ESI	DH
111	DI/EDI	BH

The *reg* field is sometimes used to specify encoding information rather than a register.

sreg Segment register. This field specifies one of the segment registers:

sreg	**Register**
000	ES
001	CS
010	SS
011	DS
100	FS
101	GS

r/m Register/memory. This 3-bit field specifies a register or memory *r/m* operand.

If the *mod* field is 11, *r/m* specifies the source register using the *reg* field codes. Otherwise, the field has one of the following values:

r/m	**Operand Address**
000	DS:[BX+SI+*disp*]
001	DS:[BX+DI+*disp*]
010	SS:[BP+SI+*disp*]
011	SS:[BP+DI+*disp*]
100	DS:[SI+*disp*]
101	DS:[DI+*disp*]
110	SS:[BP+*disp*]*
111	DS:[BX+*disp*]

* If *mod* is 00 and *r/m* is 110, then the operand is treated as a direct memory operand. This means that the operand `[BP]` is encoded as `[BP+0]` rather than having a short-form like other register indirect operands. Encoding `[BX]` takes one byte, but encoding `[BP]` takes two.

disp Displacement. These bytes give the offset for memory operands. The possible lengths (in bytes) are shown in parentheses.

data Data. These bytes give the actual value for constant values. The possible lengths (in bytes) are shown in parentheses.

If a memory operand has a segment override, the entire instruction has one of the following bytes as a prefix:

Prefix	**Segment**
00101110 (2Eh)	CS
00111110 (3Eh)	DS
00100110 (26h)	ES
00110110 (36h)	SS
01100100 (64h)	FS
01100101 (65h)	GS

Example

As an example, assume you want to calculate the encoding for the following statement (where `warray` is a 16-bit variable):

```
add     warray[bx+di], -3
```

First look up the encoding for the immediate-to-memory syntax of the **ADD** instruction:

100000*sw mod*,000,*r/m disp (0, 1, or 2) data (0, 1, or 2)*

Since the destination is a word operand, the *w* bit is set. The 8-bit immediate data must be sign-extended to 16 bits to fit into the operand, so the *s* bit is also set. The first byte of the instruction is therefore 10000011 (83h).

Since the memory operand can be anywhere in the segment, it must have a 16-bit offset (displacement). Therefore the *mod* field is 10. The *reg* field is 000, as shown in the encoding. The *r/m* coding for [bx+di+*disp*] is 001. The second byte is 10000001 (81h).

The next two bytes are the offset of `warray`. The low byte of the offset is stored first and the high byte second. For this example, assume that `warray` is located at offset 10EFh.

The last byte of the instruction is used to store the 8-bit immediate value –3 (FDh). This value is encoded as 8 bits (but sign-extended to 16 bits by the processor).

The encoding is shown here in hexadecimal:

83 81 EF 10 FD

You can confirm this by assembling the instruction and looking at the resulting assembly listing.

Interpreting 80386–80486 Encoding Extensions

This book shows 80386–80486 encodings for instructions that are available only on the 80386–80486 processors. For other instructions, encodings are shown only for the 16-bit subset available on all processors. This section tells how to convert the 80286 encodings shown in the book to 80386–80486 encodings that use extensions such as 32-bit registers and memory operands.

The extended 80386–80486 encodings differ in that they can have additional prefix bytes, a Scaled Index Base (SIB) byte, and 32-bit displacement and immediate bytes. Use of these elements is closely tied to the segment word size. The use type of the code segment determines whether the instructions are processed in 32-bit mode (**USE32**) or 16-bit mode (**USE16**). Current versions of MS-DOS® and Microsoft® Windows™ use 16-bit mode only. Windows NT uses 32-bit mode.

The bytes that can appear in an instruction encoding are:

16-Bit Encoding

Opcode	*mod-reg-r/m*	*disp*	*immed*
(1-2)	(0-1)	(0-2)	(0-2)

32-Bit Encoding

Address-Size (67h)	Operand-Size (66h)	Opcode	*mod-reg-r/m*	Scaled Index Base	*disp*	*immed*
(0-1)	(0-1)	(1-2)	(0-1)	(0-1)	(0-4)	(0-4)

Additional bytes may be added for a segment prefix, a repeat prefix, or the **LOCK** prefix.

Address-Size Prefix

The address-size prefix determines the segment word size of the operation. It can override the default size for calculating the displacement of memory addresses. The address prefix byte is 67h. The assembler automatically inserts this byte where appropriate.

In 32-bit mode (**USE32** or **FLAT** code segment), displacements are calculated as 32-bit addresses. The effective address-size prefix must be used for any instructions that must calculate addresses as 16-bit displacements. In 16-bit mode, the defaults are reversed. The prefix must be used to specify calculation of 32-bit displacements.

Operand-Size Prefix

The operand-size prefix determines the size of operands. It can override the default size of registers or memory operands. The operand-size prefix byte is 66h. The assembler automatically inserts this byte where appropriate.

In 32-bit mode, the default sizes for operands are 8 bits and 32 bits (depending on the *w* bit). For most instructions, the operand-size prefix must be used for any instructions that use 16-bit operands. In 16-bit mode, the default sizes are 8 bits and 16 bits. The prefix must be used for any instructions that use 32-bit operands. Some instructions use 16-bit operands, regardless of mode.

Encoding Differences for 32-Bit Operations

When 32-bit operations are performed, the meaning of certain bits or fields is different from their meaning in 16-bit operations. The changes may affect default operations in 32-bit mode, or 16-bit mode operations in which the address-size prefix or the operand-size prefix is used. The following fields may

have a different meaning for 32-bit operations from their meaning as described in the “Interpreting Encodings” section:

w Word/byte bit. If set, use 32-bit operands. If clear, use 8-bit operands.

s Sign bit. If set, sign-extend 8-bit and 16-bit immediate data to 32 bits.

mod Mode. This field indicates the register/memory mode. The value 11 still indicates a register-to-register operation with *r/m* containing the code for a 32-bit source register. However, other codes have different meanings as shown in the tables in the next section.

reg Register. The codes for 16-bit registers are extended to 32-bit registers. For example, if the *reg* field is 000, EAX is used instead of AX. Use of 8-bit registers is unchanged.

sreg Segment register. The 80386 has the following additional segment registers:

sreg	**Register**
100	FS
101	GS

r/m Register/memory. If the *r/m* field is used for the source register, 32-bit registers are used as for the *reg* field. If the field is used for memory operands, the meaning is completely different from the meaning used for 16-bit operations, as shown in the tables in the next section.

disp Displacement. This field is 4 bytes for 32-bit addresses.

data Data. Immediate data can be up to 4 bytes.

Scaled Index Base Byte

Many 80386–80486 extended memory operands are too complex to be represented by a single *mod-reg-r/m* byte. For these operands, a value of 100 in the *r/m* field signals the presence of a second encoding byte called the Scaled Index Base (SIB) byte. The SIB byte is made up of the following fields:

ss index base

ss Scaling Field. This two-bit field specifies one of the following scaling factors:

ss	**Scale**
00	1
01	2
10	4

11	8

index Index Register. This three-bit field specifies one of the following index registers:

index	**Register**
000	*EA*X
001	ECX
010	EDX
011	EBX
100	no index
101	EBP
110	ESI
111	EDI

Note ESP cannot be an index register. If the *index* field is 100, the *ss* field must be 00.

base Base Register. This 3-bit field combines with the *mod* field to specify the base register and the displacement. Note that the *base* field only specifies the base when the *r/m* field is 100. Otherwise, the *r/m* field specifies the base.

The possible combinations of the mod, r/m, scale, index, and base fields are as follows:

If a memory operand has a segment override, the entire instruction has one of the prefixes discussed in the preceding section, “Interpreting Encodings,” or one of the following prefixes for the segment registers available only on the 80386–80486:

Prefix		Segment
01100100	(64h)	FS
01100101	(65h)	GS

Example

Assume you want to calculate the encoding for the following statement (where `warray` is a 16-bit variable). Assume that the instruction is used in 16-bit mode.

```
add     warray[eax+ecx*2], -3
```

First look up the encoding for the immediate-to-memory syntax of the **ADD** instruction:

100000*sw* *mod*,000,*r/m* *disp (0, 1, or 2)* *data (1 or 2)*

This encoding must be expanded to account for 80386–80486 extensions. Note that the instruction operates on 16-bit data in a 16-bit mode program. Therefore, the operand-size prefix is not needed. However, the instruction does use 32-bit

registers to calculate a 32-bit effective address. Thus the first byte of the encoding must be the effective address-size prefix, 01100111 (67h).

The *opcode* byte is the same (83h) as for the 80286 example described in the "Interpreting Encodings" section.

The *mod-reg-r/m* byte must specify a based indexed operand with a scaling factor of two. This operand cannot be specified with a single byte, so the encoding must also use the SIB byte. The value 100 in the *r/m* field specifies an SIB byte. The *reg* field is 000, as shown in the encoding. The *mod* field is 10 for operands that have base and scaled index registers and a 32-bit displacement. The combined *mod*, *reg*, and *r/m* fields for the second byte are 10000100 (84h).

The SIB byte is next. The scaling factor is 2, so the *ss* field is 01. The index register is ECX, so the *index* field is 001. The base register is EAX, so the *base* field is 000. The SIB byte is 01001000 (48h).

The next 4 bytes are the offset of `warray`. The low bytes are stored first. For this example, assume that `warray` is located at offset 10EFh. This offset only requires 2 bytes, but 4 must be supplied because of the addressing mode. A 32-bit address can be safely used in 16-bit mode as long as the upper word is 0.

The last byte of the instruction is used to store the 8-bit immediate value –3 (FDh). The encoding is shown here in hexadecimal:

67 83 84 48 00 00 EF 10 FD

Instructions

This section provides an alphabetical reference to the instructions for the 8086, 8088, 80286, 80386, and 80486 processors.

AAA ASCII Adjust After Addition

Adjusts the result of an addition to a decimal digit (0–9). The previous addition instruction should place its 8-bit sum in AL. If the sum is greater than 9h, AH is incremented and the carry and auxiliary carry flags are set. Otherwise, the carry and auxiliary carry flags are cleared.

Flags

O	D	I	T	S	Z	A	P	C
?				?	?	±	?	±

Encoding 00110111

Syntax	Examples	CPU	Clock Cycles
AAA	`aaa`	88/86	8
		286	3
		386	4
		486	3

AAD ASCII Adjust Before Division

Converts unpacked BCD digits in AH (most significant digit) and AL (least significant digit) to a binary number in AX. This instruction is often used to prepare an unpacked BCD number in AX for division by an unpacked BCD digit in an 8-bit register.

Flags

O	D	I	T	S	Z	A	P	C
?				±	±	?	±	?

Encoding 11010101 00001010

Syntax	Examples	CPU	Clock Cycles
AAD	`aad`	88/86	60
		286	14
		386	19
		486	14

AAM ASCII Adjust After Multiply

Converts an 8-bit binary number less than 100 decimal in AL to an unpacked BCD number in AX. The most significant digit goes in AH and the least significant in AL. This instruction is often used to adjust the product after a **MUL** instruction that multiplies unpacked BCD digits in AH and AL. It is also used to adjust the quotient after a **DIV** instruction that divides a binary number less than 100 decimal in AX by an unpacked BCD number.

Flags

O	D	I	T	S	Z	A	P	C
?				±	±	?	±	?

Encoding

11010100 00001010

Syntax	Examples	CPU	Clock Cycles
AAM	`aam`	88/86	83
		286	16
		386	17
		486	15

AAS ASCII Adjust After Subtraction

Adjusts the result of a subtraction to a decimal digit (0–9). The previous subtraction instruction should place its 8-bit result in AL. If the result is greater than 9h, AH is decremented and the carry and auxiliary carry flags are set. Otherwise, the carry and auxiliary carry flags are cleared.

Flags

O	D	I	T	S	Z	A	P	C
?				?	?	±	?	±

Encoding

00111111

Syntax	Examples	CPU	Clock Cycles
AAS	`aas`	88/86	8
		286	3
		386	4
		486	3

ADC Add with Carry

Adds the source operand, the destination operand, and the value of the carry flag. The result is assigned to the destination operand. This instruction is used to add the more significant portions of numbers that must be added in multiple registers.

Flags

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	±

Encoding 000100*dw* *mod,reg,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
ADC *reg,reg*	`adc dx,cx`	88/86	3
		286	2
		386	2
		486	1
ADC *mem,reg*	`adc WORD PTR m32[2],dx`	88/86	16+*EA* (*W88*=24+*EA*)
		286	7
		386	7
		486	3
ADC *reg,mem*	`adc dx,WORD PTR m32[2]`	88/86	9+*EA* (*W88*=13+*EA*)
		286	7
		386	6
		486	2

Encoding 100000*sw* *mod, 010,r/m* *disp (0, 1, or 2)* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
ADC *reg,immed*	`adc dx,12`	88/86	4
		286	3
		386	2
		486	1
ADC *mem,immed*	`adc WORD PTR m32[2],16`	88/86	17+*EA* (*W88*=23+*EA*)
		286	7
		386	7
		486	3

Encoding 0001010*w* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
ADC *accum,immed*	`adc ax,5`	88/86	4
		286	3
		386	2
		486	1

ADD Add

Adds the source and destination operands and puts the sum in the destination operand.

Flags

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	±

Encoding 000000*dw* *mod,reg,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
ADD *reg,reg*	`add ax,bx`	88/86	3
		286	2
		386	2
		486	1
ADD *mem, reg*	`add total, cx`	88/86	16+*EA* (*W88*=24+*EA*)
	`add array[bx+di], dx`	286	7
		386	7
		486	3
ADD *reg,mem*	`add cx,incr`	88/86	9+*EA* (*W88*=13+*EA*)
	`add dx,[bp+6]`	286	7
		386	6
		486	2

Encoding 100000*sw* *mod, 000,r/m* *disp (p,1, or2)* *data (1or2)*

Syntax	Examples	CPU	Clock Cycles
ADD *reg,immed*	`add bx,6`	88/86	4
		286	3
		386	2
		486	1
ADD *mem,immed*	`add amount,27`	88/86	17+*EA* (*W88*=23+*EA*)
	`add pointers[bx][si],6`	286	7
		386	7
		486	3

Encoding 0000010*w* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
ADD *accum,immed*	`add ax,10`	88/86	4
		286	3
		386	2
		486	1

AND Logical AND

Performs a bitwise AND operation on the source and destination operands and stores the result in the destination operand. For each bit position in the operands, if both bits are set, the corresponding bit of the result is set. Otherwise, the corresponding bit of the result is cleared.

Flags

O	D	I	T	S	Z	A	P	C
0				±	±	?	±	0

Encoding 001000*dw* *mod,reg,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
AND *reg,reg*	`and dx,bx`	88/86	3
		286	2
		386	2
		486	1
AND *mem,reg*	`and bitmask,bx`	88/86	16+*EA* (*W88*=24+*EA*)
	`and [bp+2],dx`	286	7
		386	7
		486	3
AND *reg,mem*	`and bx,masker`	88/86	9+*EA* (*W88*=13+*EA*)
	`and dx,marray[bx+di]`	286	7
		386	6
		486	2

Encoding 100000*sw* *mod, 100, r/m* *disp (0, 1, or 2)* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
AND *reg,immed*	`and dx,0F7h`	88/86	4
		286	3
		386	2
		486	1
AND *mem,immed*	`and masker, 100lb`	88/86	17+*EA*(*W88*=24+*EA*)
		286	7
		386	7
		486	3

Encoding 0010010*w* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
AND *accum,immed*	`and ax,0B6h`	88/86	4
		286	3
		386	2
		486	1

ARPL Adjust Requested Privilege Level

80286–80486 Protected Only Verifies that the destination Requested Privilege Level (RPL) field (bits 0 and 1 of a selector value) is less than the source RPL field. If it is not, **ARPL** adjusts the destination RPL to match the source RPL. The destination operand should be a 16-bit memory or register operand containing the value of a selector. The source operand should be a 16-bit register containing the test value. The zero flag is set if the destination is adjusted; otherwise, the flag is cleared. **ARPL** is useful only in 80286–80486 protected mode. See Intel documentation for details on selectors and privilege levels.

Flags

O	D	I	T	S	Z	A	P	C
					±			

Encoding

01100011 *mod,reg,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
ARPL *reg,reg*	`arpl ax,cx`	88/86	—
		286	10
		386	20
		486	9
ARPL *mem,reg*	`arpl selector,dx`	88/86	—
		286	11
		386	21
		486	9

BOUND Check Array Bounds

80286–80486 Only Verifies that a signed index value is within the bounds of an array. The destination operand can be any 16-bit register containing the index to be checked. The source operand must then be a 32-bit memory operand in which the low and high words contain the starting and ending values, respectively, of the array. (On the 80386–80486 processors, the destination operand can be a 32-bit register; in this case, the source operand must be a 64-bit operand made up of 32-bit bounds.) If the source operand is less than the first bound or greater than the last bound, an interrupt 5 is generated. The instruction pointer pushed by the interrupt (and returned by **IRET**) points to the **BOUND** instruction rather than to the next instruction.

Flags

No change

Encoding 01100010 *mod,reg, r/m* *disp (2)*

Syntax	Examples	CPU	Clock Cycles
BOUND *reg16,mem32*	`bound di,base-4`	88/86	—
BOUND *reg32,mem64**		286	*noj*=13†
		386	*noj*=10†
		486	*noj*=7

* 80386–80486 only.

† See **INT** for timings if interrupt 5 is called.

BSF/BSR Bit Scan

80386–80486 Only Scans an operand to find the first set bit. If a set bit is found, the zero flag is cleared and the destination operand is loaded with the bit index of the first set bit encountered. If no set bit is found, the zero flag is set. **BSF** (Bit Scan Forward) scans from bit 0 to the most significant bit. **BSR** (Bit Scan Reverse) scans from the most significant bit of an operand to bit 0.

Flags

O	D	I	T	S	Z	A	P	C
					±			

Encoding 00001111 10111100 *mod, reg, r/m* *disp (0, 1, 2, or 4)*

Syntax	Examples	CPU	Clock Cycles
BSF *reg16,reg16*	`bsf cx,bx`	88/86	—
BSF *reg32,reg32*		286	—
		386	10+3*n**
		486	6–42†
BSF *reg16,mem16*	`bsf ecx,bitmask`	88/86	—
BSF *reg32,mem32*		286	—
		386	10+3*n**
		486	7–43§

Encoding 00001111 10111101 *mod, reg, r/m* *disp (0, 1, 2, or 4)*

Syntax	Examples	CPU	Clock Cycles
BSR *reg16,reg16*	`bsr cx,dx`	88/86	—
BSR *reg32,reg32*		286	—
		386	10+3*n**
		486	103 – 3*n*#
BSR *reg16,mem16*	`bsr eax,bitmask`	88/86	—
BSR *reg32,mem32*		286	—
		386	10+3*n**
		486	104 – 3*n*#

* *n* = bit position from 0 to 31.
clocks = 6 if second operand equals 0.

† Clocks = 8 +
4 for each byte scanned +
3 for each nibble scanned +
3 for each bit scanned in last nibble
or 6 if second operand equals 0.

§ Same as footnote above, but add 1 clock.

# *n* = bit position from 0 to 31.
clocks = 7 if second operand equals 0.

BSWAP Byte Swap

80486 Only Takes a single 32-bit register as operand and exchanges the first byte with the fourth, and the second byte with the third. This instruction does not alter any bit values within the bytes and is useful for quickly translating between 8086-family byte storage and storage schemes in which the high byte is stored first.

Flags No change

Encoding 00001111 11001 *reg*

Syntax	Examples	CPU	Clock Cycles
BSWAP *reg32*	`bswap eax`	88/86	—
	`bswap ebx`	286	—
		386	—
		486	1

BT/BTC/BTR/BTS Bit Tests

80386–80486 Only Copies the value of a specified bit into the carry flag, where it can be tested by a **JC** or **JNC** instruction. The destination operand specifies the value in which the bit is located; the source operand specifies the bit position. **BT** simply copies the bit to the flag. **BTC** copies the bit and complements (toggles) it in the destination. **BTR** copies the bit and resets (clears) it in the destination. **BTS** copies the bit and sets it in the destination.

Flags

O	D	I	T	S	Z	A	P	C
								±

Encoding 00001111 10111010 *mod, BBB*,r/m* *disp (0, 1, 2, or 4)* *data (1)*

Syntax	Examples	CPU	Clock Cycles
BT *reg16,immed8*†	`bt ax,4`	88/86	—
		286	—
		386	3
		486	3
BTC *reg16,immed8*†	`bts ax,4`	88/86	—
BTR *reg16,immed8*†	`btr bx,17`	286	—
BTS *reg16,immed8*†	`btc edi,4`	386	6
		486	6
BT *mem16,immed8*†	`btr DWORD PTR [si],27`	88/86	—
		286	—
	`btc color[di],4`	386	6
		486	3
BTC *mem16,immed8*†	`btc DWORD PTR [bx],27`	88/86	—
BTR *mem16,immed8*†		286	—
BTS *mem16,immed8*†	`btc maskit,4`	386	8
	`btr color[di],4`	486	8

Encoding 00001111 10*BBB*011* *mod, reg, r/m* *disp (0, 1, 2, or 4)*

Syntax	Examples	CPU	Clock Cycles
BT *reg16,reg16*†	`bt ax,bx`	88/86	—
		286	—
		386	3
		486	3
BTC *reg16,reg16*†	`btc eax,ebx`	88/86	—
BTR *reg16,reg16*†	`bts bx,ax`	286	—
BTS *reg16,reg16*†	`btr cx,di`	386	6
		486	6

Syntax	Examples	CPU	Clock Cycles
BT *mem16,reg16*†	`bt [bx],dx`	88/86	—
		286	—
		386	12
		486	8
BTC *mem16,reg16*†	`bts flags[bx],cx`	88/86	—
BTR *mem16,reg16*†	`btr rotate,cx`	286	—
BTS *mem16,reg16*†	`btc [bp+8],si`	386	13
		486	13

* *BBB* is 100 for **BT**, 111 for **BTC**, 110 for **BTR**, and 101 for **BTS**.

† Operands also can be 32 bits (*reg32* and *mem32*).

CALL Call Procedure

Calls a procedure. The instruction pushes the address of the next instruction onto the stack and jumps to the address specified by the operand. For **NEAR** calls, the offset (IP) is pushed and the new offset is loaded into IP.

For **FAR** calls, the segment (CS) is pushed and the new segment is loaded into CS. Then the offset (IP) is pushed and the new offset is loaded into IP. A subsequent **RET** instruction can pop the address so that execution continues with the instruction following the call.

Flags

No change

Encoding

11101000 *disp (2)*

Syntax	Examples	CPU	Clock Cycles
CALL *label*	`call upcase`	88/86	19 (*88*=23)
		286	7+*m*
		386	7+*m*
		486	3

Encoding

10011010 *disp (4)*

Syntax	Examples	CPU	Clock Cycles
CALL *label*	`call FAR PTR job`	88/86	28 (*88*=36)
	`call distant`	286	13+*m*,*pm*=26+*m**
		386	17+*m*,*pm*=34+*m**
		486	18,*pm*=20*

Encoding 11111111 *mod*,010,*r/m*

Syntax	Examples	CPU	Clock Cycles
CALL *reg*	`call ax`	88/86 286 386 486	16 (*88*=20) 7+*m* 7+*m* 5
CALL *mem16*	`call pointer`	88/86	21+*EA* (*88*=29+*EA*)
CALL *mem32*†	`call [bx]`	286 386 486	11+*m* 10+*m* 5

Encoding 11111111 *mod*,011,*r/m*

Syntax	Examples	CPU	Clock Cycles
CALL *mem32*	`call far_table[di]`	88/86	37+*EA* (*88*=53+*EA*)
CALL *mem48*†	`call DWORD PTR [bx]`	286 386 486	16+*m*,*pm*=29+*m** 22+*m*,*pm*=38+*m** 17,*pm*=20*

* Timings for calls through call and task gates are not shown, since they are used primarily in operating systems.

† 80386–80486 32-bit addressing mode only.

CBW Convert Byte to Word

Converts a signed byte in AL to a signed word in AX by extending the sign bit of AL into all bits of AH.

Flags No change

Encoding 10011000*

Syntax	Examples	CPU	Clock Cycles
CBW	`cbw`	88/86 286 386 486	2 2 3 3

* **CBW** and **CWDE** have the same encoding with two exceptions: in 32-bit mode, **CBW** is preceded by the operand-size byte (66h) but **CWDE** is not; in 16-bit mode, **CWDE** is preceded by the operand-size byte but **CBW** is not.

CDQ Convert Double to Quad

80386–80486 Only Converts the signed doubleword in EAX to a signed quadword in the EDX:EAX register pair by extending the sign bit of EAX into all bits of EDX.

Flags No change

Encoding 10011001*

Syntax	Examples	CPU	Clock Cycles
CDQ	`cdq`	88/86	—
		286	—
		386	2
		486	3

* **CWD** and **CDQ** have the same encoding with two exceptions: in 32-bit mode, **CWD** is preceded by the operand-size byte (66h) but **CDQ** is not; in 16-bit mode, **CDQ** is preceded by the operand-size byte but **CWD** is not.

CLC Clear Carry Flag

Clears the carry flag.

Flags

O	D	I	T	S	Z	A	P	C
								0

Encoding 11111000

Syntax	Examples	CPU	Clock Cycles
CLC	`clc`	88/86	2
		286	2
		386	2
		486	2

CLD Clear Direction Flag

Clears the direction flag. All subsequent string instructions will process up (from low addresses to high addresses) by increasing the appropriate index registers.

Flags

O D I T S Z A P C
0

Encoding

11111100

Syntax	Examples	CPU	Clock Cycles
CLD	`cld`	88/86	2
		286	2
		386	2
		486	2

CLI Clear Interrupt Flag

Clears the interrupt flag. When the interrupt flag is cleared, maskable interrupts are not recognized until the flag is set again with the **STI** instruction. In protected mode, **CLI** clears the flag only if the current task's privilege level is less than or equal to the value of the IOPL flag. Otherwise, a general-protection fault occurs.

Flags

O D I T S Z A P C
0

Encoding

11111010

Syntax	Examples	CPU	Clock Cycles
CLI	`cli`	88/86	2
		286	3
		386	3
		486	5

CLTS Clear Task-Switched Flag

80286–80486 Privileged Only Clears the task-switched flag in the Machine Status Word (MSW) of the 80286, or the CR0 register of the 80386–80486. This instruction can be used only in system software executing at privilege level

0. See Intel documentation for details on the task-switched flag and other privileged-mode concepts.

Flags No change

Encoding 00001111 00000110

Syntax	Examples	CPU	Clock Cycles
CLTS	`clts`	88/86	—
		286	2
		386	5
		486	7

CMC Complement Carry Flag

Complements (toggles) the carry flag.

Flags

O	D	I	T	S	Z	A	P	C
								±

Encoding 11110101

Syntax	Examples	CPU	Clock Cycles
CMC	`cmc`	88/86	2
		286	2
		386	2
		486	2

CMP Compare Two Operands

Compares two operands as a test for a subsequent conditional-jump or set instruction. **CMP** does this by subtracting the source operand from the destination operand and setting the flags according to the result. **CMP** is the same as the **SUB** instruction, except that the result is not stored.

Flags

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	±

Encoding 001110*dw* *mod, reg, r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
CMP *reg,reg*	`cmp di,bx`	88/86	3
	`cmp dl,cl`	286	2
		386	2
		486	1
CMP *mem,reg*	`cmp maximum,dx`	88/86	9+*EA*
	`cmp array[si],bl`	286	(*W88*=13+*EA*)
		386	7
		486	5
			2
CMP *reg,mem*	`cmp dx,minimum`	88/86	9+*EA*
	`cmp bh,array[si]`	286	(*W88*=13+*EA*)
		386	6
		486	6
			2

Encoding 100000*sw* *mod, 111,r/m* *disp (0, 1, or 2)* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
CMP *reg,immed*	`cmp bx,24`	88/86	4
		286	3
		386	2
		486	1
CMP *mem,immed*	`cmp WORD PTR [di],4`	88/86	10+*EA*
	`cmp tester,4000`	286	(*W88*=14+*EA*)
		386	6
		486	5
			2

Encoding 0011110*w* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
CMP *accum,immed*	`cmp ax,1000`	88/86	4
		286	3
		386	2
		486	1

CMPS/CMPSB/CMPSW/CMPSD Compare String

Compares two strings. DS:SI must point to the source string and ES:DI must point to the destination string (even if operands are given). For each comparison, the destination element is subtracted from the source element and the flags are updated to reflect the result (although the result is not stored). DI and SI are adjusted according to the size of the operands and the status of the direction flag. They are increased if the direction flag has been cleared with **CLD**, or decreased if the direction flag has been set with **STD**.

If the **CMPS** form of the instruction is used, operands must be provided to indicate the size of the data elements to be processed. A segment override can be given for the source (but not for the destination). If **CMPSB** (bytes), **CMPSW** (words), or **CMPSD** (doublewords on the 80386–80486 only) is used, the instruction determines the size of the data elements to be processed.

CMPS and its variations are normally used with repeat prefixes. **REPNE** (or **REPNZ**) is used to find the first match between two strings. **REPE** (or **REPZ**) is used to find the first mismatch. Before the comparison, CX should contain the maximum number of elements to compare. After a **REPNE CMPS**, the zero flag is clear if no match was found. After a **REPE CMPS**, the zero flag is set if no mismatch was found.

When the instruction finishes, ES:DI and DS:SI point to the element that follows (if the direction flag is clear) or precedes (if the direction flag is set) the match or mismatch. If CX decrements to 0, ES:DI and DS:SI point to the element that follows or precedes the last comparison. The zero flag is set or clear according to the result of the last comparison, not according to the value of CX.

Flags

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	±

Encoding

1010011*w*

Syntax	Examples		CPU	Clock Cycles
CMPS ⟦*segreg*:⟧ *src*, ⟦**ES:**⟧ *dest*	`cmps`	`source,es:dest`	88/86	22 (*W88*=30)
CMPSB ⟦⟦*segreg*:⟦ *src*, ⟧**ES:**⟧ *dest*⟧	`repne`	`cmpsw`	286	8
CMPSW ⟦⟦*segreg*:⟦ *src*, ⟧**ES:**⟧ *dest*⟧	`repe`	`cmpsb`	386	10
CMPSD ⟦⟦*segreg*:⟦ *src*, ⟧**ES:**⟧ *dest*⟧	`repne`	`cmpsd`	486	8

CMPXCHG Compare and Exchange

80486 Only Compares the destination operand to the accumulator (AL, AX, or EAX). If equal, the source operand is copied to the destination. Otherwise, the destination is copied to the accumulator. The instruction sets flags according to the result of the comparison.

Flags

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	±

Encoding

00001111 1011000*b* *mod, reg, r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
CMPXCHG *mem,reg*	`cmpxchg warr[bx],cx`	88/86	—
	`cmpxchg string,bl`	286	—
		386	—
		486	7–10
CMPXCHG *reg,reg*	`cmpxchg dl,cl`	88/86	—
	`cmpxchg bx,dx`	286	—
		386	—
		486	6

CWD Convert Word to Double

Converts the signed word in AX to a signed doubleword in the DX:AX register pair by extending the sign bit of AX into all bits of DX.

Flags

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	±

Encoding

10011001*

Syntax	Examples	CPU	Clock Cycles
CWD	`cwd`	88/86	5
		286	2
		386	2
		486	3

* **CWD** and **CDQ** have the same encoding with two exceptions: in 32-bit mode, **CWD** is preceded by the operand-size byte (66h) but **CDQ** is not; in 16-bit mode, **CDQ** is preceded by the operand-size byte but **CWD** is not.

CWDE Convert Word to Extended Double

80386–80486 Only Converts a signed word in AX to a signed doubleword in EAX by extending the sign bit of AX into all bits of EAX.

Flags No change

Encoding 10011000*

Syntax	Examples	CPU	Clock Cycles
CWDE	`cwde`	88/86	—
		286	—
		386	3
		486	3

* **CBW** and **CWDE** have the same encoding with two exceptions: in 32-bit mode, **CBW** is preceded by the operand-size byte (66h) but **CWDE** is not; in 16-bit mode, **CWDE** is preceded by the operand-size byte but **CBW** is not.

DAA Decimal Adjust After Addition

Adjusts the result of an addition to a packed BCD number (less than 100 decimal). The previous addition instruction should place its 8-bit binary sum in AL. **DAA** converts this binary sum to packed BCD format with the least significant decimal digit in the lower four bits and the most significant digit in the upper four bits. If the sum is greater than 99h after adjustment, the carry and auxiliary carry flags are set. Otherwise, the carry and auxiliary carry flags are cleared.

Flags

O	D	I	T	S	Z	A	P	C
?				±	±	±	±	±

Encoding 00100111

Syntax	Examples	CPU	Clock Cycles
DAA	`daa`	88/86	4
		286	3
		386	4
		486	2

DAS Decimal Adjust After Subtraction

Adjusts the result of a subtraction to a packed BCD number (less than 100 decimal). The previous subtraction instruction should place its 8-bit binary result in AL. **DAS** converts this binary sum to packed BCD format with the least significant decimal digit in the lower four bits and the most significant digit in the upper four bits. If the sum is greater than 99h after adjustment, the carry and auxiliary carry flags are set. Otherwise, the carry and auxiliary carry flags are cleared.

Flags

O	D	I	T	S	Z	A	P	C
?				±	±	±	±	±

Encoding

00101111

Syntax	Examples	CPU	Clock Cycles
DAS	`das`	88/86	4
		286	3
		386	4
		486	2

DEC Decrement

Subtracts 1 from the destination operand. Because the operand is treated as an unsigned integer, the **DEC** instruction does not affect the carry flag. To detect any effects on the carry flag, use the **SUB** instruction.

Flags

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	

Encoding

1111111*w* *mod*, 001,*r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
DEC *reg8*	`dec cl`	88/86	3
		286	2
		386	2
		486	1
DEC *mem*	`dec counter`	88/86	15+*EA* (*W88*=23+*EA*)
		286	7
		386	6
		486	3

Encoding 01001 *reg*

Syntax	Examples	CPU	Clock Cycles
DEC *reg16*	`dec ax`	88/86	3
DEC *reg32**		286	2
		386	2
		486	1

* 80386–80486 only.

DIV Unsigned Divide

Divides an implied destination operand by a specified source operand. Both operands are treated as unsigned numbers. If the source (divisor) is 16 bits wide, the implied destination (dividend) is the DX:AX register pair. The quotient goes into AX and the remainder into DX. If the source is 8 bits wide, the implied destination operand is AX. The quotient goes into AL and the remainder into AH. On the 80386–80486, if the source is EAX, the quotient goes into EAX and the remainder into EDX.

Flags

O	D	I	T	S	Z	A	P	C
?				?	?	?	?	?

Encoding 1111011*w* *mod, 110,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
DIV *reg*	`div cx`	88/86	*b*=80–90,*w*=144–162
	`div dl`	286	*b*=14,*w*=22
		386	*b*=14,*w*=22,*d*=38
		486	*b*=16,*w*=24,*d*=40
DIV *mem*	`div [bx]`	88/86	(*b*=86–96,*w*=150–168)+*EA**
	`div fsize`	286	*b*=17,*w*=25
		386	*b*=17,*w*=25,*d*=41
		486	*b*=16,*w*=24,*d*=40

* Word memory operands on the 8088 take (158–176)+*EA* clocks.

ENTER Make Stack Frame

80286-80486 Only Creates a stack frame for a procedure that receives parameters passed on the stack. When *immed16* is 0, **ENTER** is equivalent to `push bp`, followed by `mov bp,sp`. The first operand of the **ENTER** instruction specifies the number of bytes to reserve for local variables. The second operand specifies the nesting level for the procedure. The nesting level should be 0 for languages that do not allow access to local variables of higher-level procedures (such as C, Basic, and FORTRAN). See the complementary instruction **LEAVE** for a method of exiting from a procedure.

Flags No change

Encoding 11001000 *data (2)* *data (1)*

Syntax	Examples	CPU	Clock Cycles
ENTER *immed16*,**0**	`enter 4,0`	88/86	—
		286	11
		386	10
		486	14
ENTER *immed16*,**1**	`enter 0,1`	88/86	—
		286	15
		386	12
		486	17
ENTER *immed16*,*immed8*	`enter 6,4`	88/86	—
		286	$12+4(n-1)$
		386	$15+4(n-1)$
		486	$17+3n$

HLT Halt

Stops CPU execution until an interrupt restarts execution at the instruction following **HLT**. In protected mode, this instruction works only in privileged mode.

Flags No change

Encoding 11110100

Syntax	Examples	CPU	Clock Cycles
HLT	`hlt`	88/86	2
		286	2
		386	5
		486	4

IDIV Signed Divide

Divides an implied destination operand by a specified source operand. Both operands are treated as signed numbers. If the source (divisor) is 16 bits wide, the implied destination (dividend) is the DX:AX register pair. The quotient goes into AX and the remainder into DX. If the source is 8 bits wide, the implied destination is AX. The quotient goes into AL and the remainder into AH. On the 80386–80486, if the source is EAX, the quotient goes into EAX and the remainder into EDX.

Flags

O	D	I	T	S	Z	A	P	C
?				?	?	?	?	?

Encoding

1111011*w* *mod, 111,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
IDIV *reg*	`idiv bx` `idiv dl`	88/86	*b*=101–112,*w*=165–184
		286	*b*=17,*w*=25
		386	*b*=19,*w*=27,*d*=43
		486	*b*=19,*w*=27,*d*=43
IDIV *mem*	`idiv itemp`	88/86	(*b*=107–118,*w*=171–190)+*EA**
		286	*b*=20,*w*=28
		386	*b*=22,*w*=30,*d*=46
		486	*b*=20,*w*=28,*d*=44

* Word memory operands on the 8088 take (175–194)+*EA* clocks.

IMUL Signed Multiply

Multiplies an implied destination operand by a specified source operand. Both operands are treated as signed numbers. If a single 16-bit operand is given, the implied destination is AX and the product goes into the DX:AX register pair. If a single 8-bit operand is given, the implied destination is AL and the product goes into AX. On the 80386–80486, if the operand is EAX, the product goes into the EDX:EAX register pair. The carry and overflow flags are set if the product is sign-extended into DX for 16-bit operands, into AH for 8-bit operands, or into EDX for 32-bit operands.

Two additional syntaxes are available on the 80186–80486 processors. In the two-operand form, a 16-bit register gives one of the factors and serves as the destination for the result; a source constant specifies the other factor. In the three-operand form, the first operand is a 16-bit register where the result will be stored, the second is a 16-bit register or memory operand containing one of the factors, and the third is a constant representing the other factor. With both variations, the overflow and carry flags are set if the result is too large to fit into the 16-bit destination register. Since the low 16 bits of the product are the same for both signed and unsigned multiplication, these syntaxes can be used for either signed or unsigned numbers. On the 80386–80486, the operands can be either 16 or 32 bits wide.

A fourth syntax is available on the 80386–80486. Both the source and destination operands can be given specifically. The source can be any 16- or 32-bit memory operand or general-purpose register. The destination can be any general-purpose register of the same size. The overflow and carry flags are set if the product does not fit in the destination.

Flags

O	D	I	T	S	Z	A	P	C
±				?	?	?	?	±

Encoding

1111011*w* *mod, 101,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
IMUL *reg*	`imul dx`	88/86	*b*=80–98,*w*=128–154
		286	*b*=13,*w*=21
		386	*b*=9–14,*w*=9–22,*d*=9–38*
		486	*b*=13–18,*w*=13–26,*d*=13–42
IMUL *mem*	`imul factor`	88/86	(*b*=86–104,*w*=134–160)+*EA*†
		286	*b*=16,*w*=24
		386	*b*=12–17,*w*=12–25,*d*=12–41*
		486	*b*=13–18,*w*=13–26, *d*=13–42

* The 80386–80486 processors have an early-out multiplication algorithm. Therefore, multiplying an 8-bit or 16-bit value in EAX takes the same time as multiplying the value in AL or AX.

† Word memory operands on the 8088 take (138–164)+*EA* clocks.

Encoding

011010*s*1 *mod, reg, r/m* *disp (0, 1, or 2)* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
IMUL *reg16,immed*	`imul cx,25`	88/86	—
IMUL *reg32,immed**		286	21
		386	*b*=9–14,*w*=9–22,*d*=9–38†
		486	*b*=13–18,*w*=13–26,*d*=13–42

IMUL *reg16,reg16,immed*	`imul`	88/86	—
IMUL *reg32,reg32,immed**	`dx,ax,18`	286	21
		386	*b*=9–14,*w*=9–22,*d*=9–38†
		486	*b*=13–18,*w*=13–26,*d*=13–42

Syntax	Examples	CPU	Clock Cycles
IMUL *reg16,mem16,immed*	`imul`	88/86	—
IMUL *reg32,mem32,immed**	`bx,[si],60`	286	24
		386	*b*=12–17,*w*=12–25,*d*=12–41†
		486	*b*=13–18,*w*=13–26,*d*=13–42

Encoding 00001111 10101111 *mod,reg,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
IMUL *reg16,reg16*	`imul cx,ax`	88/86	—
IMUL *reg32,reg32**		286	—
		386	*w*=9–22,*d*=9–38
		486	*b*=13–18,*w*=13–26,*d*=13–42
IMUL *reg16,mem16*	`imul`	88/86	—
IMUL *reg32,mem32**	`dx,[si]`	286	—
		386	*w*=12–25,*d*=12–41
		486	*b*=13–18,*w*=13–26,*d*=13–42

* 80386–80486 only.

† The variations depend on the source constant size; destination size is not a factor.

IN Input from Port

Transfers a byte or word (or doubleword on the 80386–80486) from a port to the accumulator register. The port address is specified by the source operand, which can be DX or an 8-bit constant. Constants can be used only for port numbers less than 255; use DX for higher port numbers. In protected mode, a general-protection fault occurs if **IN** is used when the current privilege level is greater than the value of the IOPL flag.

Flags No change

Encoding 1110010*w* *data (1)*

Syntax	Examples	CPU	Clock Cycles
IN *accum,immed*	`in ax,60h`	88/86	10 (*W88*=14)
		286	5
		386	12,*pm*=6,26*
		486	14,*pm*=9,29*†

Encoding 1110110*w*

Syntax	Examples	CPU	Clock Cycles
IN *accum*,**DX**	`in ax,dx`	88/86	8 (*W88*=12)
	`in al,dx`	286	5
		386	13,*pm*=7,27*
		486	14,*pm*=8,28*†

* First protected-mode timing: CPL ≤ IOPL. Second timing: CPL > IOPL.

† Takes 27 clocks in virtual 8086 mode.

INC Increment

Adds 1 to the destination operand. Because the operand is treated as an unsigned integer, the **INC** instruction does not affect the carry flag. If a signed carry requires detection, use the **ADD** instruction.

Flags

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	

Encoding 1111111*w* *mod,000,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
INC *reg8*	`inc cl`	88/86	3
		286	2
		386	2
		486	1
INC *mem*	`inc vpage`	88/86	15+*EA* (*W88*=23+*EA*)
		286	7
		386	6
		486	3

Encoding 01000 *reg*

Syntax	Examples	CPU	Clock Cycles
INC *reg16*	`inc bx`	88/86	3
INC *reg32**		286	2
		386	2
		486	1

* 80386–80486 only.

INS/INSB/INSW/INSD Input from Port to String

80286-80486 Only Receives a string from a port. The string is considered the destination and must be pointed to by ES:DI (even if an operand is given). The input port is specified in DX. For each element received, DI is adjusted according to the size of the operand and the status of the direction flag. DI is increased if the direction flag has been cleared with **CLD** or decreased if the direction flag has been set with **STD**.

If the **INS** form of the instruction is used, a destination operand must be provided to indicate the size of the data elements to be processed, and DX must be specified as the source operand containing the port number. A segment override is not allowed. If **INSB** (bytes), **INSW** (words), or **INSD** (doublewords on the 80386–80486 only) is used, the instruction determines the size of the data elements to be received.

INS and its variations are normally used with the **REP** prefix. Before the repeated instruction is executed, CX should contain the number of elements to be received. In protected mode, a general-protection fault occurs if **INS** is used when the current privilege level is greater than the value of the IOPL flag.

Flags No change

Encoding 0110110*w*

Syntax	Examples	CPU	Clock Cycles
INS ⟦**ES:**⟧ *dest*, **DX**	`ins es:instr,dx`	88/86	—
INSB ⟦⟦**ES:**⟧ *dest*, **DX**⟧	`rep insb`	286	5
INSW ⟦⟦**ES:**⟧ *dest*, **DX**⟧	`rep insw`	386	15,*pm*=9,29*
INSD ⟦⟦**ES:**⟧ *dest*, **DX**⟧	`rep insd`	486	17,*pm*=10,32*

* First protected-mode timing: CPL ≤ IOPL. Second timing: CPL > IOPL.

INT Interrupt

Generates a software interrupt. An 8-bit constant operand (0 to 255) specifies the interrupt procedure to be called. The call is made by indexing the interrupt number into the Interrupt Vector Table (IVT) starting at segment 0, offset 0. In real mode, the IVT contains 4-byte pointers to interrupt procedures. In privileged mode, the IVT contains 8-byte pointers.

When an interrupt is called in real mode, the flags, CS, and IP are pushed onto the stack (in that order), and the trap and interrupt flags are cleared. **STI** can be

used to restore interrupts. See Intel documentation and the documentation for your operating system for details on using and defining interrupts in privileged mode. To return from an interrupt, use the **IRET** instruction.

Flags

O D I T S Z A P C
0 0

Encoding 11001101 *data (1)*

Syntax	Examples	CPU	Clock Cycles
INT *immed8*	`int 25h`	88/86	51 (*88*=71)
		286	23+*m*,*pm*=(40,78)+*m**
		386	37,*pm*=59,99*
		486	30,*pm*=44,71*

Encoding 11001100

Syntax	Examples	CPU	Clock Cycles
INT 3	`int 3`	88/86	52 (*88*=72)
		286	23+*m*,*pm*=(40,78)+*m**
		386	33,*pm*=59,99*
		486	26,*pm*=44,71*

* The first protected-mode timing is for interrupts to the same privilege level. The second is for interrupts to a higher privilege level. Timings for interrupts through task gates are not shown.

INTO Interrupt on Overflow

Generates Interrupt 4 if the overflow flag is set. The default MS-DOS behavior for Interrupt 4 is to return without taking any action. For **INTO** to have any effect, you must define an interrupt procedure for Interrupt 4.

Flags

O D I T S Z A P C
± ±

Encoding 11001110

Syntax	Examples	CPU	Clock Cycles
INTO	`into`	88/86	53 (*88*=73),*noj*=4
		286	24+*m*,*noj*=3,*pm*=(40,78)+*m**
		386	35,*noj*=3,*pm*=59,99*
		486	28,*noj*=3,*pm*=46,73*

* The first protected-mode timing is for interrupts to the same privilege level. The second is for interrupts to a higher privilege level. Timings for interrupts through task gates are not shown.

INVD Invalidate Data Cache

80486 Only Empties contents of the current data cache without writing changes to memory. Proper use of this instruction requires knowledge of how contents are placed in the cache. **INVD** is intended primarily for system programming. See Intel documentation for details.

Flags No change

Encoding 00001111 00001000

Syntax	Examples	CPU	Clock Cycles
INVD	`invd`	88/86	—
		286	—
		386	—
		486	4

INVLPG Invalidate TLB Entry

80486 Only Invalidates an entry in the Translation Lookaside Buffer (TLB), used by the demand-paging mechanism in virtual-memory operating systems. The instruction takes a single memory operand and calculates the effective address of the operand, including the segment address. If the resulting address is mapped by any entry in the TLB, this entry is removed. Proper use of **INVLPG** requires understanding the hardware-supported demand-paging mechanism. **INVLPG** is intended primarily for system programming. See Intel documentation for details.

Flags No change

Encoding 00001111 00000001 *mod, reg, r/m* *disp (2)*

Syntax	Examples	CPU	Clock Cycles
INVLPG	`invlpg pointer[bx]`	88/86	—
	`invlpg es:entry`	286	—
		386	—
		486	12*

* 11 clocks if address is not mapped by any TLB entry.

IRET/IRETD Interrupt Return

Returns control from an interrupt procedure to the interrupted code. In real mode, the **IRET** instruction pops IP, CS, and the flags (in that order) and resumes execution. See Intel documentation for details on **IRET** operation in privileged mode. On the 80386–80486, the **IRETD** instruction should be used to pop a 32-bit instruction pointer when returning from an interrupt called from a 32-bit segment. The **F** suffix prevents epilogue code from being generated when ending a **PROC** block. Use it to terminate interrupt service procedures.

Flags

O	D	I	T	S	Z	A	P	C
±	±	±	±	±	±	±	±	±

Encoding

11001111

Syntax	Examples	CPU	Clock Cycles
IRET	`iret`	88/86	32 (*88*=44)
IRETD*		286	17+*m*,*pm*=(31,55)+*m*†
IRETF		386	22,*pm*=38,82†
IRETDF*		486	15,*pm*=20,36

* 80386–80486 only.

† The first protected-mode timing is for interrupts to the same privilege level within a task. The second is for interrupts to a higher privilege level within a task. Timings for interrupts through task gates are not shown.

Jcondition Jump Conditionally

Transfers execution to the specified label if the flags condition is true. The *condition* is tested by checking the flags shown in the table on the following page. If *condition* is false, no jump is taken and program execution continues at the next instruction. On the 8086–80286 processors, the label given as the operand must be short (between –128 and +127 bytes from the instruction following the jump).* The 80386–80486 processors allow near jumps (–32,768 to +32,767 bytes). On the 80386–80486, the assembler generates the shortest jump possible, unless the jump size is explicitly specified.

When the 80386–80486 processors are in **FLAT** memory model, short jumps range from –128 to +127 bytes and near jumps range from –2 to +2 gigabytes. There are no far jumps.

Flags

No change

Encoding 0111*cond* *disp (1)*

Syntax	Examples	CPU	Clock Cycles
J*condition label*	`jg bigger`	88/86	16,*noj*=4
	`jo SHORT too_big`	286	7+*m*,*noj*=3
	`jpe p_even`	386	7+*m*,*noj*=3
		486	3,*noj*=1

Encoding 00001111 1000*cond* *disp (2)*

Syntax	Examples	CPU	Clock Cycles
J*condition label*†	`je next`	88/86	—
	`jnae lesser`	286	—
	`js negative`	386	7+*m*,*noj*=3
		486	3,*noj*=1

* If a source file for an 8086–80286 program contains a conditional jump outside the range of –128 to +127 bytes, the assembler emits a level 3 warning and generates two instructions (including an unconditional jump) that are the equivalent of the desired instruction. This behavior can be enabled and disabled with the **OPTION LJMP** and **OPTION NOLJMP** directives.

† Near labels are only available on the 80386–80486. They are the default.

Jump Conditions

Opcode*	Mnemonic	Flags Checked	Description
size 0010	**JB/JNAE**	CF=1	Jump if below/not above or equal (unsigned comparisons)
size 0011	**JAE/JNB**	CF=0	Jump if above or equal/not below (unsigned comparisons)
size 0110	**JBE/JNA**	CF=1 or ZF=1	Jump if below or equal/not above (unsigned comparisons)
size 0111	**JA/JNBE**	CF=0 and ZF=0	Jump if above/not below or equal (unsigned comparisons)
size 0100	**JE/JZ**	ZF=1	Jump if equal (zero)
size 0101	**JNE/JNZ**	ZF=0	Jump if not equal (not zero)
size 1100	**JL/JNGE**	SF_OF	Jump if less/not greater or equal (signed comparisons)
size 1101	**JGE/JNL**	SF=OF	Jump if greater or equal/not less (signed comparisons)
size 1110	**JLE/JNG**	ZF=1 or SF_OF	Jump if less or equal/not greater (signed comparisons)
size 1111	**JG/JNLE**	ZF=0 and SF=OF	Jump if greater/not less or equal (signed comparisons)
size 1000	**JS**	SF=1	Jump if sign
size 1001	**JNS**	SF=0	Jump if not sign

Opcode*	Mnemonic	Flags Checked	Description
size 0010	**JC**	CF=1	Jump if carry
size 0011	**JNC**	CF=0	Jump if not carry
size 0000	**JO**	OF=1	Jump if overflow
size 0001	**JNO**	OF=0	Jump if not overflow
size 1010	**JP/JPE**	PF=1	Jump if parity/parity even
size 1011	**JNP/JPO**	PF=0	Jump if no parity/parity odd

* The *size* bits are 0111 for short jumps or 1000 for 80386–80486 near jumps.

JCXZ/JECXZ Jump if CX is Zero

Transfers program execution to the specified label if CX is 0. On the 80386–80486, **JECXZ** can be used to jump if ECX is 0. If the count register is not 0, execution continues at the next instruction. The label given as the operand must be short (between –128 and +127 bytes from the instruction following the jump).

Flags

No change

Encoding

11100011 *disp (1)*

Syntax	Examples	CPU	Clock Cycles
JCXZ *label* **JECXZ** *label**	`jcxz not found`	88/86 286 386 486	18,*noj*=6 8+*m*,*noj*=4 9+*m*,*noj*=5 8,*noj*=5

* 80386–80486 only.

JMP Jump Unconditionally

Transfers program execution to the address specified by the destination operand. Jumps are near (between –32,768 and +32,767 bytes from the instruction following the jump), or short (between –128 and +127 bytes), or far (in a different code segment). Unless a distance is explicitly specified, the assembler selects the shortest possible jump. With near and short jumps, the operand specifies a new IP address. With far jumps, the operand specifies new IP and CS addresses.

When the 80386–80486 processors are in **FLAT** memory model, short jumps range from –128 to +127 bytes and near jumps range from –2 to +2 gigabytes.

Flags

No change

Encoding

11101011 *disp (1)*

Syntax	Examples	CPU	Clock Cycles
JMP *label*	`jmp SHORT exit`	88/86	15
		286	7+*m*
		386	7+*m*
		486	3

Encoding

11101001 *disp (2*)*

Syntax	Examples	CPU	Clock Cycles
JMP *label*	`jmp close`	88/86	15
	`jmp NEAR PTR distant`	286	7+*m*
		386	7+*m*
		486	3

Encoding

11101010 *disp (4*)*

Syntax	Examples	CPU	Clock Cycles
JMP *label*	`jmp FAR PTR close`	88/86	15
	`jmp distant`	286	11+*m*,*pm*=23+*m*†
		386	12+*m*,*pm*=27+*m*†
		486	17,*pm*=19†

Encoding

11111111 *mod*,100,*r/m* *disp (0 or 2)*

Syntax	Examples	CPU	Clock Cycles
JMP *reg16*	`jmp ax`	88/86	11
JMP *mem32*§		286	7+*m*
		386	7+*m*
		486	5
JMP *mem16*	`jmp WORD PTR [bx]`	88/86	18+*EA*
JMP *mem32*§	`jmp table[di]`	286	11+*m*
	`jmp DWORD PTR [si]`	386	10+*m*
		486	5

Encoding 11111111 *mod*,101,*r/m* *disp (4*)*

Syntax	Examples	CPU	Clock Cycles
JMP *mem32*	`jmp fpointer[si]`	88/86	24+*EA*
JMP *mem48*§	`jmp DWORD PTR [bx]`	286	15+*m*,*pm*=26+*m*
	`jmp FWORD PTR [di]`	386	12+*m*,*pm*=27+*m*
		486	13,*pm*=18

* On the 80386–80486, the displacement can be 4 bytes for near jumps or 6 bytes for far jumps.

† Timings for jumps through call or task gates are not shown, since they are normally used only in operating systems.

§ 80386–80486 only. You can use **DWORD PTR** to specify near register-indirect jumps or **FWORD PTR** to specify far register-indirect jumps.

LAHF Load Flags into AH Register

Transfers bits 0 to 7 of the flags register to AH. This includes the carry, parity, auxiliary carry, zero, and sign flags, but not the trap, interrupt, direction, or overflow flags.

Flags No change

Encoding 10011111

Syntax	Examples	CPU	Clock Cycles
LAHF	`lahf`	88/86	4
		286	2
		386	2
		486	3

LAR Load Access Rights

80286-80486 Protected Only Loads the access rights of a selector into a specified register. The source operand must be a register or memory operand containing a selector. The destination operand must be a register that will receive the access rights if the selector is valid and visible at the current privilege level. The zero flag is set if the access rights are transferred, or cleared if they are not. See Intel documentation for details on selectors, access rights, and other privileged-mode concepts.

Flags O D I T S Z A P C
±

Encoding 00001111 00000010 *mod, reg, r/m* *disp (0, 1, 2, or 4)*

Syntax	Examples	CPU	Clock Cycles
LAR *reg16,reg16*	`lar ax,bx`	88/86	—
LAR *reg32,reg32**		286	14
		386	15
		486	11
LAR *reg16,mem16*	`lar cx,selector`	88/86	—
LAR *reg32,mem32**		286	16
		386	16
		486	11

* 80386–80486 only.

LDS/LES/LFS/LGS/LSS Load Far Pointer

Reads and stores the far pointer specified by the source memory operand. The instruction moves the pointer's segment value into DS, ES, FS, GS, or SS (depending on the instruction). Then it moves the pointer's offset value into the destination operand. The **LDS** and **LES** instructions are available on all processors. The **LFS**, **LGS**, and **LSS** instructions are available only on the 80386–80486.

Flags No change

Encoding 11000101 *mod, reg, r/m* *disp (2)*

Syntax	Examples	CPU	Clock Cycles
LDS *reg,mem*	`lds si,fpointer`	88/86	16+*EA* (*88*=24+*EA*)
		286	7,*pm*=21
		386	7,*pm*=22
		486	6,*pm*=12

Encoding 11000100 *mod, reg, r/m* *disp (2)*

Syntax	Examples	CPU	Clock Cycles
LES *reg,mem*	`les di,fpointer`	88/86	16+*EA* (*88*=24+*EA*)
		286	7,*pm*=21
		386	7,*pm*=22
		486	6,*pm*=12

Encoding 00001111 10110100 *mod, reg, r/m* *disp (2 or 4)*

Syntax	Examples	CPU	Clock Cycles
LFS *reg,mem*	`lfs edi,fpointer`	88/86	—
		286	—
		386	7,*pm*=25
		486	6,*pm*=12

Encoding 00001111 10110101 *mod, reg, r/m* *disp (2 or 4)*

Syntax	Examples	CPU	Clock Cycles
LGS *reg,mem*	`lgs bx,fpointer`	88/86	—
		286	—
		386	7,*pm*=25
		486	6,*pm*=12

Encoding 00001111 10110010 *mod, reg, r/m* *disp (2 or 4)*

Syntax	Examples	CPU	Clock Cycles
LSS *reg,mem*	`lss bp,fpointer`	88/86	—
		286	—
		386	7,*pm*=22
		486	6,*pm*=12

LEA Load Effective Address

Calculates the effective address (offset) of the source memory operand and stores the result in the destination register. If the source operand is a direct memory address, the assembler encodes the instruction in the more efficient `MOV reg,immediate` form (equivalent to **MOV** *reg*, **OFFSET** *mem*).

Flags No change

Encoding 10001101 *mod, reg, r/m* *disp (2)*

Syntax	Examples	CPU	Clock Cycles
LEA *reg16,mem*	`lea bx,npointer`	88/86	2+*EA*
LEA *reg32,mem**		286	3
		386	2
		486	1†

* 80386–80486 only.

† 2 if index register used.

LEAVE High Level Procedure Exit

Terminates the stack frame of a procedure. **LEAVE** reverses the action of a previous **ENTER** instruction by restoring SP and BP to the values they had before the procedure stack frame was initialized. **LEAVE** is equivalent to `mov sp,bp`, followed by `pop bp`.

Flags No change

Encoding 11001001

Syntax	Examples	CPU	Clock Cycles
LEAVE	`leave`	88/86	—
		286	5
		386	4
		486	5

LES/LFS/LGS Load Far Pointer to Extra Segment

See **LDS**.

LGDT/LIDT/LLDT Load Descriptor Table

Loads a value from an operand into a descriptor table register. **LGDT** loads into the Global Descriptor Table, **LIDT** into the Interrupt Vector Table, and **LLDT** into the Local Descriptor Table. These instructions are available only in privileged mode. See Intel documentation for details on descriptor tables and other protected-mode concepts.

Flags No change

Encoding 00001111 00000001 *mod, 010,r/m* *disp (2)*

Syntax	Examples	CPU	Clock Cycles
LGDT *mem48*	`lgdt descriptor`	88/86	—
		286	11
		386	11
		486	11

Encoding 00001111 00000001 *mod, 011,r/m* *disp (2)*

Syntax	Examples	CPU	Clock Cycles
LIDT *mem48*	`lidt descriptor`	88/86	—
		286	12
		386	11
		486	11

Encoding 00001111 00000000 *mod, 010,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
LLDT *reg16*	`lldt ax`	88/86	—
		286	17
		386	20
		486	11
LLDT *mem16*	`lldt selector`	88/86	—
		286	19
		386	24
		486	11

LMSW Load Machine Status Word

80286-80486 Privileged Only Loads a value from a memory operand into the Machine Status Word (MSW). This instruction is available only in privileged mode. See Intel documentation for details on the MSW and other protected-mode concepts.

Flags No change

Encoding 00001111 00000001 *mod, 110,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
LMSW *reg16*	`lmsw ax`	88/86	—
		286	3
		386	10
		486	13
LMSW *mem16*	`lmsw machine`	88/86	—
		286	6
		386	13
		486	13

LOCK Lock the Bus

Locks out other processors during execution of the next instruction. This instruction is a prefix. It must precede an instruction that accesses a memory location that another processor might attempt to access at the same time. See Intel documentation for details on multiprocessor environments.

Flags

No change

Encoding

11110000

Syntax	Examples	CPU	Clock Cycles
LOCK *instruction*	`lock xchg ax,sem`	88/86	2
		286	0
		386	0
		486	1

LODS/LODSB/LODSW/LODSD Load Accumulator from String

Loads the accumulator register with an element from a string in memory. DS:SI must point to the source element, even if an operand is given. For each source element loaded, SI is adjusted according to the size of the operand and the status of the direction flag. SI is incremented if the direction flag has been cleared with **CLD** or decremented if the direction flag has been set with **STD**.

If the **LODS** form of the instruction is used, an operand must be provided to indicate the size of the data elements to be processed. A segment override can be given. If **LODSB** (bytes), **LODSW** (words), or **LODSD** (doublewords on the 80386–80486 only) is used, the instruction determines the size of the data elements to be processed and whether the element will be loaded to AL, AX, or EAX.

LODS and its variations are not used with repeat prefixes, since there is no reason to repeatedly load memory values to a register.

Flags

No change

Encoding 1010110*w*

Syntax	Examples	CPU	Clock Cycles
LODS [[*segreg*:]]*src*	`lods es:source`	88/86	12 (*W88*=16)
LODSB [[[*segreg*:]]*src*]]	`lodsw`	286	5
LODSW[[[*segreg*:]]*src*]]		386	5
LODSD [[[*segreg*:]]*src*]]		486	5

LOOP/LOOPW/LOOPD Loop

Loops repeatedly to a specified label. **LOOP** decrements CX (without changing any flags) and, if the result is not 0, transfers execution to the address specified by the operand. On the 80386–80486, **LOOP** uses the 16-bit CX in 16-bit mode and the 32-bit ECX in 32-bit mode. The default can be overridden with **LOOPW** (CX) or **LOOPD** (ECX). If CX is 0 after being decremented, execution continues at the next instruction. The operand must specify a short label (between –128 and +127 bytes from the instruction following the **LOOP** instruction).

Flags No change

Encoding 11100010 *disp (1)*

Syntax	Examples	CPU	Clock Cycles
LOOP *label*	`loop wend`	88/86	17,*noj*=5
LOOPW *label**		286	8+*m*,*noj*=4
LOOPD *label**		386	11+*m*
		486	7,*noj*=6

* 80386–80486 only.

LOOP*condition*/LOOP*condition*W/LOOP*condition*D Loop Conditionally

Loops repeatedly to a specified label if *condition* is met and if CX is not 0. On the 80386–80486, these instructions use the 16-bit CX in 16-bit mode and the 32-bit ECX in 32-bit mode. This default can be overridden with the **W** (CX) or **D** (ECX) forms of the instruction. The instruction decrements CX (without changing any flags) and tests whether the zero flag was set by a previous instruction (such as **CMP**). With **LOOPE** and **LOOPZ** (they are synonyms),

execution is transferred to the label if the zero flag is set and CX is not 0. With **LOOPNE** and **LOOPNZ** (they are synonyms), execution is transferred to the label if the zero flag is cleared and CX is not 0. Execution continues at the next instruction if the condition is not met. Before entering the loop, CX should be set to the maximum number of repetitions desired.

Flags No change

Encoding 11100001 *disp (1)*

Syntax	Examples	CPU	Clock Cycles
LOOPE *label*	`loopz again`	88/86	18,*noj*=6
LOOPEW *label**		286	8+*m*,*noj*=4
LOOPED *label**		386	11+*m*
LOOPZ *label*		486	9,*noj*=6
LOOPZW *label**			
LOOPZD *label**			

Encoding 11100000 *disp (1)*

Syntax	Examples	CPU	Clock Cycles
LOOPNE *label*	`loopnz for_next`	88/86	19,*noj*=5
LOOPNEW *label**		286	8,*noj*=4
LOOPNED *label**		386	11+*m*
LOOPNZ *label*		486	9,*noj*=6
LOOPNZW *label**			
LOOPNZD *label**			

* 80386–80486 only.

LSL Load Segment Limit

80286-80486 Protected Only Loads the segment limit of a selector into a specified register. The source operand must be a register or memory operand containing a selector. The destination operand must be a register that will receive the segment limit if the selector is valid and visible at the current privilege level. The zero flag is set if the segment limit is transferred, or cleared if it is not. See Intel documentation for details on selectors, segment limits, and other protected-mode concepts.

Flags

O	D	I	T	S	Z	A	P	C
					±			

Encoding 00001111 00000011 *mod, reg, r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
LSL *reg16,reg16*	`lsl ax,bx`	88/86	—
LSL *reg32,reg32**		286	14
		386	20,25†
		486	10
LSL *reg16,mem16*	`lsl cx,seg_lim`	88/86	—
LSL *reg32,mem32**		286	16
		386	21,26†
		486	10

* 80386–80486 only.

† The first value is for byte granular; the second is for page granular.

LSS Load Far Pointer to Stack Segment

See **LDS**.

LTR Load Task Register

80286-80486 Protected Only Loads a value from the specified operand to the current task register. **LTR** is available only in privileged mode. See Intel documentation for details on task registers and other protected-mode concepts.

Flags No change

Encoding 00001111 00000000 *mod, 011,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
LTR *reg16*	`ltr ax`	88/86	—
		286	17
		386	23
		486	20
LTR *mem16*	`ltr task`	88/86	—
		286	19
		386	27
		486	20

MOV Move Data

Moves the value in the source operand to the destination operand. If the destination operand is SS, interrupts are disabled until the next instruction is executed (except on early versions of the 8088 and 8086).

Flags No change

Encoding 100010*dw* *mod, reg, r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
MOV *reg,reg*	`mov dh,bh`	88/86	2
	`mov dx,cx`	286	2
	`mov bp,sp`	386	2
		486	1
MOV *mem,reg*	`mov array[di],bx`	88/86	9+*EA* (*W88*=13+*EA*)
	`mov count,cx`	286	3
		386	2
		486	1
MOV *reg,mem*	`mov bx,pointer`	88/86	8+*EA* (*W88*=12+*EA*)
	`mov dx,matrix[bx+di]`	286	5
		386	4
		486	1

Encoding 1100011*w* *mod*, 000,*r/m* *disp (0, 1, or 2)* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
MOV *mem,immed*	`mov [bx],15`	88/86	10+*EA* (*W88*=14+*EA*)
	`mov color,7`	286	3
		386	2
		486	1

Encoding 1011*w reg* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
MOV *reg,immed*	`mov cx,256`	88/86	4
	`mov dx,OFFSET string`	286	2
		386	2
		486	1

Encoding 101000*aw* *disp (2)*

Syntax	Examples	CPU	Clock Cycles
MOV *mem,accum*	`mov total,ax`	88/86	10 (*W88*=14)
		286	3
		386	2
		486	1
MOV *accum,mem*	`mov al,string`	88/86	10 (*W88*=14)
		286	5
		386	4
		486	1

Encoding 100011*d*0 *mod,sreg, r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
MOV *segreg,reg16*	`mov ds,ax`	88/86	2
		286	2,*pm*=17
		386	2,*pm*=18
		486	3,*pm*=9
MOV *segreg,mem16*	`mov es,psp`	88/86	8+*EA* (*88*=12+*EA*)
		286	5,*pm*=19
		386	5,*pm*=19
		486	3,*pm*=9
MOV *reg16,segreg*	`mov ax,ds`	88/86	2
		286	2
		386	2
		486	3
MOV *mem16,segreg*	`mov stack_save,ss`	88/86	9+*EA* (*88*=13+*EA*)
		286	3
		386	2
		486	3

MOV Move to/from Special Registers

80386–80486 Only Moves a value from a special register to or from a 32-bit general-purpose register. The special registers include the control registers CR0, CR2, and CR3; the debug registers DR0, DR1, DR2, DR3, DR6, and DR7; and the test registers TR6 and TR7. On the 80486, the test registers TR3, TR4, and TR5 are also available. See Intel documentation for details on special registers.

Flags

O	D	I	T	S	Z	A	P	C
?				?	?	?	?	?

Encoding 00001111 001000*d*0 11, *reg**, *r/m*

Syntax	Examples	CPU	Clock Cycles
MOV *reg32, controlreg*	`mov eax,cr2`	88/86	—
		286	—
		386	6
		486	4
MOV *controlreg,reg32*	`mov cr0,ebx`	88/86	—
		286	—
		386	CR0=10,CR2=4,CR3=5
		486	4,CR0=16

Encoding 00001111 001000*d*1 11, *reg**, *r/m*

Syntax	Examples	CPU	Clock Cycles
MOV *reg32,debugreg*	`mov edx,dr3`	88/86	—
		286	—
		386	DR0–3=22,DR6–7=14
		486	10
MOV *debugreg,reg32*	`mov dr0,ecx`	88/86	—
		286	—
		386	DR0–3=22,DR6–7=16
		486	11

Encoding 00001111 001001*d*0 11,*reg**, *r/m*

Syntax	Examples	CPU	Clock Cycles
MOV *reg32,testreg*	`mov edx,tr6`	88/86	—
		286	—
		386	12
		486	4,TR3=3
MOV *testreg, reg32*	`mov tr7,eax`	88/86	—
		286	—
		386	12
		486	4,TR3=6

* The *reg* field contains the register number of the special register (for example, 000 for CR0, 011 for DR7, or 111 for TR7).

MOVS/MOVSB/MOVSW/MOVSD Move String Data

Moves a string from one area of memory to another. DS:SI must point to the source string and ES:DI to the destination address, even if operands are given. For each element moved, DI and SI are adjusted according to the size of the operands and the status of the direction flag. They are increased if the direction flag has been cleared with **CLD**, or decreased if the direction flag has been set with **STD**.

If the **MOVS** form of the instruction is used, operands must be provided to indicate the size of the data elements to be processed. A segment override can be given for the source operand (but not for the destination). If **MOVSB** (bytes), **MOVSW** (words), or **MOVSD** (doublewords on the 80386–80486 only) is used, the instruction determines the size of the data elements to be processed.

MOVS and its variations are normally used with the **REP** prefix.

Flags No change

Encoding 1010010*w*

Syntax	Examples	CPU	Clock Cycles
MOVS [[**ES:**]*dest*,[[*segreg*:]]*src*	`rep movsb`	88/86	18 (*W88*=26)
MOVSB [[[**ES:**]*dest*,[[*segreg*:]]*src*]]	`movs dest,es:source`	286	5
MOVSW [[[**ES:**]*dest*,[[*segreg*:]]*src*]]		386	7
MOVSD [[[**ES:**]*dest*,[[*segreg*:]]*src*]]		486	7

MOVSX Move with Sign-Extend

80386–80486 Only Moves and sign-extends the value of the source operand to the destination register. **MOVSX** is used to copy a signed 8-bit or 16-bit source operand to a larger 16-bit or 32-bit destination register.

Flags No change

Encoding 00001111 1011111*w* *mod, reg, r/m* *disp (0, 1, 2, or 4)*

Syntax	Examples	CPU	Clock Cycles
MOVSX *reg,reg*	`movsx eax,bx`	88/86	—
	`movsx ecx,bl`	286	—
	`movsx bx,al`	386	3
		486	3

Syntax	Examples		CPU	Clock Cycles
MOVSX *reg,mem*	`movsx`	`cx,bsign`	88/86	—
	`movsx`	`edx,wsign`	286	—
	`movsx`	`eax,bsign`	386	6
			486	3

MOVZX Move with Zero-Extend

80386–80486 Only Moves and zero-extends the value of the source operand to the destination register. **MOVZX** is used to copy an unsigned 8-bit or 16-bit source operand to a larger 16-bit or 32-bit destination register.

Flags No change

Encoding 00001111 1011011*w* *mod, reg, r/m* *disp (0, 1, 2, or 4)*

Syntax	Examples		CPU	Clock Cycles
MOVZX *reg,reg*	`movzx`	`eax,bx`	88/86	—
	`movzx`	`ecx,bl`	286	—
	`movzx`	`bx,al`	386	3
			486	3
MOVZX *reg,mem*	`movzx`	`cx,bunsign`	88/86	—
	`movzx`	`edx,wunsign`	286	—
	`movzx`	`eax,bunsign`	386	6
			486	3

MUL Unsigned Multiply

Multiplies an implied destination operand by a specified source operand. Both operands are treated as unsigned numbers. If a single 16-bit operand is given, the implied destination is AX and the product goes into the DX:AX register pair. If a single 8-bit operand is given, the implied destination is AL and the product goes into AX. On the 80386–80486, if the operand is EAX, the product goes into the EDX:EAX register pair. The carry and overflow flags are set if DX is not 0 for 16-bit operands or if AH is not 0 for 8-bit operands.

Flags

O	D	I	T	S	Z	A	P	C
±				?	?	?	?	±

Encoding 1111011*w* *mod*, 100, *r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
MUL *reg*	`mul bx`	88/86	*b*=70–77,*w*=118–133
	`mul dl`	286	*b*=13,*w*=21
		386	*b*=9–14,*w*=9–22,*d*=9–38*
		486	*b*=13–18,*w*=13–26,*d*=13–42
MUL *mem*	`mul factor`	88/86	(*b*=76–83,*w*=124–139)+*EA*†
	`mul WORD PTR [bx]`	286	*b*=16,*w*=24
		386	*b*=12–17,*w*=12–25,*d*=12–41*
		486	*b*=13–18,*w*=13–26,*d*=13–42

* The 80386–80486 processors have an early-out multiplication algorithm. Therefore, multiplying an 8-bit or 16-bit value in EAX takes the same time as multiplying the value in AL or AX.

† Word memory operands on the 8088 take (128–143)+*EA* clocks.

NEG Two's Complement Negation

Replaces the operand with its two's complement. **NEG** does this by subtracting the operand from 0. If the operand is 0, the carry flag is cleared. Otherwise, the carry flag is set. If the operand contains the maximum possible negative value (–128 for 8-bit operands or –32,768 for 16-bit operands), the value does not change, but the overflow and carry flags are set.

Flags

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	±

Encoding 1111011*w* *mod*, 011, *r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
NEG *reg*	`neg ax`	88/86	3
		286	2
		386	2
		486	1
NEG *mem*	`neg balance`	88/86	16+*EA* (*W88*=24+*EA*)
		286	7
		386	6
		486	3

NOP No Operation

Performs no operation. **NOP** can be used for timing delays or alignment.

Flags No change

Encoding 10010000*

Syntax	Examples	CPU	Clock Cycles
NOP	`nop`	88/86	3
		286	3
		386	3
		486	3

* The encoding is the same as **XCHG AX,AX**.

NOT One's Complement Negation

Toggles each bit of the operand by clearing set bits and setting cleared bits.

Flags No change

Encoding 1111011*w* *mod*, 010, *r/m* *disp (0,1,or2)*

Syntax	Examples	CPU	Clock Cycles
NOT *reg*	`not ax`	88/86	3
		286	2
		386	2
		486	1
NOT *mem*	`not masker`	88/86	16+*EA* (*W88*=24+*EA*)
		286	7
		386	6
		486	3

OR Inclusive OR

Performs a bitwise OR operation on the source and destination operands and stores the result to the destination operand. For each bit position in the operands, if either or both bits are set, the corresponding bit of the result is set. Otherwise, the corresponding bit of the result is cleared.

Flags

O	D	I	T	S	Z	A	P	C
0				±	±	?	±	0

Encoding 000010*dw* *mod*, *reg*, *r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
OR *reg,reg*	`or ax,dx`	88/86	3
		286	2
		386	2
		486	1
OR *mem,reg*	`or bits,dx`	88/86	16+*EA* (*W88*=24+*EA*)
	`or [bp+6],cx`	286	7
		386	7
		486	3
OR *reg,mem*	`or bx,masker`	88/86	9+*EA* (*W88*=13+*EA*)
	`or dx,color[di]`	286	7
		386	6
		486	2

Encoding 100000*sw* *mod*,001, *r/m* *disp (0, 1, or 2)* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
OR *reg,immed*	`or dx,110110b`	88/86	4
		286	3
		386	2
		486	1
OR *mem,immed*	`or flag_rec,8`	88/86	(*b*=17,*w*=25)+*EA*
		286	7
		386	7
		486	3

Encoding 0000110*w* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
OR *accum,immed*	`or ax,40h`	88/86	4
		286	3
		386	2
		486	1

OUT Output to Port

Transfers a byte or word (or a doubleword on the 80386–80486) to a port from the accumulator register. The port address is specified by the destination operand, which can be DX or an 8-bit constant. In protected mode, a general-protection fault occurs if **OUT** is used when the current privilege level is greater than the value of the IOPL flag.

Flags No change

Encoding 1110011*w* *data (1)*

Syntax	Examples	CPU	Clock Cycles
OUT *immed8,accum*	`out 60h,al`	88/86	10 (*88*=14)
		286	3
		386	10,*pm*=4,24*
		486	16,*pm*=11,31*

Encoding 1110111*w*

Syntax	Examples	CPU	Clock Cycles
OUT DX,*accum*	`out dx,ax`	88/86	8 (*88*=12)
	`out dx,al`	286	3
		386	11,*pm*=5,25*
		486	16,*pm*=10,30*

* First protected-mode timing: CPL < IOPL. Second timing: CPL > IOPL.

OUTS/OUTSB/OUTSW/OUTSD Output String to Port

80186–80486 Only Sends a string to a port. The string is considered the source and must be pointed to by DS:SI (even if an operand is given). The output port is specified in DX. For each element sent, SI is adjusted according to the size of the operand and the status of the direction flag. SI is increased if the direction flag has been cleared with **CLD**, or decreased if the direction flag has been set with **STD**.

If the **OUTS** form of the instruction is used, an operand must be provided to indicate the size of data elements to be sent. A segment override can be given. If **OUTSB** (bytes), **OUTSW** (words), or **OUTSD** (doublewords on the 80386–80486 only) is used, the instruction determines the size of the data elements to be sent.

OUTS and its variations are normally used with the **REP** prefix. Before the instruction is executed, CX should contain the number of elements to send. In protected mode, a general-protection fault occurs if **OUTS** is used when the current privilege level is greater than the value of the IOPL flag.

Flags No change

Encoding 0110111*w*

Syntax	Examples	CPU	Clock Cycles
OUTS DX, ⟦*segreg*:⟧ *src*	`rep outs`	88/86	—
OUTSB ⟦**DX,** ⟦*segreg*:⟧ *src*⟧	`dx,buffer`	286	5
OUTSW ⟦**DX,** ⟦*segreg*:⟧ *src*⟧	`outsb`	386	14,*pm*=8,28*
OUTSD ⟦**DX,** ⟦*segreg*:⟧ *src*⟧	`rep outsw`	486	17,*pm*=10,32*

* First protected-mode timing: CPL < IOPL. Second timing: CPL > IOPL.

POP Pop

Pops the top of the stack into the destination operand. The value at SS:SP is copied to the destination operand and SP is increased by 2. The destination operand can be a memory location, a general-purpose 16-bit register, or any segment register except CS. Use **RET** to pop CS. On the 80386–80486, 32-bit values can be popped by giving a 32-bit operand. ESP is increased by 4 for 32-bit pops.

Flags No change

Encoding 01011 *reg*

Syntax	Examples	CPU	Clock Cycles
POP *reg16*	`pop cx`	88/86	8 (*88*=12)
POP *reg32**		286	5
		386	4
		486	1

Encoding 10001111 *mod*,000,*r/m* *disp (2)*

Syntax	Examples	CPU	Clock Cycles
POP *mem16*	`pop param`	88/86	17+*EA* (*88*=25+*EA*)
POP *mem32**		286	5
		386	5
		486	6

Encoding 000,*sreg*,111

Syntax	Examples	CPU	Clock Cycles
POP *segreg*	`pop es`	88/86	8 (*88*=12)
	`pop ds`	286	5,*pm*=20
	`pop ss`	386	7,*pm*=21
		486	3,*pm*=9

Encoding 00001111 10,*sreg*,001

Syntax	Examples	CPU	Clock Cycles
POP *segreg**	`pop fs`	88/86	—
	`pop gs`	286	—
		386	7,*pm*=21
		486	3,*pm*=9

* 80386–80486 only.

POPA/POPAD Pop All

80186-80486 Only Pops the top 16 bytes on the stack into the eight general-purpose registers. The registers are popped in the following order: DI, SI, BP, SP, BX, DX, CX, AX. The value for the SP register is actually discarded rather than copied to SP. **POPA** always pops into 16-bit registers. On the 80386–80486, use **POPAD** to pop into 32-bit registers.

Flags No change

Encoding 01100001

Syntax	Examples	CPU	Clock Cycles
POPA	`popa`	88/86	—
POPAD*		286	19
		386	24
		486	9

* 80386–80486 only.

POPF/POPFD Pop Flags

Pops the value on the top of the stack into the flags register. **POPF** always pops into the 16-bit flags register. On the 80386–80486, use **POPFD** to pop into the 32-bit flags register.

Flags

O	D	I	T	S	Z	A	P	C
±	±	±	±	±	±	±	±	±

Encoding

10011101

Syntax	Examples	CPU	Clock Cycles
POPF	`popf`	88/86	8 (*88*=12)
POPFD*		286	5
		386	5
		486	9,*pm*=6

* 80386–80486 only.

PUSH/PUSHW/PUSHD Push

Pushes the source operand onto the stack. SP is decreased by 2 and the source value is copied to SS:SP. The operand can be a memory location, a general-purpose 16-bit register, or a segment register. On the 80186–80486 processors, the operand can also be a constant. On the 80386–80486, 32-bit values can be pushed by specifying a 32-bit operand. ESP is decreased by 4 for 32-bit pushes. On the 8088 and 8086, **PUSH SP** saves the value of SP after the push. On the 80186–80486 processors, **PUSH SP** saves the value of SP before the push. The **PUSHW** and **PUSHD** instructions push a word (2 bytes) and a doubleword (4 bytes), respectively.

Flags

No change

Encoding

01010 *reg*

Syntax	Examples	CPU	Clock Cycles
PUSH *reg16*	`push dx`	88/86	11 (*88*=15)
PUSH *reg32**		286	3
PUSHW *reg16*		386	2
PUSHD *reg32**		486	1

Encoding 11111111 *mod*, 110,*r/m* *disp (2)*

Syntax	Examples	CPU	Clock Cycles
PUSH *mem16*	`push [di]`	88/86	16+*EA* (*88*=24+*EA*)
PUSH *mem32**	`push fcount`	286	5
		386	5
		486	4

Encoding 00,*sreg*,110

Syntax	Examples	CPU	Clock Cycles
PUSH *segreg*	`push es`	88/86	10 (*88*=14)
PUSHW *segreg*	`push ss`	286	3
PUSHD *segreg**	`push cs`	386	2
		486	3

Encoding 00001111 10,*sreg*,000

Syntax	Examples	CPU	Clock Cycles
PUSH *segreg*	`push fs`	88/86	—
PUSHW *segreg*	`push gs`	286	—
PUSHD *segreg**		386	2
		486	3

Encoding 011010*s*0 *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
PUSH *immed*	`push 'a'`	88/86	—
PUSHW *immed*	`push 15000`	286	3
PUSHD *immed**		386	2
		486	1

* 80386–80486 only.

PUSHA/PUSHAD Push All

80186–80486 Only Pushes the eight general-purpose registers onto the stack. The registers are pushed in the following order: AX, CX, DX, BX, SP, BP, SI, DI. The value pushed for SP is the value before the instruction. **PUSHA** always pushes 16-bit registers. On the 80386–80486, use **PUSHAD** to push 32-bit registers.

Flags No change

Encoding 01100000

Syntax	Examples	CPU	Clock Cycles
PUSHA	`pusha`	88/86	—
PUSHAD*		286	17
		386	18
		486	11

* 80386–80486 only.

PUSHF/PUSHFD Push Flags

Pushes the flags register onto the stack. **PUSHF** always pushes the 16-bit flags register. On the 80386–80486, use **PUSHFD** to push the 32-bit flags register.

Flags No change

Encoding 10011100

Syntax	Examples	CPU	Clock Cycles
PUSHF	`pushf`	88/86	10(*88*=14)
PUSHFD*		286	3
		386	4
		486	4,*pm*=3

* 80386–80486 only.

RCL/RCR/ROL/ROR Rotate

Rotates the bits in the destination operand the number of times specified in the source operand. **RCL** and **ROL** rotate the bits left; **RCR** and **ROR** rotate right.

ROL and **ROR** rotate the number of bits in the operand. For each rotation, the leftmost or rightmost bit is copied to the carry flag as well as rotated. **RCL** and **RCR** rotate through the carry flag. The carry flag becomes an extension of the operand so that a 9-bit rotation is done for 8-bit operands, or a 17-bit rotation for 16-bit operands.

On the 8088 and 8086, the source operand can be either CL or 1. On the 80186–80486, the source operand can be CL or an 8-bit constant. On the 80186–80486, rotate counts larger than 31 are masked off, but on the 8088 and 8086, larger rotate counts are performed despite the inefficiency involved. The

overflow flag is modified only by single-bit variations of the instruction; for multiple-bit variations, the overflow flag is undefined.

Flags

O	D	I	T	S	Z	A	P	C
±								±

Encoding 1101000*w* *mod, TTT*,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
ROL *reg*,**1**	`ror ax,1`	88/86	2
ROR *reg*,**1**	`rol dl,1`	286	2
		386	3
		486	3
RCL *reg*,**1**	`rcl dx,1`	88/86	2
RCR *reg*,**1**	`rcr bl,1`	286	2
		386	9
		486	3
ROL *mem*,**1**	`ror bits,1`	88/86	15+*EA* (*W88*=23+*EA*)
ROR *mem*,**1**	`rol WORD PTR [bx],1`	286	7
		386	7
		486	4
RCL *mem*,**1**	`rcl WORD PTR [si],1`	88/86	15+*EA* (*W88*=23+*EA*
RCR *mem*,**1**	`rcr WORD PTR m32[0],1`	286	7
		386	10
		486	4

Encoding 1101001*w* *mod, TTT*,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
ROL *reg*,**CL**	`ror ax,cl`	88/86	8+4*n*
ROR *reg*,**CL**	`rol dx,cl`	286	5+*n*
		386	3
		486	3
RCL *reg*,**CL**	`rcl dx,cl`	88/86	8+4*n*
RCR *reg*,**CL**	`rcr bl,cl`	286	5+*n*
		386	9
		486	8–30
ROL *mem*,**CL**	`ror color,cl`	88/86	20+*EA*+4*n*
ROR *mem*,**CL**			(*W88*=28+*EA*+4*n*)
	`rol WORD PTR [bp+6],cl`	286	8+*n*
		386	7
		486	4

Syntax	Examples	CPU	Clock Cycles
RCL *mem*,**CL** **RCR** *mem*,**CL**	`rcr WORD PTR [bx+di],cl`	88/86	20+*EA*+4*n* (*W88*=28+*EA*+4*n*)
	`rcl masker`	286	8+*n*
		386	10
		486	9–31

Encoding 1100000*w* *mod,TTT*,r/m* *disp (0, 1, or 2)* *data (1)*

Syntax	Examples	CPU	Clock Cycles
ROL *reg,immed8* **ROR** *reg,immed8*	`rol ax,13` `ror bl,3 286`	88/86	—
		286	5+*n*
		386	3
		486	2
RCL *reg,immed8* **RCR** *reg,immed8*	`rcl bx,5` `rcr si,9`	88/86	—
		286	5+*n*
		386	9
		486	8–30
ROL *mem,immed8* **ROR** *mem,immed8*	`rol BYTE PTR [bx],10` `ror bits,6`	88/86	—
		286	8+*n*
		386	7
		486	4
RCL *mem,immed8* **RCR** *mem,immed8*	`rcl WORD PTR [bp+8],` `rcr masker,3`	88/86	—
		286	8+*n*
		386	10
		486	9–31

* *TTT* represents one of the following bit codes: 000 for **ROL**, 001 for **ROR**, 010 for **RCL**, or 011 for **RCR**.

REP Repeat String

Repeats a string instruction the number of times indicated by CX. First, CX is compared to 0; if it equals 0, execution proceeds to the next instruction. Otherwise, CX is decremented, the string instruction is performed, and the loop continues. **REP** is used with **MOVS** and **STOS**. **REP** also can be used with **INS** and **OUTS** on the 80186–80486 processors. On all processors except the 80386–80486, combining a repeat prefix with a segment override can cause errors if an interrupt occurs.

Flags No change

Encoding 11110011 1010010*w*

Syntax	Examples	CPU	Clock Cycles
REP MOVS *dest*,*src*	`rep movs source,dest`	88/86	9+17*n* (*W88*=9+25*n*)
REP MOVSB ⟦*dest*,*src*⟧	`rep movsw`	286	5+4*n*
REP MOVSW ⟦*dest*,*src*⟧		386	7+4*n*
REP MOVSD ⟦*dest*,*src*⟧*		486	12+3*n*#

Encoding 11110011 1010101*w*

Syntax	Examples	CPU	Clock Cycles
REP STOS *dest*	`rep stosb`	88/86	9+10*n* (*W88*=9+14*n*)
REP STOSB ⟦*dest*⟧	`rep stos dest`	286	4+3*n*
REP STOSW ⟦*dest*⟧		386	5+5*n*
REP STOSD ⟦*dest*⟧*		486	7+4*n*†

Encoding 11110011 1010101*w*

Syntax	Examples	CPU	Clock Cycles
REP LODS *dest*	`rep lodsb`	88/86	—
REP LODSB ⟦*dest*⟧	`rep lods dest`	286	—
REP LODSW ⟦*dest*⟧		386	—
REP LODSD ⟦*dest*⟧*		486	7+4*n*†

Encoding 11110011 0110110*w*

Syntax	Examples	CPU	Clock Cycles
REP INS *dest*,**DX**	`rep insb`	88/86	—
REP INSB ⟦*dest*,**DX**⟧	`rep ins dest,dx`	286	5+4*n*
REP INSW ⟦*dest*,**DX**⟧		386	13+6*n*,*pm*=(7,27)+6*n*§
REP INSD ⟦*dest*,**DX**⟧*			16+8*n*,*pm*=(10,30)+8*n*
		486	§

Encoding 11110011 0110111*w*

Syntax	Examples	CPU	Clock Cycles
REP OUTS DX,*src*	`rep outs dx,source`	88/86	—
REP OUTSB ⟦*src*⟧	`rep outsw`	286	5+4*n*
REP OUTSW ⟦*src*⟧		386	12+5*n*,*pm*=(6,26)+5*n*§
REP OUTSD ⟦*src*⟧*		486	17+5*n*,*pm*=(11,31)+5*n*§

* 80386–80486 only.

5 if *n* = 0, 13 if *n* = 1.

† 5 if *n* = 0.

§ First protected-mode timing: CPL ≤ IOPL. Second timing: CPL > IOPL.

REP*condition* Repeat String Conditionally

Repeats a string instruction as long as *condition* is true and the maximum count has not been reached. **REPE** and **REPZ** (they are synonyms) repeat while the zero flag is set. **REPNE** and **REPNZ** (they are synonyms) repeat while the zero flag is cleared. The conditional-repeat prefixes should only be used with **SCAS** and **CMPS**, since these are the only string instructions that modify the zero flag. Before executing the instruction, CX should be set to the maximum allowable number of repetitions. First, CX is compared to 0; if it equals 0, execution proceeds to the next instruction. Otherwise, CX is decremented, the string instruction is performed, and the loop continues. On all processors except the 80386–80486, combining a repeat prefix with a segment override may cause errors if an interrupt occurs during a string operation.

Flags

O	D	I	T	S	Z	A	P	C
					±			

Encoding 11110011 1010011*w*

Syntax	Examples	CPU	Clock Cycles
REPE CMPS *src,dest*	`repz cmpsb`	88/86	9+22*n* (*W88*=9+30*n*)
REPE CMPSB ⟦*src,dest*⟧	`repe cmps`	286	5+9*n*
REPE CMPSW ⟦*src,dest*⟧	`src,dest`	386	5+9*n*
REPE CMPSD ⟦*src,dest*⟧*		486	7+7*n*#

Encoding 11110011 1010111*w*

Syntax	Examples	CPU	Clock Cycles
REPE SCAS *dest*	`repe scas dest`	88/86	9+15*n* (*W88*=9+19*n*)
REPE SCASB ⟦*dest*⟧	`repz scasw`	286	5+8*n*
REPE SCASW ⟦*dest*⟧		386	5+8*n*
REPE SCASD ⟦*dest*⟧*		486	7+5*n*#

Encoding 11110010 1010011*w*

Syntax	Examples	CPU	Clock Cycles
REPNE CMPS *src,dest*	`repne cmpsw`	88/86	9+22*n* (*W88*=9+30*n*)
REPNE CMPSB ⟦*src,dest*⟧	`repnz cmps`	286	5+9*n*
REPNE CMPSW ⟦*src,dest*⟧	`src,dest`	386	5+9*n*
REPNE CMPSD ⟦*src,dest*⟧*		486	7+7*n*#

Encoding 11110010 1010111*w*

Syntax	Examples	CPU	Clock Cycles
REPNE SCAS *des*	`repne scas dest`	88/86	9+15*n* (*W88*=9+19*n*)
REPNE SCASB ⟦*dest*⟧	`repnz scasb`	286	5+8*n*
REPNE SCASW ⟦*dest*⟧		386	5+8*n*
REPNE SCASD ⟦*dest*⟧*		486	7+5*n**

* 80386–80486 only.

5 if n=0.

RET/RETN/RETF Return from Procedure

Returns from a procedure by transferring control to an address popped from the top of the stack. A constant operand can be given indicating the number of additional bytes to release. The constant is normally used to adjust the stack for arguments pushed before the procedure was called. The size of a return (near or far) is the size of the procedure in which the **RET** is defined with the **PROC** directive. **RETN** can be used to specify a near return; **RETF** can specify a far return. A near return pops a word into IP. A far return pops a word into IP and then pops a word into CS. After the return, the number of bytes given in the operand (if any) is added to SP.

Flags No change

Encoding 11000011

Syntax	Examples	CPU	Clock Cycles
RET	`ret`	88/86	16 (*88*=20)
RETN	`retn`	286	11+*m*
		386	10+*m*
		486	5

Encoding 11000010 *data (2)*

Syntax	Examples	CPU	Clock Cycles
RET *immed16*	`ret 2`	88/86	20 (*88*=24)
RETN *immed16*	`retn 8`	286	11+*m*
		386	10+*m*
		486	5

Encoding 11001011

Syntax	Examples	CPU	Clock Cycles
RET	`ret`	88/86	26 (*88*=34)
RETF	`retf`	286	15+*m*,*pm*=25+*m*,55*
		386	18+*m*,*pm*=32+*m*,62*
		486	13,*pm*=18,33*

Encoding 11001010 *data (2)*

Syntax	Examples	CPU	Clock Cycles
RET *immed16*	`ret 8`	88/86	25 (*88*=33)
RETF *immed16*	`retf 32`	286	15+*m*,*pm*=25+*m*,55*
		386	18+*m*,*pm*=32+*m*,68*
		486	14,*pm*=17,33*

* The first protected-mode timing is for a return to the same privilege level; the second is for a return to a lesser privilege level.

ROL/ROR Rotate

See **RCL**/**RCR**.

SAHF Store AH into Flags

Transfers AH into bits 0 to 7 of the flags register. This includes the carry, parity, auxiliary carry, zero, and sign flags, but not the trap, interrupt, direction, or overflow flags.

Flags

O	D	I	T	S	Z	A	P	C
				±	±	±	±	±

Encoding 10011110

Syntax	Examples	CPU	Clock Cycles
SAHF	`sahf`	88/86	4
		286	2
		386	3
		486	2

SAL/SAR Shift

See **SHL/SHR/SAL/SAR**.

SBB Subtract with Borrow

Adds the carry flag to the second operand, then subtracts that value from the first operand. The result is assigned to the first operand. **SBB** is used to subtract the least significant portions of numbers that must be processed in multiple registers.

Flags

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	±

Encoding 000110*dw* *mod, reg, r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
SBB *reg,reg*	`sbb dx,cx`	88/86	3
		286	2
		386	2
		486	1
SBB *mem,reg*	`sbb WORD PTR m32[2],dx`	88/86	16+*EA* (*W88*=24+*EA*)
		286	7
		386	6
		486	3
SBB *reg,mem*	`sbb dx,WORD PTR m32[2]`	88/86	9+*EA* (*W88*=13+*EA*)
		286	7
		386	7
		486	2

Encoding 100000*sw* *mod,011, r/m* *disp (0, 1, or 2)* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
SBB *reg,immed*	`sbb dx,45`	88/86	4
		286	3
		386	2
		486	1
SBB *mem,immed*	`sbb WORD PTR m32[2],40`	88/86	17+*EA* (*W88*=25+*EA*)
		286	7
		386	7
		486	3

Encoding 0001110*w* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
SBB *accum,immed*	`sbb ax,320 88/86`	4	
		86	3
		386	2
		486	1

SCAS/SCASB/SCASW/SCASD Scan String Flags

Scans a string to find a value specified in the accumulator register. The string to be scanned is considered the destination. ES:DI must point to that string, even if an operand is specified. For each element, the destination element is subtracted from the accumulator value and the flags are updated to reflect the result (although the result is not stored). DI is adjusted according to the size of the operands and the status of the direction flag. DI is increased if the direction flag has been cleared with **CLD**, or decreased if the direction flag has been set with **STD**.

If the **SCAS** form of the instruction is used, an operand must be provided to indicate the size of the data elements to be processed. No segment override is allowed. If **SCASB** (bytes), **SCASW** (words), or **SCASD** (doublewords on the 80386–80486 only) is used, the instruction determines the size of the data elements to be processed and whether the element scanned for is in AL, AX, or EAX.

SCAS and its variations are normally used with repeat prefixes. **REPNE** (or **REPNZ**) is used to find the first element in a string that matches the value in the accumulator register. **REPE** (or **REPZ**) is used to find the first mismatch. Before the scan, CX should contain the maximum number of elements to scan. After a **REPNE SCAS**, the zero flag is clear if the string does not contain the accumulator value. After a **REPE SCAS**, the zero flag is set if the string contains nothing but the accumulator value.

When the instruction finishes, ES:DI points to the element that follows (if the direction flag is clear) or precedes (if the direction flag is set) the match or mismatch. If CX decrements to 0, ES:DI points to the element that follows or precedes the last comparison. The zero flag is set or clear according to the result of the last comparison, not according to the value of CX.

Flags

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	±

Encoding 1010111*w*

Syntax	Examples		CPU	Clock Cycles
SCAS ⟦**ES:**⟧ *dest*	`repne`	`scasw`	88/86	15 (*W88*=19)
SCASB ⟦⟦**ES:**⟧ *dest*⟧	`repe`	`scasb`	286	7
SCASW ⟦⟦**ES:**⟧ *dest*⟧	`scas`	`es:destin`	386	7
SCASD ⟦⟦**ES:**⟧ *dest*⟧*			486	6

* 80386–80486 only

SET*condition* Set Conditionally

80386–80486 Only Sets the byte specified in the operand to 1 if *condition* is true or to 0 if *condition* is false. The condition is tested by checking the flags shown in the table on the following page. The instruction is used to set Boolean flags conditionally.

Flags No change

Encoding 00001111 1001*cond* *mod*,000,*r/m*

Syntax	Examples		CPU	Clock Cycles
SET*condition reg8*	`setc`	`dh`	88/86	—
	`setz`	`al`	286	—
	`setae`	`bl`	386	4
			486	true=4, false=3
SET*condition mem8*	`seto`	`BTYE PTR [ebx]`	88/86	—
	`setle`	`flag`	286	—
	`sete`	`Booleans[di]`	386	5
			486	true=3, false=4

Set Conditions

Opcode	Mnemonic	Flags Checked	Description
10010010	**SETB/SETNAE**	CF=1	Set if below/not above or equal (unsigned comparisons)
10010011	**SETAE/SETNB**	CF=0	Set if above or equal/not below (unsigned comparisons)
10010110	**SETBE/SETNA**	CF=1 or ZF=1	Set if below or equal/not above (unsigned comparisons)
10010111	**SETA/SETNBE**	CF=0 and ZF=0	Set if above/not below or equal (unsigned comparisons)
10010100	**SETE/SETZ**	ZF=1	Set if equal/zero
10010101	**SETNE/SETNZ**	ZF=0	Set if not equal/not zero
Opcode	**Mnemonic**	**Flags Checked**	**Description**

10011100	**SETL/SETNGE**	SF_OF	Set if less/not greater or equal (signed comparisons)
10011101	**SETGE/SETNL**	SF=OF	Set if greater or equal/not less (signed comparisons)
10011110	**SETLE/SETNG**	ZF=1 or SF_OF	Set if less or equal/not greater or equal (signed comparisons)
10011111	**SETG/SETNLE**	ZF=0 and SF=OF	Set if greater/not less or equal (signed comparisons)
10011000	**SETS**	SF=1	Set if sign
10011001	**SETNS**	SF=0	Set if not sign
10010010	**SETC**	F=1	Set if carry
10010011	**SETNC**	CF=0	Set if not carry
10010000	**SETO**	OF=1	Set if overflow
10010001	**SETNO**	OF=0	Set if not overflow
10011010	**SETP/SETPE**	PF=1	Set if parity/parity even
10011011	**SETNP/SETPO**	PF=0	Set if no parity/parity odd

SGDT/SIDT/SLDT Store Descriptor Table

80286-80486 Only Stores a descriptor table register into a specified operand. **SGDT** stores the Global Descriptor Table; **SIDT**, the Interrupt Vector Table; and **SLDT**, the Local Descriptor Table. These instructions are generally useful only in privileged mode. See Intel documentation for details on descriptor tables and other protected-mode concepts.

Flags No change

Encoding 00001111 00000001 *mod*,000,*r/m* *disp (2)*

Syntax	Examples	CPU	Clock Cycles
SGDT *mem48*	`sgdt descriptor`	88/86	—
		286	11
		386	9
		486	10

Encoding 00001111 00000001 *mod*,001,*r/m* *disp (2)*

Syntax	Examples	CPU	Clock Cycles
SIDT *mem48*	`sidt descriptor`	88/86	—
		286	12
		386	9
		486	10

Encoding 00001111 00000000 *mod*, 000,*r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
SLDT *reg16*	`sldt ax`	88/86	—
		286	2
		386	2
		486	2
SLDT *mem16*	`sldt selector`	88/86	—
		286	3
		386	2
		486	3

SHL/SHR/SAL/SAR Shift

Shifts the bits in the destination operand the number of times specified by the source operand. **SAL** and **SHL** shift the bits left; **SAR** and **SHR** shift right.

With **SHL**, **SAL**, and **SHR**, the bit shifted off the end of the operand is copied into the carry flag, and the leftmost or rightmost bit opened by the shift is set to 0. With **SAR**, the bit shifted off the end of the operand is copied into the carry flag, and the leftmost bit opened by the shift retains its previous value (thus preserving the sign of the operand). **SAL** and **SHL** are synonyms.

On the 8088 and 8086, the source operand can be either CL or 1. On the 80186–80486 processors, the source operand can be CL or an 8-bit constant. On the 80186–80486 processors, shift counts larger than 31 are masked off, but on the 8088 and 8086, larger shift counts are performed despite the inefficiency. Only single-bit variations of the instruction modify the overflow flag; for multiple-bit variations, the overflow flag is undefined.

Flags

O	D	I	T	S	Z	A	P	C
±				±	±	?	±	±

Encoding 1101000*w* mod,*TTT**,r/m *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
SAR *reg*,**1**	`sar di,1`	88/86	2
	`sar cl,1`	286	2
		386	3
		486	3
SAL *reg*,**1**	`shr dh,1`	88/86	2
SHL *reg*,**1**	`shl si,1`	286	2
SHR *reg*,**1**	`sal bx,1`	386	3
SAR *mem*,**1**	`sar count,1`	486	3
		88/86	15+*EA* (*W88*=23+*EA*)
		286	7
		386	7
		486	4
SAL *mem*,**1**	`sal WORD PTR m32[0],1`	88/86	15+*EA* (*W88*=23+*EA*)
SHL *mem*,**1**	`shl index,1`	286	7
SHR *mem*,**1**	`shr unsign[di],1`	386	7
		486	4

Encoding 1101001*w* *mod,TTT*,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
SAR *reg*,**CL**	`sar bx,cl`	88/86	8+4*n*
	`sar dx,cl`	286	5+*n*
		386	3
		486	3
SAL *reg*,**CL**	`shr dx,cl`	88/86	8+4*n*
SHL *reg*,**CL**	`shl di,cl`	286	5+*n*
SHR *reg*,**CL**	`sal ah,cl`	386	3
		486	3
SAR *mem*,**CL**	`sar sign,cl`	88/86	20+*EA*+4*n* (*W88*=28+*EA*+4*n*)
	`sar WORD PTR [bp+8],cl`	286	8+*n*
		386	7
		486	4
SAL *mem*,**CL**	`shr WORD PTR m32[2],cl`	88/86	20+*EA*+4*n*
SHL *mem*,**CL**	`sal BYTE PTR [di],cl`		(*W88*=28+*EA*+4*n*)
SHR *mem*,**CL**	`shl index,cl`	286	8+*n*
		386	7
		486	4

Encoding 1100000w *mod,TTT*,r/m* *disp (0, 1, or 2)* *data (1)*

Syntax	Examples	CPU	Clock Cycles
SAR *reg,immed8*	`sar bx,5`	88/86	—
	`sar cl,5`	286	5+*n*
		386	3
		486	2
SAL *reg,immed8*	`sal cx,6`	88/86	—
SHL *reg,immed8*	`shl di,2`	286	5+*n*
SHR *reg,immed8*	`shr bx,8`	386	3
		486	2
SAR *mem,immed8*	`sar sign_count,3`	88/86	—
	`sar WORD PTR [bx],5`	286	8+*n*
		386	7
		486	4
SAL *reg,immed8*	`shr mem16,11`	88/86	—
SHL *reg,immed8*	`shl unsign,4`	286	8+*n*
SHR *reg,immed8*	`sal array[bx+di],14`	386	7
		486	4

* *TTT* represents one of the following bit codes: 100 for **SHL** or **SAL**, 101 for **SHR**, or 111 for **SAR**.

SHLD/SHRD Double Precision Shift

80386–80486 Only Shifts the bits of the second operand into the first operand. The number of bits shifted is specified by the third operand. **SHLD** shifts the first operand to the left by the number of positions specified in the count. The positions opened by the shift are filled by the most significant bits of the second operand. **SHRD** shifts the first operand to the right by the number of positions specified in the count. The positions opened by the shift are filled by the least significant bits of the second operand. The count operand can be either CL or an 8-bit constant. If a shift count larger than 31 is given, it is adjusted by using the remainder (modulo) of a division by 32.

Flags

O	D	I	T	S	Z	A	P	C
?				±	±	?	±	±

Encoding 00001111 10100100 *mod,reg,r/m* *disp (0, 1, or 2)* *data (1)*

Syntax	Examples	CPU	Clock Cycles
SHLD *reg16,reg16,immed8*	`shld ax,dx,10`	88/86	—
SHLD *reg32,reg32,immed8*		286	—
		386	3
		486	2
SHLD *mem16,reg16,immed8*	`shld bits,cx,5`	88/86	—
SHLD *mem32,reg32,immed8*		286	—
		386	7
		486	3

Encoding 00001111 10101100 *mod,reg,r/m* *disp (0, 1, or 2)* *data (1)*

Syntax	Examples	CPU	Clock Cycles
SHRD *reg16,reg16,immed8*	`shrd cx,si,3`	88/86	—
SHRD *reg32,reg32,immed8*		286	—
		386	3
		486	2
SHRD *mem16,reg16,immed8*	`shrd [di],dx,13`	88/86	—
SHRD *mem32,reg32,immed8*		286	—
		386	7
		486	3

Encoding 00001111 10100101 *mod,reg,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
SHLD *reg16,reg16,***CL**	`shld ax,dx,cl`	88/86	—
SHLD *reg32,reg32,***CL**		286	—
		386	3
		486	3
SHLD *mem16,reg16,***CL**	`shld`	88/86	—
SHLD *mem32,reg32,***CL**	`masker,ax,cl`	286	—
		386	7
		486	4

Encoding 00001111 10101101 *mod,reg,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
SHRD *reg16,reg16,***CL**	`shrd bx,dx,cl`	88/86	—
SHRD *reg32,reg32,***CL**		286	—
		386	3
		486	3
SHRD *mem16,reg16,***CL**	`shrd [bx],dx,cl`	88/86	—
SHRD *mem32,reg32,***CL**		286	—
		386	7
		486	4

SMSW Store Machine Status Word

80286-80486 Only Stores the Machine Status Word (MSW) into a specified memory operand. **SMSW** is generally useful only in protected mode. See Intel documentation for details on the MSW and other protected-mode concepts.

Flags No change

Encoding 00001111 00000001 *mod*,100,*r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
SMSW *reg16*	`smsw ax`	88/86	—
		286	2
		386	2
		486	2
SMSW *mem16*	`smsw machine`	88/86	—
		286	3
		386	3
		486	3

STC Set Carry Flag

Sets the carry flag.

Flags

O	D	I	T	S	Z	A	P	C
								1

Encoding 11111001

Syntax	Examples	CPU	Clock Cycles
STC	`stc`	88/86	2
		286	2
		386	2
		486	2

STD Set Direction Flag

Sets the direction flag. All subsequent string instructions will process down (from high addresses to low addresses).

Flags

O	D	I	T	S	Z	A	P	C
	1							

Encoding 11111101

Syntax	Examples	CPU	Clock Cycles
STD	`std`	88/86	2
		286	2
		386	2
		486	2

STI Set Interrupt Flag

Sets the interrupt flag. When the interrupt flag is set, maskable interrupts are recognized. If interrupts were disabled by a previous **CLI** instruction, pending interrupts will not be executed immediately; they will be executed after the instruction following **STI**.

Flags

O	D	I	T	S	Z	A	P	C
		1						

Encoding 11111011

Syntax	Examples	CPU	Clock Cycles
STI	`sti`	88/86	2
		286	2
		386	3
		486	5

STOS/STOSB/STOSW/STOSD Store String Data

Stores the value of the accumulator in a string. The string is the destination and must be pointed to by ES:DI, even if an operand is given. For each source element loaded, DI is adjusted according to the size of the operand and the status of the direction flag. DI is incremented if the direction flag has been cleared with **CLD** or decremented if the direction flag has been set with **STD**.

If the **STOS** form of the instruction is used, an operand must be provided to indicate the size of the data elements to be processed. No segment override is allowed. If **STOSB** (bytes), **STOSW** (words), or **STOSD** (doublewords on the 80386–80486 only) is used, the instruction determines the size of the data elements to be processed and whether the element comes from AL, AX, or EAX.

STOS and its variations are often used with the **REP** prefix to fill a string with a repeated value. Before the repeated instruction is executed, CX should contain the number of elements to store.

Flags No change

Encoding 1010101*w*

Syntax	Examples	CPU	Clock Cycles
STOS ⟦**ES:**⟧ *dest*	`stos es:dstring`	88/86	11 (*W88*=15)
STOSB ⟦⟦**ES:**⟧ *dest*⟧	`rep stosw`	286	3
STOSW ⟦⟦**ES:**⟧ *dest*⟧	`rep stosb`	386	4
STOSD ⟦⟦**ES:**⟧ *dest*⟧*		486	5

* 80386–80486 only

STR Store Task Register

80286-80486 Only Stores the current task register to the specified operand. This instruction is generally useful only in privileged mode. See Intel documentation for details on task registers and other protected-mode concepts.

Flags No change

Encoding 00001111 00000000 *mod*, 001, *reg* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
STR *reg16*	`str cx`	88/86	—
		286	2
		386	2
		486	2
STR *mem16*	`str taskreg`	88/86	—
		286	3
		386	2
		486	3

SUB Subtract

Subtracts the source operand from the destination operand and stores the result in the destination operand.

Flags

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	±

Encoding 001010*dw* *mod*, *reg*, *r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
SUB *reg,reg*	`sub ax,bx`	88/86	3
	`sub bh,dh`	286	2
		386	2
		486	1
SUB *mem,reg*	`sub tally,bx`	88/86	16+*EA* (*W88*=24+*EA*)
	`sub array[di],bl`	286	7
		386	6
		486	3

Syntax	Examples	CPU	Clock Cycles
SUB *reg,mem*	`sub cx,discard`	88/86	9+*EA* (*W88*=13+*EA*)
	`sub al,[bx]`	286	7
		386	7
		486	2

Encoding 100000*sw* *mod*,101,*r/m* *disp (0, 1, or 2)* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
SUB *reg,immed*	`sub dx,45`	88/86	4
	`sub bl,7`	286	3
		386	2
		486	1
SUB *mem,immed*	`sub total,4000`	88/86	17+*EA* (*W88*=25+*EA*)
	`sub BYTE PTR [bx+di],2`	286	7
		386	7
		486	3

Encoding 0010110*w* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
SUB *accum,immed*	`sub ax,32000`	88/86	4
		286	3
		386	2
		486	1

TEST Logical Compare

Tests specified bits of an operand and sets the flags for a subsequent conditional jump or set instruction. One of the operands contains the value to be tested. The other contains a bit mask indicating the bits to be tested. **TEST** works by doing a bitwise AND operation on the source and destination operands. The flags are modified according to the result, but the destination operand is not changed. This instruction is the same as the **AND** instruction, except the result is not stored.

Flags

O	D	I	T	S	Z	A	P	C
0				±	±	?	±	0

Encoding 1000010*w* *mod, reg, r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
TEST *reg,reg*	`test dx,bx`	88/86	3
	`test bl,ch`	286	2
		386	2
		486	1
TEST *mem,reg*	`test dx,flags`	88/86	9+*EA* (*W88*=13+*EA*)
TEST *reg,mem**	`test bl,bitarray[bx]`	286	6
		386	5
		486	2

Encoding 1111011*w* *mod*,000,*r/m* *disp (0, 1, or 2)* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
TEST *reg,immed*	`test cx,30h`	88/86	5
	`test cl,1011b`	286	3
		386	2
		486	1
TEST *mem,immed*	`test masker,1`	88/86	11+*EA*
	`test BYTE PTR [bx],03h`	286	6
		386	5
		486	2

Encoding 1010100*w* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
TEST *accum,immed*	`test ax,90h`	88/86	4
		286	3
		386	2
		486	1

* MASM transposes **TEST** *reg,mem*; that is, it is encoded as **TEST** *mem,reg*.

VERR/VERW Verify Read or Write

80286-80486 Protected Only Verifies that a specified segment selector is valid and can be read or written to at the current privilege level. **VERR** verifies that the selector is readable. **VERW** verifies that the selector can be written to. If the segment is verified, the zero flag is set. Otherwise, the zero flag is cleared.

Flags

O	D	I	T	S	Z	A	P	C
					±			

Encoding 00001111 00000000 *mod*, 100,*r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
VERR *reg16*	`verr ax`	88/86	—
		286	14
		386	10
		486	11
VERR *mem16*	`verr selector`	88/86	—
		286	16
		386	11
		486	11

Encoding 00001111 00000000 *mod*, 101,*r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
VERW *reg16*	`verw cx`	88/86	—
		286	14
		386	15
		486	11
VERW *mem16*	`verw selector`	88/86	—
		286	16
		386	16
		486	11

WAIT Wait

Suspends processor execution until the processor receives a signal that a coprocessor has finished a simultaneous operation. It should be used to prevent a coprocessor instruction from modifying a memory location that is being modified simultaneously by a processor instruction. **WAIT** is the same as the coprocessor **FWAIT** instruction.

Flags No change

Encoding 10011011

Syntax	Examples	CPU	Clock Cycles
WAIT	`wait`	88/86	4
		286	3
		386	6
		486	1–3

WBINVD Write Back and Invalidate Data Cache

80486 Only Empties the contents of the current data cache after writing changes to memory. Proper use of this instruction requires knowledge of how contents are placed in the cache. **WBINVD** is intended primarily for system programming. See Intel documentation for details.

Flags No change

Encoding 00001111 00001001

Syntax	Examples	CPU	Clock Cycles
WBINVD	`wbinvd`	88/86	—
		286	—
		386	—
		486	5

XADD Exchange and Add

80486 Only Adds the source and destination operands and stores the sum in the destination; simultaneously, the original value of the destination is moved to the source. The instruction sets flags according to the result of the addition.

Flags

O	D	I	T	S	Z	A	P	C
±				±	±	±	±	±

Encoding 00001111 1100000*b* *mod, reg, r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
XADD *mem,reg*	`xadd warr[bx],ax`	88/86	—
	`xadd string,bl`	286	—
		386	—
		486	4
XADD *reg,reg*	`xadd dl,al`	88/86	—
	`xadd bx,dx`	286	—
		386	—
		486	3

XCHG Exchange

Exchanges the values of the source and destination operands.

Flags No change

Encoding 1000011*w* *mod,reg,r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
XCHG *reg,reg*	`xchg cx,dx`	88/86	4
	`xchg bl,dh`	286	3
	`xchg al,ah`	386	3
		486	3
XCHG *reg,mem*	`xchg [bx],ax`	88/86	17+*EA* (*W88*=25+*EA*)
XCHG *mem,reg*	`xchg bx,pointer`	286	5
		386	5
		486	5

Encoding 10010 *reg*

Syntax	Examples	CPU	Clock Cycles
XCHG *accum,reg16**	`xchg ax,cx`	88/86	3
XCHG *reg16,accum**	`xchg cx,ax`	286	3
		386	3
		486	3

* On the 80386–80486, the accumulator may also be exchanged with a 32-bit register.

XLAT/XLATB Translate

Translates a value from one coding system to another by looking up the value to be translated in a table stored in memory. Before the instruction is executed, BX should point to a table in memory and AL should contain the unsigned position of the value to be translated from the table. After the instruction, AL contains the table value at the specified position. No operand is required, but one can be given to specify a segment override. DS is assumed unless a segment override is given. **XLATB** is a synonym for **XLAT**. Either version allows an operand, but neither requires one.

Flags No change

Encoding 11010111

Syntax	Examples	CPU	Clock Cycles
XLAT [[*segreg*:] *mem*]	`xlat`	88/86	11
XLATB [[*segreg*:] *mem*]	`xlatb es:table`	286	5
		386	5
		486	4

XOR Exclusive OR

Performs a bitwise exclusive OR operation on the source and destination operands and stores the result in the destination. For each bit position in the operands, if both bits are set or if both bits are cleared, the corresponding bit of the result is cleared. Otherwise, the corresponding bit of the result is set.

Flags

O	D	I	T	S	Z	A	P	C
0				±	±	?	±	0

Encoding 001100*dw* *mod, reg, r/m* *disp (0, 1, or 2)*

Syntax	Examples	CPU	Clock Cycles
XOR *reg,reg*	`xor cx,bx`	88/86	3
	`xor ah,al`	286	2
		386	2
		486	1
XOR *mem,reg*	`xor [bp+10],cx`	88/86	16+*EA* (*W88*=24+*EA*)
	`xor masked,bx`	286	7
		386	6
		486	3
XOR *reg,mem*	`xor cx,flags`	88/86	9+*EA* (*W88*=13+*EA*)
	`xor bl,bitarray[di]`	286	7
		386	7
		486	2

Encoding 100000*sw* *mod*,110,*r/m* *disp (0, 1, or 2)* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
XOR *reg,immed*	`xor bx,10h`	88/86	4
	`xor bl,1`	286	3
		386	2
		486	1
XOR *mem,immed*	`xor Boolean,1`	88/86	17+*EA* (*W88*=25+*EA*)
	`xor switches[bx],101b`	286	7
		386	7
		486	3

Encoding 0011010*w* *data (1 or 2)*

Syntax	Examples	CPU	Clock Cycles
XOR *accum,immed*	`xor ax,01010101b`	88/86	4
		286	3
		386	2
		486	1

C H A P T E R 5

Coprocessor

Topical Cross-reference for Coprocessor Instructions 146
Interpreting Coprocessor Instructions. 148
Syntax 148
Examples 148
Clock Speeds 148
Instruction Size. 148
Architecture 149

Topical Cross-reference for Coprocessor Instructions

Arithmetic

FABS	FADD/FIADD	FADDP
FCHS	FDIV/FIDIV	FDIVP
FDIVR/FIDIVR	FDIVRP	FMUL/FIMUL
FMULP	FPREM	FPREM1§
FRNDINT	FSCALE	FSQRT
FSUB/FISUB	FSUBP	FSUBR/FISUBR
FSUBRP	FXTRACT	

Compare

FCOM/FICOM	FCOMP/FICOMP	FCOMPP
FSTSW/FNSTSW	FTST	FUCOM§
FUCOMP§	FUCOMPP§	FXAM

Load

FLD/FILD/FBLD	FLDCW	FLDENV
FRSTOR	FXCH	

Load Constant

FLD1	FLDL2E	FLDL2T
FLDLG2	FLDLN2	FLDPI
FLDZ		

Processor Control

FCLEX/FNCLEX	FDECSTP	FDISI/FNDISI*
FENI/FNENI*	FFREE	FINCSTP
FINIT/FNINIT	FLDCW	FNOP
FRSTOR	FSAVE/FNSAVE	FSETPM-
FSTCW/FNSTCW	FSTENV/FNSTENV	FSTSW/FNSTSW
FWAIT		

Store Data

FSAVE/FNSAVE	FST/FIST	FSTCW/FNSTCW
FSTENV/FNSTENV	FSTP/FISTP/FBSTP	FSTSW/FNSTSW

Transcendental

F2XM1	FCOS§	FPATAN
FPREM	FPREM1§	FPTAN
FSIN§	FSINCOS§	FYL2P1
FYL2X		

* 8087 only † 80287 only. § 80387–80486 only.

Interpreting Coprocessor Instructions

This section provides an alphabetical reference to instructions of the 8087, 80287, and 80387 coprocessors. The format is the same as the processor instructions except that encodings are not provided. Differences are noted in the following.

The 80486 has the coprocessor built in. This one chip executes all the instructions listed in the previous section and this section.

Syntax

Syntaxes in Column 1 use the following abbreviations for operand types:

Syntax	Operand
reg	A coprocessor stack register
memreal	A direct or indirect memory operand storing a real number
memint	A direct or indirect memory operand storing a binary integer
membcd	A direct or indirect memory operand storing a BCD number

Examples

The position of the examples in Column 2 is not related to the clock speeds in Column 3.

Clock Speeds

Column 3 shows the clock speeds for each processor. Sometimes an instruction may have more than one possible clock speed. The following abbreviations are used to specify variations:

Abbreviation	Description
EA	Effective address. This applies only to the 8087. See the Processor Section, "Timings on the 8088 and 8086 Processors," for an explanation of effective address timings.
s,l,t	Short real, long real, and 10-byte temporary real.
w,d,q	Word, doubleword, and quadword binary integer.
to, fr	To or from stack top. On the 80387 and 80486, the *to* clocks represent timings when ST is the destination. The *fr* clocks represent timings when ST is the source.

Instruction Size

The instruction size is always 2 bytes for instructions that do not access memory. For instructions that do access memory, the size is 4 bytes on the 8087 and 80287. On the 80387 and 80486, the size for instructions that access memory is 4 bytes in 16-bit mode, or 6 bytes in 32-bit mode.

On the 8087, each instruction must be preceded by the **WAIT** (also called **FWAIT**) instruction, thereby increasing the instruction's size by 1 byte. The assembler inserts **WAIT** automatically by default, or with the **.8087** directive.

Architecture

The 8087, 80287, and 80387 coprocessors, along with the 80486, have several common elements of architecture. All have a register stack made up of eight 80-bit data registers. These can contain floating-point numbers in the temporary real format. The coprocessors also have 14 bytes of control registers. Figure 5.1 shows the format of registers.

Fig. 5.1 Coprocessor Registers

The most important control registers are the control word and the status word. Figure 5.2 shows the format of these registers.

Fig. 5.2 Control Word and Status Word

F2XM1 2^X–1

Calculates $Y = 2^X - 1$. X is taken from ST. The result, Y, is returned in ST. X must be in the range $0 \le X \le 0.5$ on the 8087/287, or in the range $-1.0 \le X \le +1.0$ on the 80387–80486.

Syntax	Examples	CPU	Clock Cycles
F2XM1	`f2xm1`	87	310–630
		287	310–630
		387	211–476
		486	140–279

FABS Absolute Value

Converts the element in ST to its absolute value.

Syntax	Examples	CPU	Clock Cycles
FABS	`fabs`	87	10–17
		287	10–17
		387	22
		486	3

FADD/FADDP/FIADD Add

Adds the source to the destination and returns the sum in the destination. If two register operands are specified, one must be ST. If a memory operand is specified, the sum replaces the value in ST. Memory operands can be 32- or 64-bit real numbers or 16- or 32-bit integers. If no operand is specified, ST is added to ST(1) and the stack is popped, returning the sum in ST. For **FADDP**, the source must be ST; the sum is returned in the destination and ST is popped.

Syntax	Examples	CPU	Clock Cycles

FADD ⟦*reg*,*reg*⟧	`fadd st,st(2)`
	`fadd st(5),st`
	`fadd`
		87
		287
		387
		486
			70–100
			70–100
			to=23–31, *fr*=26–34
			8–20

FADDP *reg*,**ST**	`faddp st(6),st`	87 287 387 486	75–105 75–105 23–31 8–20
FADD *memreal*	`fadd QWORD PTR [bx]` `fadd shortreal`	87 287 387 486	(*s*=90–120,*s*=95–125)+*EA* *s*=90–120,*l*=95–125 *s*=24–32,*l*=29–37 8–20

FIADD *memint*	`fiadd int16`
	`fiadd warray[di]`
	`fiadd double`
		87
		
		287
		
		387
		486
			(*w*=102–137,*d*=108–143)+*EA*
			w=102–137,*d*=108–143
			w=71–85,*d*=57–72
			w=20–35,*d*=19–32

FBLD Load BCD

See **FLD**.

FBSTP Store BCD and Pop

See **FST**.

FCHS Change Sign

Reverses the sign of the value in ST.

Syntax	Examples	CPU	Clock Cycles
FCHS	`fchs`	87	10–17
		287	10–17
		387	24–25
		486	6

FCLEX/FNCLEX Clear Exceptions

Clears all exception flags, the busy flag, and bit 7 in the status word. Bit 7 is the interrupt-request flag on the 8087, and the error-status flag on the 80287, 80387, and 80486. The instruction has wait and no-wait versions.

Syntax	Examples	CPU	Clock Cycles*
FCLEX	`fclex`	87	2–8
FNCLEX		287	2–8
		387	11
		486	7

* These timings reflect the no-wait version of the instruction. The wait version may take additional clock cycles.

FCOM/FCOMP/FCOMPP/FICOM/FICOMP Compare

Compares the specified source operand to ST and sets the condition codes of the status word according to the result. The instruction subtracts the source operand from ST without changing either operand. Memory operands can be 32- or 64-bit real numbers or 16- or 32-bit integers. If no operand is specified or if two pops are specified, ST is compared to ST(1) and the stack is popped. If one pop is specified with an operand, the operand is compared to ST. If one of the operands is a NAN, an invalid-operation exception occurs (see **FUCOM** for an alternative method of comparing on the 80387–80486).

Syntax	Examples	CPU	Clock Cycles
FCOM ⟦*reg*⟧	`fcom st(2)` `fcom`	87 287 387 486	40–50 40–50 24 4
FCOMP ⟦*reg*⟧	`fcomp st(7)` `fcomp`	87 287 387 486	42–52 42–52 26 4
FCOMPP	`fcompp`	87 287 387 486	45–55 45–55 26 5
FCOM *memreal*	`fcom shortreals[di]` `fcom longreal`	87 287 387 486	(*s*=60–70,*l*=65–75)+*EA* *s*=60–70,*l*=65–75 *s*=26,*l*=31 4
FCOMP *memreal*	`fcomp longreal` `fcomp shorts[di]`	87 287 387 486	(*s*=63–73,*l*=67–77)+*EA* *s*=63–73,*l*=67–77 *s*=26,*l*=31 4
FICOM *memint*	`ficom double` `ficom warray[di]`	87 287 387 486	(*w*=72–86,*d*=78–91)+*EA* *w*=72–86,*d*=78–91 *w*=71–75,*d*=56–63 *w*=16–20,*d*=15–17
FICOMP *memint*	`ficomp WORD PTR [bp+6]` `ficomp darray[di]`	87 287 387 486	(*w*=74–88,*d*=80–93)+*EA* *w*=74–88,*d*=80–93 *w*=71–75,*d*=56–63 *w*=16–20,*d*=15–17

Condition Codes for FCOM

C3	C2	C1	C0	Meaning
0	0	?	0	ST > source
0	0	?	1	ST < source
1	0	?	0	ST = source
1	1	?	1	ST is not comparable to source

FCOS Cosine

80387–80486 Only Replaces a value in radians in ST with its cosine. If |ST| < 2^{63}, the C2 bit of the status word is cleared and the cosine is calculated. Otherwise, C2 is set and no calculation is performed. ST can be reduced to the required range with **FPREM** or **FPREM1**.

Syntax	Examples	CPU	Clock Cycles
FCOS	`fcos`	87	—
		287	—
		387	123–772*
		486	257–354†

* For operands with an absolute value greater than π/4, up to 76 additional clocks may be required.

† For operands with an absolute value greater than π/4, add *n* clocks where *n* = *operand*/(π/4).

FDECSTP Decrement Stack Pointer

Decrements the stack-top pointer in the status word. No tags or registers are changed, and no data is transferred. If the stack pointer is 0, **FDECSTP** changes it to 7.

Syntax	Examples	CPU	Clock Cycles
FDECSTP	`fdecstp`	87	6–12
		287	6–12
		387	22
		486	3

FDISI/FNDISI Disable Interrupts

8087 Only Disables interrupts by setting the interrupt-enable mask in the control word. This instruction has wait and no-wait versions. Since the 80287, 80387, and 80486 do not have an interrupt-enable mask, the instruction is recognized but ignored on these coprocessors.

Syntax	Examples	CPU	Clock Cycles*
FDISI	`fdisi`	87	2–8
FNDISI		287	2
		387	2
		486	3

* These timings reflect the no-wait version of the instruction. The wait version may take additional clock cycles.

FDIV/FDIVP/FIDIV Divide

Divides the destination by the source and returns the quotient in the destination. If two register operands are specified, one must be ST. If a memory operand is specified, the quotient replaces the value in ST. Memory operands can be 32- or 64-bit real numbers or 16- or 32-bit integers. If no operand is specified, ST(1) is divided by ST and the stack is popped, returning the result in ST. For **FDIVP**, the source must be ST; the quotient is returned in the destination register and ST
is popped.

Syntax	Examples	CPU	Clock Cycles
FDIV [*reg*,*reg*]	`fdiv st,st(2)`	87	193–203
	`fdiv st(5),st`	287	193–203
		387	*to*=88, *fr*=91
		486	73
FDIVP *reg*,**ST**	`fdivp st(6),st`	87	197–207
		287	197–207
		387	91
		486	73
FDIV *memreal*	`fdiv DWORD PTR [bx]`	87	(*s*=215–225,*l*=220–230)+*EA*
	`fdiv shortreal[di]`		
	`fdiv longreal`	287	*s*=215–225,*l*=220–230
		387	*s*=89,*l*=94
		486	73
FIDIV *memint*	`fidiv int16`	87	(*w*=224–238,*d*=230–243)+*EA*
	`fidiv warray[di]`		
	`fidiv double`	287	*w*=224–238,*d*=230–243
		387	*w*=136–140,*d*=120–127
		486	*w*=85–89,*d*=84–86

FDIVR/FDIVRP/FIDIVR Divide Reversed

Divides the source by the destination and returns the quotient in the destination. If two register operands are specified, one must be ST. If a memory operand is specified, the quotient replaces the value in ST. Memory operands can be 32- or 64-bit real numbers or 16- or 32-bit integers. If no operand is specified, ST is divided by ST(1) and the stack is popped, returning the result in ST. For **FDIVRP**, the source must be ST; the quotient is returned in the destination register and ST is popped.

Syntax	Examples	CPU	Clock Cycles

FDIVR [[*reg*,*reg*]]	`fdivr st,st(2)`
	`fdivr st(5),st`
	`fdivr`
		87
		287
		387
		486
			194–204
			194–204
			to=88, *fr*=91
			73

FDIVRP *reg*,**ST**	`fdivrp st(6),st`	87 287 387 486	198–208 198–208 91 73
FDIVR *memreal*	`fdivr longreal` `fdivr shortreal[di]`	87 287 387 486	(*s*=216–226,*l*=221–231)+*EA* *s*=216–226,*l*=221–231 *s*=89,*l*=94 73
FIDIVR *memint*	`fidivr double` `fidivr warray[di]`	87 287 387 486	(*w*=225–239,*d*=231–245)+*EA* *w*=225–239,*d*=231–245 *w*=135–141,*d*=121–128 *w*=85–89,*d*=84–86

FENI/FNENI Enable Interrupts

8087 Only Enables interrupts by clearing the interrupt-enable mask in the control word. This instruction has wait and no-wait versions. Since the 80287, 80387, and 80486 do not have interrupt-enable masks, the instruction is recognized but ignored on these coprocessors.

Syntax	Examples	CPU	Clock Cycles*
FENI	`feni`	87	2–8
FNENI		287	2
		387	2
		486	3

* These timings reflect the no-wait version of the instruction. The wait version may take additional clock cycles.

FFREE Free Register

Changes the specified register's tag to empty without changing the contents of the register.

Syntax	Examples	CPU	Clock Cycles
FFREE ST(*i*)	`ffree st(3)`	87	9–16
		287	9–16
		387	18
		486	3

FIADD/FISUB/FISUBR/ FIMUL/FIDIV/FIDIVR Integer Arithmetic

See **FADD**, **FSUB**, **FSUBR**, **FMUL**, **FDIV**, and **FDIVR**.

FICOM/FICOMP Compare Integer

See **FCOM**.

FILD Load Integer

See **FLD**.

FINCSTP Increment Stack Pointer

Increments the stack-top pointer in the status word. No tags or registers are changed, and no data is transferred. If the stack pointer is 7, **FINCSTP** changes it
to 0.

Syntax	Examples	CPU	Clock Cycles
FINCSTP	`fincstp`	87	6–12
		287	6–12
		387	21
		486	3

FINIT/FNINIT Initialize Coprocessor

Initializes the coprocessor and resets all the registers and flags to their default values. The instruction has wait and no-wait versions. On the 80387–80486, the condition codes of the status word are cleared. On the 8087/287, they are unchanged.

Syntax	Examples	CPU	Clock Cycles*
FINIT	`finit`	87	2–8
FNINIT		287	2–8
		387	33
		486	17

* These timings reflect the no-wait version of the instruction. The wait version may take additional clock cycles.

FIST/FISTP Store Integer

See **FST**.

FLD/FILD/FBLD Load

Pushes the specified operand onto the stack. All memory operands are automatically converted to temporary-real numbers before being loaded. Memory operands can be 32-, 64-, or 80-bit real numbers or 16-, 32-, or 64-bit integers.

Syntax	Examples	CPU	Clock Cycles
FLD *reg*	`fld st(3)`	87	17–22
		287	17–22
		387	14
		486	4
FLD *memreal*	`fld longreal`	87	(*s*=38–56,*l*=40–60,*t*=53–65)+*EA*
	`fld shortarray[bx+di]`		
		287	*s*=38–56,*l*=40–60,*t*=53–65
	`fld tempreal`		
		387	*s*=20,*1*=25,*t*=44
		486	*s*=3,*l*=3,*t*=6
FILD *memint*	`fild mem16`	87	(*w*=46–54,*d*=52–60,*q*=60–68)+*EA*
	`fild DWORD PTR [bx]`		
	`fild quads[si]`	287	*w*=46-54,*d*=52-60,*q*=60-68
		387	*w*=61–65,*d*=45–52,*q*=56–67
		486	*w*=13–16,*d*=9–12,*q*=10–18
FBLD *membcd*	`fbld packbcd`	87	(290–310)+*EA*
		287	290–310
		387	266–275
		486	70–103

FLD1/FLDZ/FLDPI/FLDL2E/ FLDL2T/FLDLG2/FLDLN2 Load Constant

Pushes a constant onto the stack. The following constants can be loaded:

Instruction	Constant
FLD1	+1.0
FLDZ	+0.0
FLDPI	π

Instruction	Constant
FLDL2E	$Log_2(e)$
FLDL2T	$Log_2(10)$
FLDLG2	$Log_{10}(2)$
FLDLN2	$Log_e(2)$

Syntax	Examples	CPU	Clock Cycles
FLD1	`fld1`	87	15–21
		287	15–21
		387	24
		486	4
FLDZ	`fldz`	87	11–17
		287	11–17
		387	20
		486	4
FLDPI	`fldpi`	87	16–22
		287	16–22
		387	40
		486	8
FLDL2E	`fldl2e`	87	15–21
		287	15–21
		387	40
		486	8
FLDL2T	`fldl2t`	87	16–22
		287	16–22
		387	40
		486	8
FLDLG2	`fldlg2`	87	18–24
		287	18–24
		387	41
		486	8
FLDLN2	`fldln2`	87	17–23
		287	17–23
		387	41
		486	8

FLDCW Load Control Word

Loads the specified word into the coprocessor control word. The format of the control word is shown in the "Interpreting Coprocessor Instructions" section.

Syntax	Examples	CPU	Clock Cycles
FLDCW *mem16*	`fldcw ctrlword`	87	(7–14)+*EA*
		287	7–14
		387	19
		486	4

FLDENV/FLDENVW/FLDENVD Load Environment State

Loads the 14-byte coprocessor environment state from a specified memory location. The environment includes the control word, status word, tag word, instruction pointer, and operand pointer. On the 80387–80486 in 32-bit mode, the environment state is 28 bytes.

Syntax	Examples	CPU	Clock Cycles
FLDENV *mem*	`fldenv [bp+10]`	87	(35–45)+*EA*
FLDENVW *mem**		287	35–45
FLDENVD *mem**		387	71
		486	44,*pm*=34

* 80387–80486 only.

FMUL/FMULP/FIMUL Multiply

Multiplies the source by the destination and returns the product in the destination. If two register operands are specified, one must be ST. If a memory operand is specified, the product replaces the value in ST. Memory operands can be 32- or 64-bit real numbers or 16- or 32-bit integers. If no operand is specified, ST(1) is multiplied by ST and the stack is popped, returning the product in ST. For **FMULP**, the source must be ST; the product is returned in the destination register and ST is popped.

Syntax	Examples		CPU	Clock Cycles
FMUL [[*reg*,*reg*]]	`fmul`	`st,st(2)`	87	130–145 (90–105)*
	`fmul`	`st(5),st`	287	130–145 (90–105)*
	`fmul`		387	*to*=46–54 (49), *fr*= 29–57 (52)†
			486	16
FMULP *reg*,**ST**	`fmulp`	`st(6),st`	87	134–148 (94–108)*
			287	134–148 (94–108)*
			387	29–57 (52)†
			486	16
FMUL *memreal*	`fmul`	`DWORD PTR [bx]`	87	(*s*=110–125,*l*=154–168)+*EA*§
	`fmul`	`shortreal[di+3]`		
	`fmul`	`longreal`	287	*s*=110–125,*l*=154 –168§
			387	*s*=27–35,*l*=32–57
			486	*s*=11,*l*=14
FIMUL *memint*	`fimul`	`int16`	87	(*w*=124–138,*d*=130 –144)+*EA*
	`fimul`	`warray[di]`		
	`fimul`	`double`	287	*w*=124–138,*d*=130 –144
			387	*w*=76–87,*d*=61–82
			486	*w*=23–27,*d*=22–24

* The clocks in parentheses show times for short values—those with 40 trailing zeros in their fraction because they were loaded from a short-real memory operand.

† The clocks in parentheses show typical speeds.

§ If the register operand is a short value—having 40 trailing zeros in its fraction because it was loaded from a short-real memory operand—then the timing is (112–126)+*EA* on the 8087 or 112–126 on the 80287.

FN*instruction* No-Wait Instructions

Instructions that have no-wait versions include **FCLEX**, **FDISI**, **FENI**, **FINIT**, **FSAVE**, **FSTCW**, **FSTENV**, and **FSTSW**. Wait versions of instructions check for unmasked numeric errors; no-wait versions do not. When the **.8087** directive is used, the assembler puts a **WAIT** instruction before the wait versions and a **NOP** instruction before the no-wait versions.

FNOP No Operation

Performs no operation. **FNOP** can be used for timing delays or alignment.

Syntax	Examples	CPU	Clock Cycles
FNOP	`fnop`	87	10–16
		287	10–16
		387	12
		486	3

FPATAN Partial Arctangent

Finds the partial tangent by calculating Z = ARCTAN(Y / X). X is taken from ST and Y from ST(1). On the 8087/287, Y and X must be in the range $0 \leq Y < X < \infty$. On the 80387–80486, there is no restriction on X and Y. X is popped from the stack and Z replaces Y in ST.

Syntax	Examples	CPU	Clock Cycles
FPATAN	`fpatan`	87	250–800
		287	250–800
		387	314–487
		486	218–303

FPREM Partial Remainder

Calculates the remainder of ST divided by ST(1), returning the result in ST. The remainder retains the same sign as the original dividend. The calculation uses the following formula:

```
remainder = ST – ST(1) * quotient
```

The *quotient* is the exact value obtained by chopping ST / ST(1) toward 0. The instruction is normally used in a loop that repeats until the reduction is complete, as indicated by the condition codes of the status word.

Syntax	Examples	CPU	Clock Cycles
FPREM	`fprem`	87	15–190
		287	15–190
		387	74–155
		486	70–138

Condition Codes for FPREM and FPREM1

C3	C2	C1	C0	Meaning
?	1	?	?	Incomplete reduction
0	0	0	0	*quotient* MOD 8 = 0
0	0	0	1	*quotient* MOD 8 = 4
0	0	1	0	*quotient* MOD 8 = 1
0	0	1	1	*quotient* MOD 8 = 5
1	0	0	0	*quotient* MOD 8 = 2
1	0	0	1	*quotient* MOD 8 = 6
1	0	1	0	*quotient* MOD 8 = 3
1	0	1	1	*quotient* MOD 8 = 7

FPREM1 Partial Remainder (IEEE Compatible)

80387–80486 Only Calculates the remainder of ST divided by ST(1), returning the result in ST. The remainder retains the same sign as the original dividend. The calculation uses the following formula:

```
remainder = ST – ST(1) * quotient
```

The *quotient* is the integer nearest to the exact value of ST / ST(1). When two integers are equally close to the given value, the even integer is used. The instruction is normally used in a loop that repeats until the reduction is complete, as indicated by the condition codes of the status word. See **FPREM** for the possible condition codes.

Syntax	Examples	CPU	Clock Cycles
FPREM1	`fprem1`	87	—
		287	—
		387	95–185
		486	72–167

FPTAN Partial Tangent

Finds the partial tangent by calculating Y / X = TAN(Z). Z is taken from ST. Z must be in the range $0 \le Z \le \pi / 4$ on the 8087/287. On the 80387–80486, |Z| must be less than 2^{63}. The result is the ratio Y / X. Y replaces Z, and X is pushed into ST. Thus, Y is returned in ST(1) and X in ST.

Syntax	Examples	CPU	Clock Cycles
FPTAN	`fptan`	87	30–540
		287	30–540
		387	191–497*
		486	200–273†

* For operands with an absolute value greater than π/4, up to 76 additional clocks may be required.

† For operands with an absolute value greater than π/4, add *n* clocks where *n* = *operand*/(π/4).

FRNDINT Round to Integer

Rounds ST from a real number to an integer. The rounding control (RC) field of the control word specifies the rounding method, as shown in the introduction to this section.

Syntax	Examples	CPU	Clock Cycles
FRNDINT	`frndint`	87	16–50
		287	16–50
		387	66–80
		486	21–30

FRSTOR/FRSTORW/FRSTORD Restore Saved State

Restores the 94-byte coprocessor state to the coprocessor from the specified memory location. In 32-bit mode on the 80387–80486, the environment state takes 108 bytes.

Syntax	Examples	CPU	Clock Cycles
FRSTOR *mem*	`frstor [bp-94]`	87	(197–207)+*EA*
FRSTORW *mem**		287	†
FRSTORD *mem**		387	308
		486	131,*pm*=120

* 80387–80486 only.

† Clock counts are not meaningful in determining overall execution time of this instruction. Timing is determined by operand transfers.

FSAVE/FSAVEW/FSAVED/FNSAVE/ FNSAVEW/FNSAVED Save Coprocessor State

Stores the 94-byte coprocessor state to the specified memory location. In 32-bit mode on the 80387–80486, the environment state takes 108 bytes. This instruction has wait and no-wait versions. After the save, the coprocessor is initialized as if **FINIT** had been executed.

Syntax	Examples	CPU	Clock Cycles§
FSAVE *mem*	`fsave [bp-94]`	87	(197–207)+*EA*
FSAVEW *mem**	`fsave cobuffer`	287	†
FSAVED *mem**		387	375–376
FNSAVE *mem*		486	154,*pm*=143
FNSAVEW *mem**			
FNSAVED *mem**			

* 80387–80486 only.

† Clock counts are not meaningful in determining overall execution time of this instruction. Timing is determined by operand transfers.

§ These timings reflect the no-wait version of the instruction. The wait version may take additional clock cycles.

FSCALE Scale

Scales by powers of 2 by calculating the function $Y = Y * 2^X$. X is the scaling factor taken from ST(1), and Y is the value to be scaled from ST. The scaled result replaces the value in ST. The scaling factor remains in ST(1). If the scaling factor is not an integer, it will be truncated toward zero before the scaling.

On the 8087/287, if X is not in the range $-2^{15} \leq X < 2^{15}$ or if X is in the range $0 < X < 1$, the result will be undefined. The 80387–80486 have no restrictions on the range of operands.

Syntax	Examples	CPU	Clock Cycles
FSCALE	`fscale`	87	32–38
		287	32–38
		387	67–86
		486	30–32

FSETPM Set Protected Mode

80287 Only Sets the 80287 to protected mode. The instruction and operand pointers are in the protected-mode format after this instruction. On the 80387–80486, **FSETPM** is recognized but interpreted as **FNOP**, since the 80386/486 processors handle addressing identically in real and protected mode.

Syntax	Examples	CPU	Clock Cycles
FSETPM	`fsetpm`	87	—
		287	2–8
		387	12
		486	3

FSIN Sine

80387–80486 Only Replaces a value in radians in ST with its sine. If $|ST| < 2^{63}$, the C2 bit of the status word is cleared and the sine is calculated. Otherwise, C2 is set and no calculation is performed. ST can be reduced to the required range with **FPREM** or **FPREM1**.

Syntax	Examples	CPU	Clock Cycles
FSIN	`fsin`	87	—
		287	—
		387	122–771*
		486	257–354†

* For operands with an absolute value greater than π/4, up to 76 additional clocks may be required.

† For operands with an absolute value greater than π/4, add *n* clocks where *n* = *operand*/(π/4).

FSINCOS Sine and Cosine

80387–80486 Only Computes the sine and cosine of a radian value in ST. The sine replaces the value in ST, and then the cosine is pushed onto the stack. If $|ST| < 2^{63}$, the C2 bit of the status word is cleared and the sine and cosine are calculated. Otherwise, C2 is set and no calculation is performed. ST can be reduced to the required range with **FPREM** or **FPREM1**.

Syntax	Examples	CPU	Clock Cycles
FSINCOS	`fsincos`	87	—
		287	—
		387	194–809*
		486	292–365†

* For operands with an absolute value greater than π/4, up to 76 additional clocks may be required.

† For operands with an absolute value greater than π/4, add *n* clocks where *n* = *operand*/(π/4).

FSQRT Square Root

Replaces the value of ST with its square root. (The square root of –0 is –0.)

Syntax	Examples	CPU	Clock Cycles
FSQRT	`fsqrt`	87	180–186
		287	180–186
		387	122–129
		486	83–87

FST/FSTP/FIST/FISTP/FBSTP Store

Stores the value in ST to the specified memory location or register. Temporary-real values in registers are converted to the appropriate integer, BCD, or floating-point format as they are stored. With **FSTP**, **FISTP**, and **FBSTP**, the ST register value is popped off the stack. Memory operands can be 32-, 64-, or 80-bit real numbers for **FSTP** or 16-, 32-, or 64-bit integers for **FISTP**.

Syntax	Examples	CPU	Clock Cycles
FST *reg*	`fst st(6)` `fst st`	87	15–22
		287	15–22
		387	11
		486	3
FSTP *reg*	`fstp st` `fstp st(3)`	87	17–24
		287	17–24
		387	12
		486	3
FST *memreal*	`fst shortreal` `fst longs[bx]`	87	(*s*=84–90,*l*=96–104)+*EA*
		287	*s*=84–90,*l*=96–104
		387	*s*=44,*l*=45
		486	*s*=7,*l*=8
FSTP *memreal*	`fstp longreal` `fstp tempreals[bx]`	87	(*s*=86–92,*l*=98–106, *t*=52–58)+*EA*
		287	*s*=86–92,*l*=98–106, *t*=52–58
		387	*s*=44,*l*=45,*t*=53
		486	*s*=7,*l*=8,*t*=6

Syntax	Examples	CPU	Clock Cycles
FIST *memint*	`fist int16` `fist doubles[8]`	87	(*w*=80–90,*d*=82–92)+*EA*
		287	*w*=80–90,*d*=82–92
		387	*w*=82-95,*d*=79-93
		486	*w*=29–34,*d*=28–34
FISTP *memint*	`fistp longint` `fistp doubles[bx]`	87	(*w*=82–92,*d*=84–94, *q*=94–105)+*EA*
		287	*w*=82–92,*d*=84–94, *q*=94–105
		387	*w*=82–95,*d*=79–93, *q*=80–97
		486	29–34
FBSTP *membcd*	`fbstp bcds[bx]`	87	(520–540)+*EA*
		287	520–540
		387	512–534
		486	172–176

FSTCW/FNSTCW Store Control Word

Stores the control word to a specified 16-bit memory operand. This instruction has wait and no-wait versions.

Syntax	Examples	CPU	Clock Cycles*
FSTCW *mem16* **FNSTCW** *mem16*	`fstcw ctrlword`	87	12–18
		287	12–18
		387	15
		486	3

* These timings reflect the no-wait version of the instruction. The wait version may take additional clock cycles.

FSTENV/FSTENVW/FSTENVD/FNSTENV/FNSTENVW/ FNSTENVD Store Environment State

Stores the 14-byte coprocessor environment state to a specified memory location. The environment state includes the control word, status word, tag word, instruction pointer, and operand pointer. On the 80387–80486 in 32-bit mode, the environment state is 28 bytes.

Syntax	Examples	CPU	Clock Cycles†
FSTENV *mem*	`fstenv [bp-14]`	87	(40–50)+*EA*
FSTENVW *mem**		287	40–50
FSTENVD *mem**		387	103–104
FNSTENV *mem*		486	67,*pm*=56
FNSTENVW *mem**			
FNSTENVD *mem**			

* 80387–80486 only.

† These timings reflect the no-wait version of the instruction. The wait version may take additional clock cycles.

FSTSW/FNSTSW Store Status Word

Stores the status word to a specified 16-bit memory operand. On the 80287, 80387, and 80486, the status word can also be stored to the processor's AX register. This instruction has wait and no-wait versions.

Syntax	Examples	CPU	Clock Cycles*
FSTSW *mem16*	`fstsw statword`	87	12–18
FNSTSW *mem16*		287	12–18
		387	15
		486	3
FSTSW AX	`fstsw ax`	87	—
FNSTSW AX		287	10–16
		387	13
		486	3

* These timings reflect the no-wait version of the instruction. The wait version may take additional clock cycles.

FSUB/FSUBP/FISUB Subtract

Subtracts the source operand from the destination operand and returns the difference in the destination operand. If two register operands are specified, one must be ST. If a memory operand is specified, the result replaces the value in ST. Memory operands can be 32- or 64-bit real numbers or 16- or 32-bit integers. If no operand is specified, ST is subtracted from ST(1) and the stack is popped, returning the difference in ST. For **FSUBP**, the source must be ST; the difference (destination minus source) is returned in the destination register and ST is popped.

Syntax	Examples	CPU	Clock Cycles

FSUB ⟦*reg*,*reg*⟧	`fsub st,st(2)`
	`fsub st(5),st`
	`fsub`
		87
		287
		387
		486
			70–100
			70–100
			to=29–37, *fr*=26–34
			8–20

FSUBP *reg*,**ST**	`fsubp st(6),st`	87 287 387 486	75–105 75–105 26–34 8–20
FSUB *memreal*	`fsub longreal` `fsub shortreals[di]`	87 287 387 486	(*s*=90–120,*s*=95–125)+*EA* *s*=90–120,*l*=95–125 *s*=24–32,*l*=28–36 8–20
FISUB *memint*	`fisub double` `fisub warray[di]`	87 287 387 486	(*w*=102–137,*d*=108-143)+*EA* *w*=102–137,*d*=108–143 *w*=71–83,*d*=57–82 *w*=20–35,*d*=19–32

FSUBR/FSUBRP/FISUBR Subtract Reversed

Subtracts the destination operand from the source operand and returns the result in the destination operand. If two register operands are specified, one must be ST. If a memory operand is specified, the result replaces the value in ST. Memory operands can be 32- or 64-bit real numbers or 16- or 32-bit integers. If no operand is specified, ST(1) is subtracted from ST and the stack is popped, returning the difference in ST. For **FSUBRP**, the source must be ST; the difference (source minus destination) is returned in the destination register and ST is popped.

Syntax	Examples	CPU	Clock Cycles

FSUBR ⟦*reg*,*reg*⟧	`fsubr st,st(2)`
	`fsubr st(5),st`
	`fsubr`
		87
		287
		387
		486
			70–100
			70–100
			to=29–37, *fr*=26–34
			8–20

FSUBRP *reg*,**ST**	`fsubrp st(6),st`	87 287 387 486	75–105 75–105 26–34 8–20

Syntax	Examples	CPU	Clock Cycles

FSUBR *memreal*	`fsubr QWORD PTR [bx]`
	`fsubr shortreal[di]`
	`fsubr longreal`
		87
		
		287
		387
		486
			(*s*=90–120,*s*=95–125)+*EA*
			s=90–120,*l*=95–125
			s=25–33,*l*=29–37
			8–20

FISUBR *memint*	`fisubr int16`
	`fisubr warray[di]`
	`fisubr double`
		87
		
		287
		
		387
		486
			(*w*=103–139,*d*=109–144)+*EA*
			w=103–139,*d*=109–144
			w=72–84,*d*=58–83
			w=20–55,*d*=19–32

FTST Test for Zero

Compares ST with +0.0 and sets the condition of the status word according to the result.

Syntax	Examples	CPU	Clock Cycles
FTST	`ftst`	87 287 387 486	38–48 38–48 28 4

Condition Codes for FTST

C3	C2	C1	C0	Meaning
0	0	?	0	ST is positive
0	0	?	1	ST is negative
1	0	?	0	ST is 0
1	1	?	1	ST is not comparable (NAN or projective infinity)

FUCOM/FUCOMP/FUCOMPP Unordered Compare

80387–80486 Only Compares the specified source to ST and sets the condition codes of the status word according to the result. The instruction subtracts the source operand from ST without changing either operand. Memory operands are not allowed. If no operand is specified or if two pops are specified, ST is compared to ST(1). If one pop is specified with an operand, the given register is compared to ST.

Unlike **FCOM**, **FUCOM** does not cause an invalid-operation exception if one of the operands is NAN. Instead, the condition codes are set to unordered.

Syntax	Examples	CPU	Clock Cycles
FUCOM [[*reg*]]	`fucom st(2)`	87	—
	`fucom`	287	—
		387	24
		486	4
FUCOMP [[*reg*]]	`fucomp st(7)`	87	—
	`fucomp`	287	—
		387	26
		486	4
FUCOMPP	`fucompp`	87	—
		287	—
		387	26
		486	5

Condition Codes for FUCOM

C3	C2	C1	C0	Meaning
0	0	?	0	ST > source
0	0	?	1	ST < source
1	0	?	0	ST = source
1	1	?	1	Unordered

FWAIT Wait

Suspends execution of the processor until the coprocessor is finished executing. This is an alternate mnemonic for the processor **WAIT** instruction.

Syntax	Examples	CPU	Clock Cycles
FWAIT	`fwait`	87	4
		287	3
		387	6
		486	1–3

FXAM Examine

Reports the contents of ST in the condition flags of the status word.

Syntax	Examples	CPU	Clock Cycles
FXAM	`fxam`	87	12–23
		287	12–23
		387	30–38
		486	8

Condition Codes for FXAM

C3	C2	C1	C0	Meaning
0	0	0	0	+ Unnormal*
0	0	0	1	+ NAN
0	0	1	0	– Unnormal*
0	0	1	1	– NAN
0	1	0	0	+ Normal
0	1	0	1	+ Infinity
0	1	1	0	– Normal
0	1	1	1	– Infinity
1	0	0	0	+ 0
1	0	0	1	Empty
1	0	1	0	– 0
1	0	1	1	Empty
1	1	0	0	+ Denormal
1	1	0	1	Empty*
1	1	1	0	– Denormal
1	1	1	1	Empty*

* Not used on the 80387–80486. Unnormals are not supported by the 80387–80486. Also, the 80387–80486 use two codes instead of four to identify empty registers.

FXCH Exchange Registers

Exchanges the specified (destination) register and ST. If no operand is specified, ST and ST(1) are exchanged.

Syntax	Examples	CPU	Clock Cycles
FXCH ⟦*reg*⟧	`fxch st(3)`	87	10–15
	`fxch`	287	10–15
		387	18
		486	4

FXTRACT Extract Exponent and Significand

Extracts the exponent and significand (mantissa) fields of ST. The exponent replaces the value in ST, and then the significand is pushed onto the stack**.**

Syntax	Examples	CPU	Clock Cycles
FXTRACT	`fxtract`	87	27–55
		287	27–55
		387	70–76
		486	16–20

FYL2X Y $\log_2$(X)

Calculates Z = Y $\log_2$(X). X is taken from ST and Y from ST(1). The stack is popped, and the result, Z, replaces Y in ST. X must be in the range $0 < X < \infty$ and Y in the range $-\infty < Y < \infty$.

Syntax	Examples	CPU	Clock Cycles
FYL2X	`fyl2x`	87	900–1100
		287	900–1100
		387	120–538
		486	196–329

FYL2XP1 Y $\log_2$(X+1)

Calculates Z = Y $\log_2$(X + 1). X is taken from ST and Y from ST(1). The stack is popped once, and the result, Z, replaces Y in ST. X must be in the range $0 < |X| < (1 - (\sqrt{2} / 2))$. Y must be in the range $-\infty < Y < \infty$.

Syntax	Examples	CPU	Clock Cycles
FYL2XP1	`fyl2xp1`	87	700–1000
		287	700–1000
		387	257–547
		486	171–326

CHAPTER 6

Macros

Introduction 180
BIOS.INC. 180
CMACROS.INC, CMACROS.NEW 180
MS-DOS.INC 183
MACROS.INC 184
PROLOGUE.INC. 185
WIN.INC 185

Introduction

Each of the INCLUDE files is listed with the names of the macros it contains. Macros listed take the form:

```
<macroname>MACRO[[ <variables[[:=<default value>]], ..>]]
```

Some variables are listed as *name:req*. In these cases, *req* indicates that *macroname* cannot be called without the variable *name* supplied.

For specific information on the macros themselves, see the contents of the commented *.INC file.

BIOS.INC

@Cls MACRO pagenum

@GetCharAtr MACRO pagenum

@GetCsr MACRO pagenum

@GetMode MACRO

@PutChar MACRO chr, atrib, pagenum, loops

@PutCharAtr MACRO chr, atrib, pagenum, loops

@Scroll MACRO distance:REQ, atrib:=<07h>, upcol, uprow, dncol, dnrow

@SetColor MACRO color

@SetCsrPos MACRO column, row, pagenum

@SetCsrSize MACRO first, last

@SetMode MACRO mode

@SetPage MACRO pagenum

@SetPalette MACRO color

CMACROS.INC, CMACROS.NEW

These two include files contain the same macros. Use CMACROS.NEW for programs written in MASM 6.0 and later. Use CMACROS.INC for programs written in MASM 5.1 or earlier, or if you have problems with CMACROS.NEW.

@reverse MACRO list

arg MACRO args

assumes MACRO s,ln

```
callcrt MACRO funcname
cBegin MACRO pname
cEnd MACRO pname
cEpilog MACRO procname, flags, cbParms, cbLocals, reglist, userparms
cProc MACRO pname:REQ, attribs, autoSave
cPrologue MACRO procname, flags, cbParms, cbLocals, reglist, userparms
createSeg MACRO segName, logName, aalign, combine, class, grp
cRet MACRO
defGrp MACRO foo:vararg
errn$ MACRO l,x
errnz MACRO x
externA MACRO names:req, langtype
externB MACRO names:req, langtype
externCP MACRO n,c
externD MACRO names:req, langtype
externDP MACRO n,c
externFP MACRO names:req, langtype
externNP MACRO names:req, langtype
externP MACRO n,c
externQ MACRO names:req, langtype
externT MACRO names:req, langtype
externW MACRO names:req, langtype
farPtr MACRO n,s,o
globalB MACRO name:req, initVal:=<?>, repCount, langType
globalCP MACRO n,i,s,c
globalD MACRO name:req, initVal:=<?>, repCount, langType
globalDP MACRO n,i,s,c
globalQ MACRO name:req, initVal:=<?>, repCount, langType
globalT MACRO name:req, initVal:=<?>, repCount, langType
globalW MACRO name:req, initVal:=<?>, repCount, langType
labelB MACRO names:req,langType
labelCP MACRO n,c
```

```
labelD MACRO names:req,langType
labelDP MACRO n,c
labelFP MACRO names:req,langType
labelNP MACRO names:req,langType
labelP MACRO n,c
labelQ MACRO names:req,langType
labelT MACRO names:req,langType
labelW MACRO names:req,langType
lbl MACRO names:req
localB MACRO name
localCP MACRO n
localD MACRO name
localDP MACRO n
localQ MACRO name
localT MACRO name
localV MACRO name,a
localW MACRO name
logName&_assumes MACRO s
logName&_sbegin MACRO
n MACRO
outif MACRO name:req, defval:=<0>, onmsg, offmsg
parmB MACRO names:req
parmCP MACRO n
parmD MACRO names:req
parmDP MACRO n
parmQ MACRO names:req
parmR MACRO n,r,r2
parmT MACRO names:req
parmW MACRO names:req
regPtr MACRO n,s,o
save MACRO r
sBegin MACRO name:req
```

sEnd MACRO name
setDefLangType MACRO overLangType
staticB MACRO name:req, initVal:=<?>, repCount
staticCP MACRO name:req, i, s
staticD MACRO name:req, initVal:=<?>, repCount
staticDP MACRO name:req, i, s
staticI MACRO name:req, initVal:=<?>, repCount
staticQ MACRO name:req, initVal:=<?>, repCount
staticT MACRO name:req, initVal:=<?>, repCount
staticW MACRO name:req, initVal:=<?>, repCount

MS-DOS.INC

NPVOID TYPEDEF NEAR PTR
FPVOID TYPEDEF FAR PTR
FILE_INFO STRUCT
@ChDir MACRO path:REQ, segmnt
@ChkDrv MACRO drive
@CloseFile MACRO handle:REQ
@DelFile MACRO path:REQ, segmnt
@Exit MACRO return
@FreeBlock MACRO segmnt
@GetBlock MACRO graphs:REQ, retry:=<0>
@GetChar MACRO ech:=<1>, cc:=<1>, clear:=<0>
@GetDate MACRO
@GetDir MACRO buffer:REQ, drive, segmnt
@GetDrv MACRO
@GetDTA MACRO
@GetFileSize MACRO handle:REQ
@GetFirst MACRO path:REQ, atrib, segmnt
@GetInt MACRO interrupt:REQ
@GetNext MACRO

@GetStr MACRO ofset:REQ, terminator, limit, segmnt
@GetTime MACRO
@GetVer MACRO
@MakeFile MACRO path:REQ, atrib:=<0>, segmnt, kind
@MkDir MACRO path:REQ, segmnt
@ModBlock MACRO graphs:REQ, segmnt
@MoveFile MACRO old:REQ, new:REQ, segold, segnew
@MovePtrAbs MACRO handle:REQ, distance
@MovePtrRel MACRO handle:REQ, distance
@OpenFile MACRO path:REQ, access:=<0>, segmnt
@PrtChar MACRO chr:VARARG
@Read MACRO ofset:REQ, bytes:REQ, handle:=<0>, segmnt
@RmDir MACRO path:REQ, segmnt
@SetDate MACRO month:REQ, day:REQ, year:REQ
@SetDrv MACRO drive:REQ
@SetDTA MACRO buffer:REQ, segmnt
@SetInt MACRO interrupt:REQ, vector:REQ, segmnt
@SetTime MACRO hour:REQ, minutes:REQ, seconds:REQ, hundredths:REQ
@ShowChar MACRO chr:VARARG
@ShowStr MACRO ofset:REQ, segmnt
@TSR MACRO paragraphs:REQ, return
@Write MACRO ofset:REQ, bytes:REQ, handle:=<1>, segmnt

MACROS.INC

@ArgCount MACRO arglist:VARARG
@ArgI MACRO index:REQ, arglist:VARARG
@ArgRev MACRO arglist
@PopAll MACRO
@PushAll MACRO
@RestoreRegs MACRO
@SaveRegs MACRO regs:VARARG
echof MACRO format:REQ, args:VARARG
pushc MACRO op

PROLOGUE.INC

cEpilogue MACRO szProcName, flags, cbParams, cbLocals, rgRegs, rgUserParams

cPrologue MACRO szProcName, flags, cbParams, cbLocals, rgRegs, rgUserParams

WIN.INC

The include file WIN.INC is WINDOWS.H processed by H2INC, and slightly modified to reduce unnecessary warnings.

C H A P T E R 7

Tables

ASCII Chart 188
Key Codes 190
MS-DOS Program Segment Prefix (PSP) 192
Color Display Attributes 193
Hexadecimal-Binary-Decimal Conversion 194

ASCII Codes

Key Codes

MS-DOS Program Segment Prefix (PSP)

1 Opcode for INT 20h instruction (CDh 20h)

2 Segment of first allocatable address following the program (used for memory allocation)

3 Reserved or used by MS-DOS

4 Opcode for far call to MS-DOS function dispatcher

5 Vector for terminate routine

6 Vector for CTRL+C handler routine

7 Vector for error handler routine

8 Segment address of program's environment block

9 Opcode for MS-DOS INT 21h and far return (you can do a far call to this address to execute MS-DOS calls)

10 First command-line argument (formatted as uppercase 11-character filename)

11 Second command-line argument (formatted as uppercase 11-character filename)

12 Number of bytes in command-line argument

13 Unformatted command line and/or default Disk Transfer Area (DTA)

Color Display Attributes

Background Bits				Num	Color	Foreground Bits*				Num	Color
F	R	G	B			I	R	G	B		
0	0	0	0	0	Black	0	0	0	0	0	Black
0	0	0	1	1	Blue	0	0	0	1	1	Blue
0	0	1	0	2	Green	0	0	1	0	2	Green
0	0	1	1	3	Cyan	0	0	1	1	3	Cyan
0	1	0	0	4	Red	0	1	0	0	4	Red
0	1	0	1	5	Magenta	0	1	0	1	5	Magenta
0	1	1	0	6	Brown	0	1	1	0	6	Brown
0	1	1	1	7	White	0	1	1	1	7	White
1	0	0	0	8	Black blink	1	0	0	0	8	Dark gray
1	0	0	1	9	Blue blink	1	0	0	1	9	Light Blue
1	0	1	0	A	Green blink	1	0	1	0	A	Light green
1	0	1	1	B	Cyan blink	1	0	1	1	B	Light cyan
1	1	0	0	C	Red blink	1	1	0	0	C	Light red
1	1	0	1	D	Magenta blink	1	1	0	1	D	Light Magenta
1	1	1	0	E	Brown blink	1	1	1	0	E	Yellow
1	1	1	1	F	White blink	1	1	1	1	F	Bright White

F Flashing bit G Green bit I Intensity bit
R Red bit B Blue bit

* On monochrome monitors, the blue bit is set and the red and green bits are cleared (001) for underline; all color bits are set (111) for normal text.

Hexadecimal-Binary-Decimal Conversion

Hex Number	Binary Number	Decimal Digit 000X	Decimal Digit 00X0	Decimal Digit 0X00	Decimal Digit X000
0	0000	0	0	0	0
1	0001	1	16	256	4,096
2	0010	2	32	512	8,192
3	0011	3	48	768	12,288
4	0100	4	64	1,024	16,384
5	0101	5	80	1,280	20,480
6	0110	6	96	1,536	24,576
7	0111	7	112	1,792	28,672
8	1000	8	128	2,048	32,768
9	1001	9	144	2,304	36,864
A	1010	10	160	2,560	40,960
B	1011	11	176	2,816	45,056
C	1100	12	192	3,072	49,152
D	1101	13	208	3,328	53,248
E	1110	14	224	3,584	57,344
F	1111	15	240	3,840	61,440

Programmer's Guide

Microsoft® MASM

Assembly-Language Development System
Version 6.1

For MS-DOS® and Windows™ Operating Systems

Microsoft Corporation

Information in this document is subject to change without notice. Companies, names, and data used in examples herein are fictitious unless otherwise noted. No part of this document maybe reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Microsoft Corporation.

©1992 Microsoft Corporation. All rights reserved.

Microsoft, MS, MS-DOS, XENIX, CodeView, and QuickC are registered trademarks and Microsoft QuickBasic, QuickPascal, Windows and Windows NT are trademarks of Microsoft Corporation in the USA and other countries.

U.S. Patent No. 4,955,066

Hercules is a registered trademark of Hercules Computer Technology.
IBM, PS/2, and OS/2 are registered trademarks of International Business Machines Corporation.
Intel is a registered trademark of Intel Corporation.
NEC and V25 are registered trademarks and V35 is a trademark of NEC Corporation.

Document No. DB35747-1292
Printed in the United States of America.

Contents

Introduction. xiii
New and Extended Features in MASM 6.1 xiii
MASM Features New Since Version 5.1 xiv
MASM Features New Since Version 6.0 xv
ML and MASM Command Lines xvi
Compatibility with Earlier Versions of MASM xvi
A Word About Instruction Timings xvii
Books for Further Reading xviii
Document Conventions xix
Getting Assistance and Reporting Problems xx

Chapter 1 Understanding Global Concepts 1
The Processing Environment 1
8086-Based Processors 2
Operating Systems 4
Segmented Architecture 5
Segment Protection 6
Segmented Addressing. 7
Segment Arithmetic 7
Language Components of MASM 8
Reserved Words 8
Identifiers 9
Predefined Symbols 10
Integer Constants and Constant Expressions 11
Operators 13
Data Types. 14
Registers 16
Statements 21
The Assembly Process 22
Generating and Running Executable Programs 23
Using the OPTION Directive 24
Conditional Directives 28

Chapter 2 Organizing Segments 31
Physical Memory Segments 32
Logical Segments. 32
Using Simplified Segment Directives 33

Defining Basic Attributes with .MODEL 34
Specifying a Processor and Coprocessor 38
Creating a Stack. 38
Creating Data Segments 39
Creating Code Segments. 40
Starting and Ending Code with .STARTUP and .EXIT 41
Using Full Segment Definitions 44
Defining Segments with the SEGMENT Directive. 44
Controlling the Segment Order 47
Setting the ASSUME Directive for Segment Registers 49
Defining Segment Groups 51

Chapter 3 Using Addresses and Pointers 53
Programming Segmented Addresses 53
Initializing Default Segment Registers. 53
Near and Far Addresses 57
Operands 60
Register Operands 61
Immediate Operands 61
Direct Memory Operands 62
Indirect Memory Operands 64
The Program Stack. 71
Saving Operands on the Stack. 71
Saving Flags on the Stack 73
Saving Registers on the Stack (80186–80486 Only). 74
Accessing Data with Pointers and Addresses 74
Defining Pointer Types with TYPEDEF 75
Defining Register Types with ASSUME. 77
Basic Pointer and Address Operations 78

Chapter 4 Defining and Using Simple Data Types 85
Declaring Integer Variables 85
Allocating Memory for Integer Variables 85
Data Initialization 87
Working with Simple Variables 88
Copying Data. 89
Adding and Subtracting Integers 92
Multiplying and Dividing Integers. 95
Manipulating Numbers at the Bit Level. 98
Logical Instructions 99
Shifting and Rotating Bits 100

Multiplying and Dividing with Shift Instructions 102

Chapter 5 Defining and Using Complex Data Types. 105
Arrays and Strings 105
Declaring and Referencing Arrays. 105
Declaring and Initializing Strings 108
Processing Strings 110
Structures and Unions 117
Declaring Structure and Union Types 118
Defining Structure and Union Variables. 121
Referencing Structures, Unions, and Fields 126
Nested Structures and Unions 128
Records 129
Declaring Record Types 130
Defining Record Variables 131
Record Operators 133

Chapter 6 Using Floating-Point and Binary Coded Decimal Numbers 135
Using Floating-Point Numbers 136
Declaring Floating-Point Variables and Constants. 136
Storing Numbers in Floating-Point Format. 138
Using a Math Coprocessor 139
Coprocessor Architecture. 140
Instruction and Operand Formats 141
Coordinating Memory Access. 145
Using Coprocessor Instructions 146
Using An Emulator Library. 155
Using Binary Coded Decimal Numbers 156
Defining BCD Constants and Variables 157
BCD Calculations on a Coprocessor 157
BCD Calculations on the Main Processor 158

Chapter 7 Controlling Program Flow. 161
Jumps. 161
Unconditional Jumps 162
Conditional Jumps. 164
Loops. 172
Loop-Generating Directives 173
Writing Loop Conditions 178
Procedures 180
Defining Procedures 180

Passing Arguments on the Stack 182
Declaring Parameters with the PROC Directive 184
Using Local Variables. 188
Creating Local Variables Automatically 190
Declaring Procedure Prototypes 193
Calling Procedures with INVOKE 194
Generating Prologue and Epilogue Code. 198
MS-DOS Interrupts 204
Calling MS-DOS and ROM-BIOS Interrupts 204
Replacing an Interrupt Routine 206

Chapter 8 Sharing Data and Procedures Among Modules and Libraries 211
Selecting Data-Sharing Methods. 211
Sharing Symbols with Include Files 212
Organizing Modules 212
Declaring Symbols Public and External 214
Positioning External Declarations. 228
Using Alternatives to Include Files 219
PUBLIC and EXTERN 220
Other Alternatives 221
Developing Libraries. 221
Associating Libraries with Modules 222
Using EXTERN with Library Routines 223

Chapter 9 Using Macros 225
Text Macros. 226
Macro Procedures 226
Creating Macro Procedures. 227
Passing Arguments to Macros 228
Specifying Required and Default Parameters 229
Defining Local Symbols in Macros 232
Assembly-Time Variables and Macro Operators 233
Text Delimiters and the Literal-Character Operator 234
Expansion Operator 235
Substitution Operator 237
Defining Repeat Blocks with Loop Directives 239
REPEAT Loops. 240
WHILE Loops. 241
FOR Loops and Variable-Length Parameters 242
FORC Loops. 244
String Directives and Predefined Functions 245

Returning Values with Macro Functions 248
Returning Values with EXITM 248
Using Macro Functions with Variable-Length Parameter Lists 249
Expansion Operator in Macro Functions 251
Advanced Macro Techniques 251
Defining Macros within Macros 251
Testing for Argument Type and Environment 252
Using Recursive Macros 255

Chapter 10 Writing a Dynamic-Link Library For Windows 257
Overview of DLLs 257
Loading a DLL 258
Building a DLL 260
DLL Code 261
DLL Data 265
DLL Stack 265
DLL Extension Names 266
Summary 266
Example of a DLL: SYSINFO 267
Entry Routine for SYSINFO 268
Expanding SYSINFO 270

Chapter 11 Writing Memory-Resident Software 273
Terminate-and-Stay-Resident Programs 273
Structure of a TSR 274
Passive TSRs 274
Active TSRs 275
Interrupt Handlers in Active TSRs 275
Auditing Hardware Events for TSR Requests 275
Monitoring System Status 277
Determining Whether to Invoke the TSR 279
Example of a Simple TSR: ALARM 279
Using MS-DOS in Active TSRs 285
Understanding MS-DOS Stacks 285
Determining MS-DOS Activity 285
Interrupting MS-DOS Functions 286
Monitoring the Critical Error Flag 287
Preventing Interference 288
Trapping Errors 288
Preserving an Existing Condition 289
Preserving Existing Data 290

Communicating Through the Multiplex Interrupt 290
The Multiplex Handler 291
Using the Multiplex Interrupt Under MS-DOS Version 2.x. 292
Deinstalling a TSR 292
Example of an Advanced TSR: SNAP 293
Building SNAP.EXE 294
Outline of SNAP 295

Chapter 12 Mixed-Language Programming 307
Naming and Calling Conventions 308
Naming Conventions 309
The C Calling Convention. 309
The Pascal Calling Convention 310
The STDCALL and SYSCALL Calling Conventions. 311
Writing an Assembly Procedure For a Mixed-Language Program 312
The MASM/High-Level–Language Interface. 313
The C/MASM Interface 315
The C++/MASM Interface 322
The FORTRAN/MASM Interface 323
The Basic/MASM Interface 328

Chapter 13 Writing 32-Bit Applications 335
32-Bit Memory Addressing 335
MASM Directives for 32-Bit Programming. 336
Sample Program. 337

Appendixes

Appendix A Differences Between MASM 6.1 and 5.1. 341
New Features of Version 6.1 342
The Assembler, Environment, and Utilities. 342
Segment Management 343
Data Types 344
Procedures, Loops, and Jumps 347
Simplifying Multiple-Module Projects 348
Expanded State Control 349
New Processor Instructions. 350
Renamed Directives 350
Macro Enhancements. 351
MASM 6.1 Programming Practices 352
Compatibility Between MASM 5.1 and 6.1. 352

Rewriting Code for Compatibility 353
Using the OPTION Directive 361
Changes to Instruction Encodings 377

Appendix B BNF Grammar 379

Appendix Generating and Reading Assembly Listings. 397
Generating Listing Files 397
Precedence of Command-Line Options and Listing Directives. 399
Reading the Listing File 399
Generated Code 399
Error Messages. 400
Symbols and Abbreviations 400
Reading Tables in a Listing File 404

Appendix D MASM Reserved Words 407
Operands and Symbols. 407
Special Operands for the 80386/486 409
Predefined Symbols 409
Registers. 409
Operators and Directives 410
Processor Instructions 412
8086/8088 Processor Instructions. 412
80186 Processor Instructions 413
80286 Processor Instructions 413
80286 and 80386 Privileged-Mode Instructions 413
80386 Processor Instructions 413
80486 Processor Instructions 414
Instruction Prefixes 414
Coprocessor Instructions 414
8087 Coprocessor Instructions 414
80287 Privileged-Mode Instruction 415
80387 Instructions. 415

Appendix E Default Segment Names 417

Glossary 421

Index. 435

Figures and Tables

Figures

1.1 Segment Allocation. 6
1.2 Calculating Physical Addresses 8
1.3 Registers for 8088-80286 Processors 17
1.4 Extended Registers for the 80386/486 Processors 18
1.5 Flags for 8088-80486 Processors. 20
3.1 Stack Status Before and After Pushes and Pops 72
4.1 Integer Formats 87
4.2 Shifts and Rotates 101
6.1 Encoding for Real Numbers in IEEE Format 138
6.2 Coprocessor Data Registers. 140
6.3 Status of the Register Stack. 142
6.4 Status of the Register Stack and Memory Locations 143
6.5 Status of the Previously Initialized Register Stack 144
6.6 Status of the Already Initialized Register Stack 144
6.7 Status of the Register Stack: Main Memory and Coprocessor. 148
6.8 Coprocessor Control Registers. 154
6.9 Coprocessor and Processor Control Flags. 155
7.1 Program Arguments on the Stack 183
7.2 Local Variables on the Stack 190
7.3 Operation of Interrupts 206
8.1 Using EXTERNDEF for Variables. 215
8.2 Using PROTO and INVOKE 217
8.3 Using PUBLIC and EXTERN. 221
11.1 Time Line of Interaction Between Interrupt Handlers
for a Typical TSR. 278
11.2 Flowchart for SNAP.EXE: Installation Phase 296
11.3 Flowchart for SNAP.EXE Resident Phase 297
11.4 Flowchart for SNAP.EXE Deinstallation Phase. 298
12.1 C String Format 316
12.2 C Stack Frame. 320
12.3 FORTRAN String Frame 324
12.4 FORTRAN Stack Frame. 327
12.5 Basic String Descriptor Format 330
12.6 Basic Stack Frame 333
B.1 BNF Definition of the TYPEDEF Directive. 380

Tables

1.1 8086 Family of Processors 2
1.2 The MS-DOS and Windows Operating Systems Compared 4
1.3 Operator Precedence 14
2.1 Attributes of Memory Models 35
3.1 Indirect Addressing with 16-Bit Registers 68
4.1 Division Operations 97
5.1 Requirements for String Instructions 112
6.1 Ranges of Floating-Point Variables 136
6.2 Coprocessor Operand Formats 141
6.3 Control-Flag Settings After Comparison or Test 151
7.1 Conditional Jumps Based on Comparisons of Two Values 167
9.1 MASM Macro Operators 234
11.1 MS-DOS Internal Stacks 286
12.1 Naming and Calling Conventions 309
12.2 Register Conventions for Simple Return Values 317
A.1 Requirements for String Instructions. 353
C.1 Options for Generating or Modifying Listing Files 398
C.2 Symbols and Abbreviations in Listings 400
C.3 Symbols in Timing Column 401

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 6.1. This book documents enhanced features of the language and the programming environment for MASM 6.1.

This *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" in this introduction.

This book teaches you how to write efficient code with the new and advanced features of MASM. *Getting Started* explains how to set up MASM 6.1. *Environment and Tools* introduces the integrated development environment called the Programmer's WorkBench (PWB). It also includes a detailed reference to Microsoft tools and utilities such as Microsoft® CodeView®, LINK, and NMAKE. 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.

New and Extended Features in MASM 6.1

MASM 6.1 continues the break with tradition established by version 6.0. It incorporates conveniences of high-level languages while offering all the traditional advantages of assembly-language programming.

For example, MASM 6.1 includes the Programmer's WorkBench, which provides the same integrated software development environment enjoyed by programmers of Microsoft high-level languages such as C and Basic. From within PWB you can edit, build, debug, or run a program. 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.

MASM Features New Since Version 5.1

MASM 6.1 includes several features designed to make programming more efficient and productive. The following list briefly describes how MASM 6.1 improves on the language features of the popular version 5.1.

- MASM 6.1 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 Simple Data Types."
- The syntax for defining and using structures and records has been enhanced since version 5.1. 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 Simple Data Types."
- 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 since version 5.1. 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 makes 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 6.1 than in version 5.1. 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."

MASM 6.1 has other improved capabilities, such as:

- The **.STARTUP** and **.EXIT** directives automatically generate appropriate startup and exit code for your assembly-language programs. See Chapter 2, "Organizing Segments."
- MASM 6.1 supports flat memory model, available with the new Microsoft® Windows NT ™ operating system. Flat model allows segments as large as 4 gigabytes instead of 64K (kilobytes). Offsets are 32 bits instead of 16 bits. See Chapter 2, "Organizing Segments."
- The program H2INC.EXE converts C include files to MASM include files and translates data structures and declarations. See Chapter 20 in *Environment and Tools*.
- MASM 6.1 provides a library of assembly routines that let you create a terminate-and-stay-resident program (TSR) in a high-level language.

MASM 6.1 includes many other minor new features as well as extensive support for features of earlier versions of MASM. For a complete list of enhancements, refer to Appendix A, "Differences between MASM 6.1 and 5.1." The cross-references in Appendix A guide you to the chapters where the new features are described in detail.

MASM Features New Since Version 6.0

MASM 6.1 offers several new features:

- ML now runs in 32-bit protected mode under MS-DOS, giving it direct access to extended memory for assembling very large source files.
- A collection of tools lets you write a dynamic-link library (DLL) for the Microsoft® Windows ™ operating system without the Windows Software Development Kit. The LIBW.LIB library provides access to all functions in the Windows application programming interface (API), so your DLL can display menus, dialog boxes, and scroll bars. Chapter 10, "Writing a Dynamic-Link Library for Windows," shows you how.

- Program listings now show instruction timings. The number of required processor cycles appears adjacent to each instruction in the listing, based on the selected processor. For an example listing and instructions on how to use this feature, see Appendix C, "Generating and Reading Assembly Listings."
- All utilities have been updated for version 6.1. Documentation is clearer and better arranged, with a new *Environment and Tools* reference book.
- Version 6.1 generates debugging information for CodeView version 4.0 and later.
- MASM 6.1 provides even greater compatibility with version 5.1 than does MASM 6.0. Many programs written with version 5.1 will assemble unchanged under MASM 6.1.

ML and MASM Command Lines

MASM 6.1 provides an updated version of the command-line driver, ML, introduced in version 6.0. ML is more powerful and flexible than the MASM driver of version 5.1. ML assembles and links with one command. It recognizes all the old MASM driver command syntax, however, to support existing batch files and makefiles that use MASM command lines.

Note The name MASM has traditionally referred to the Microsoft Macro Assembler. It is used in that context throughout this book. However, MASM also refers to MASM.EXE, which has been replaced by ML.EXE. In MASM 6.1, MASM.EXE 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

MASM 6.1 is fully compatible with version 6.0 and, in many cases, with version 5.1. Code written for MASM 5.1 will often assemble correctly without modification under MASM 6.1. However, MASM 6.1 provides the **OPTION** directive to let 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.

For information about obsolete features that will not assemble correctly under MASM 6.1, see Appendix A, "Differences Between MASM 6.1 and 5.1." The appendix also explains how to update code to use the new features.

A Word About Instruction Timings

As an assembly-language programmer, whether novice or expert, you are probably interested in producing lightning-fast code. After all, one of the main reasons to program in assembly is to take advantage of its ability to streamline execution speeds to the limit of the processor. This book will help you write efficient and fast programs.

When discussing the speed of individual instructions, the chapters in this book often speak of "timing," which is the number of processor cycles required to carry out an instruction. The *Reference* lists instruction timings for processors in the 8086 family. It is tempting to use timing as the only criterion when judging an instruction's actual execution speed, but the world within the processor is not so simple.

The clock for instruction timing does not begin ticking until the processor has read and begins to execute an instruction. When you read about instruction timings (in this book or any other), keep in mind that other factors also influence the real speed of an instruction: the instruction's size, whether it resides in cache memory, whether it accesses memory, its position in the processor's prefetch queue, and the processor type. These factors make it impossible to say precisely how fast an instruction executes. Accept the references to timing in this book as guidelines, but use these simple rules to write fast code:

- Whenever possible, use registers rather than constant values, and constant values rather than memory.
- Minimize changes in program flow.
- Smaller is often better. For example, the instructions

  ```
  dec  bx
  sub  bx, 1
  ```

  accomplish the same thing and have the same timings on 80386/486 processors. But the first instruction is 3 bytes smaller than the second, and so may reach the processor faster.
- When possible, use the string instructions described in Chapter 5, "Defining and Using Complex Data Types."

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. Out of print.

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

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

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

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

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

Books About MS-DOS and BIOS

"Terminate-and-Stay-Resident Utilities." *MS-DOS Encyclopedia.* Redmond, WA: Microsoft Press, 1989.

Duncan, Ray. *Advanced MS-DOS Programming: The Microsoft Guide for Assembly Language and C Programmers*. 2d ed. Redmond, WA: Microsoft Press, 1988.

Duncan, Ray. *Extending DOS: Programmer's Guide to Protected-Mode DOS*. Redding, MA: Addison-Wesley. 1991.

Jourdain, Robert. *Programmer's Problem Solver for the IBM PC, XT and AT*. New York: Brady Communications, 1985. Out of print.

Microsoft MS-DOS Programmer's Reference. Redmond, WA: Microsoft Press, 1991.

Norton, Peter and Richard Wilton. *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: Maximum Video Performance from the EGA, VGA, HGC, and MCGA*. Redmond, WA: Microsoft Press, 1987. Out of print.

Books and Articles About Windows

Kauler, Barry. *Windows Assembly Language & Systems Programming: Object-Oriented & Systems Programming in Assembly Language for Windows 3.0 and 3.1*. New York, NY: Prentice Hall, 1993.

Klein, Mike. *Windows Programmer's Guide to DLLs & Memory Management*. Carmel, IN: Sams, 1992.

Petzold, Charles. *Programming Windows*. 3d ed. Redmond, WA: Microsoft Press, 1992.

Petzold, Charles. "Environments." *PC Magazine*. New York, NY: Ziff-Davis Publishing Company, June 1990–1992.

Programmer's Reference. 4 vols. Microsoft Windows Software Development Kit (SDK). Redmond, WA: Microsoft Press, 1992.

Books About Other Topics

Nelson, Ross P. *The 80386/80486 Programming Guide*. 2d ed. Redmond, WA: Microsoft Press, 1991.

Startz, Richard. *8087/80287/80387 for the IBM PC and Compatibles: Applications and Programming with Intel's Math Coprocessors*. Bowie, MD: Robert J. Brady Co., 1988. Out of print.

Document Conventions

The following document conventions are used throughout this manual:

Example of Convention	Description
SAMPLE2.ASM	Uppercase letters indicate filenames, 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.
placeholder	Italic letters indicate placeholders for information you must supply, such as a filename. Italics are used occasionally 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 source of the problem:

- The version of MS-DOS or Windows you run.
- Your system configuration: the type of machine you use, its total memory, and its total free memory at assembler execution time, as well as any other information you think might be useful.
- The command line you used for the assembler, linker, or other MASM tool that was running when the problem occurred.
- Any object files or libraries you linked with if the problem occurred at link time.

If your program is very large, reduce it to the smallest possible program that still produces the problem.

Note the circumstances of the error and notify Microsoft Corporation by following the instructions in the section "Microsoft Support Services" in the introduction to *Environment and Tools*. If you have comments or suggestions regarding any of the books accompanying this product, please indicate them on the Document Feedback page at the back of this book and send it to Microsoft.

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

CHAPTER 1

Understanding Global Concepts

With the development of the Microsoft Macro Assembler (MASM) version 6.1, 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 a review of 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 this chapter reviews available processors and operating systems and how they work together. The section also discusses segmented architecture and how it affects a protected-mode operating environment such as Windows.

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 remainder of this book was written with the assumption 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.

The Processing Environment

The processing environment for MASM 6.1 includes the processor on which your programs run, the operating system your programs 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.

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 support the same segmented architecture. Knowing the differences between the various 8086-based processors can help you select the appropriate 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 using 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 following.

Table 1.1 8086 Family of Processors

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

Processor Modes

Real mode allows only one process to run at a time. The mode gets its name from the fact that addresses in real mode always correspond to real locations in memory. The MS-DOS operating system runs in real mode.

Windows 3.1 operates only in protected mode, but runs MS-DOS programs in real mode or in a simulation of real mode called virtual-86 mode. In protected mode, more than one process can be active at any one time. The operating system protects memory belonging to one process from access by another process; hence the name protected mode.

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. For more information, see "Operating Systems," following.

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 protected mode services, allowing the operating system to run multiple processes at the same time. The 80286 is the minimum for running Windows 3.1 and 16-bit versions of OS/2®.

80386

Unlike its predecessors, the 80386 processor can handle both 16-bit and 32-bit data. It supports the entire instruction set of the 80286, and adds several new instructions as well. Software written for the 80286 runs unchanged on the 80386, but is faster because the chip operates at higher speeds.

The 80386 implements many new hardware-level features, including paged memory, multiple virtual 8086 processes, addressing of up to 4 gigabytes of memory, and specialized debugging registers. Thirty-two–bit operating systems such as Windows NT and OS/2 2.0 can run only on an 80386 or higher processor.

80486

The 80486 processor is an enhanced version of the 80386, with instruction “pipelining” that executes many instructions two to three times faster. The chip incorporates both a math coprocessor and an 8K (kilobyte) memory cache. (The math coprocessor is disabled on a variation of the chip called the 80486SX.) The 80486 includes 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.

Operating Systems

With MASM, you can create programs that run under MS-DOS, Windows, or Windows NT—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 Help for PWB.

MS-DOS and Windows 3.1 provide different processing modes. MS-DOS runs in the single-process real mode. Windows 3.1 operates in protected mode, allowing multiple processes to run simultaneously.

Although Windows requires another operating system for loading and file services, it provides many functions normally associated with an operating system. When an application requests an MS-DOS service, Windows often provides the service without invoking MS-DOS. For consistency, this book refers to Windows as an operating system.

MS-DOS and Windows (in protected mode) differ primarily in system access methods, size of addressable memory, and segment selection. Table 1.2 summarizes these differences.

Table 1.2 The MS-DOS and Windows Operating Systems Compared

Operating System	System Access	Available Active Processes	Addressable Memory	Contents of Segment Register	Word Length
MS-DOS and Windows real mode	Direct to hardware and OS call	One	1 megabyte	Actual address	16 bits
Windows virtual-86 mode	Operating system call	Multiple	1 megabyte	Segment selectors	16 bits
Windows protected mode	Operating system call	Multiple	16 megabytes	Segment selectors	16 bits
Windows NT	Operating system call	Multiple	512 megabytes	Segment selectors	32 bits

MS-DOS

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

Windows

As you can see in Table 1.2, protected mode allows for much larger data structures than real mode, since addressable memory extends to 16 megabytes. In protected mode, segment registers contain selector values rather than actual segment addresses. 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 in protected mode.

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 affect interrupts (such as **CLI**, **STI**, **IN**, and **OUT**) are available at privilege levels normally used only by systems programmers.

Windows protected mode provides each application with up to 16 megabytes of "virtual memory," even on computers that have less physical memory. The term virtual memory refers to the operating system's ability to use a swap area on the hard disk as an extension of real memory. When a Windows application requires more memory than is available, Windows writes sections of occupied memory to the swap area, thus freeing those sections for other use. It then provides the memory to the application that made the memory request. When the owner of the swapped data regains control, Windows restores the data from disk to memory, swapping out other memory if required.

Windows NT

Windows NT uses the so-called "flat model" of 80386/486 processors. This model places the processor's entire address space within one 32-bit segment. The section "Defining Basic Attributes with .MODEL" in Chapter 2 explains how to use the flat model. In flat model, your program can (in theory) access up to 4 gigabytes of virtual memory. Since code, data, and stack reside in the same segment, each segment register can hold the same value, which need never change.

Segmented Architecture

The 8086 family of processors employs a segmented architecture—that is, each address is represented as a segment and an offset. Segmented addresses affect many aspects of assembly-language programming, especially addresses and pointers.

Segmented architecture was originally designed to enable a 16-bit processor to access an address space larger than 64K. (The section "Segmented Addressing," later in this chapter, explains how the processor uses both the segment and offset to create addresses larger than 64K.) MS-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. Windows 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 compatible with the older 16-bit processors, but allow flat model 32-bit offset values up to 4 gigabytes. Offset values of this magnitude remove the memory limitations of segmented architecture. The Windows NT operating system uses 32-bit addressing.

Segment Protection

Segmented architecture is an important part of the Windows memory-protection scheme. In a "multitasking" operating system in which numerous programs can run simultaneously, programs cannot access the code and data of another process without permission.

In MS-DOS, the data and code segments are usually allocated adjacent to each other, as shown in Figure 1.1. In Windows, the data and code segments can be anywhere in memory. The programmer knows nothing about, and has no control over, their location. The operating system can even move the segments to a new memory location or to disk while the program is running.

Figure 1.1 Segment Allocation

Segment protection makes software development easier and more reliable in Windows than in MS-DOS, because Windows immediately detects illegal

memory accesses. The operating system intercepts illegal memory accesses, terminates the program, and displays a message. This makes it easier for you to track down and fix the bug.

Because it runs in real mode, MS-DOS contains no mechanism for detecting an improper memory access. A program that overwrites data not belonging to it may continue to run and even terminate correctly. The error may not surface until later, when MS-DOS or another program reads the corrupted memory.

Segmented Addressing

Segmented addressing refers to the internal mechanism that combines a segment value and an offset value to form a complete memory address. The two parts of an address are represented as

segment:*offset*

The *segment* portion always consists of 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 1 megabyte of memory, the BIOS and operating system on International Standard Architecture (IBM PC/AT and compatible) computers use part of this memory, leaving the remainder available for programs.

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, in which addresses do not correspond to a known segment and offset.

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, called the “physical address,” to access an actual location in the 1-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 addresses 0000:F800, 0F00:0800, and 0F80:0000 all refer to the same physical address 0F800.

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. For detailed information, see Help or the *Reference*.

Reserved Words

A reserved word has a special meaning fixed by the language. You can use it only under certain conditions. Reserved words in MASM 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 “Predefined Symbols,” later in this chapter).

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 before the first reference to **STR**, **MASK**, or **NAME**:

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

The section “Using the OPTION Directive,” later in this chapter, discusses the **OPTION** directive. Appendix D provides a complete list of MASM reserved words.

With the /Zm command-line option or **OPTION M510** in effect, MASM does not reserve any operators or instructions that do not apply to the current CPU mode. For example, you can use the symbol `ENTER` when assembling under the default CPU mode but not under **.286** mode, since the 80186/486 processors recognize `ENTER` as an instruction. The **USE32**, **FLAT**, **FAR32**, and **NEAR32** segment types and the 80386/486 register names are not keywords with processors other than the 80386/486.

Identifiers

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 6.1 predefines some special symbols starting with @ (see “Predefined Symbols,” following).

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. (For more information about visibility and scope, see "Sharing Symbols with Include Files" in Chapter 8.)

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 according to the kinds of information they provide. Case is important only if the /Cp option is used. (For additional details, see Help on ML command-line options.)

The predefined symbols for segment information include:

Symbol	Description
@code	Returns the name of the code segment.
@CodeSize	Returns an integer representing the default code distance.
@CurSeg	Returns the name of the current segment.
@data	Expands to DGROUP.
@DataSize	Returns an integer representing the default data distance.
@fardata	Returns the name of the segment defined by the **.FARDATA** directive.
@fardata?	Returns 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 "Creating a Stack" in Chapter 2.)
@WordSize	Provides the size attribute of the current segment.

The predefined symbols for environment information include:

Symbol	Description
@Cpu	Contains a bit mask specifying the processor mode.
@Environ	Returns values of environment variables during assembly.
@Interface	Contains information about the language parameters.
@Version	Represents the text equivalent of the MASM version number. In MASM 6.1, this expands to 610.

The predefined symbols for date and time information include:

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

The predefined symbols for file information include:

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.

The predefined symbols for macro string manipulation include:

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.

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     bx, 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 the default radix is not hexadecimal)
- `o` or `q` for octal
- `t` for decimal (or `d` if the default radix is not hexadecimal)
- `h` for hexadecimal

Radix specifiers can be either uppercase or lowercase letters; sample code in this book is in lowercase. If you do not specify a radix, the assembler interprets the integer according to the current radix. The default radix is decimal, but you can change the default 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, MASM interprets `ABCh` as an identifier. The hexadecimal digits A through F can be either uppercase or lowercase letters. Sample code in this book is in uppercase letters.

Constant expressions contain integer constants and (optionally) operators such as shift, logical, and arithmetic operators. The assembler evaluates constant expressions 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. Symbolic constants are used to assign names to constant values. You can use a symbol with an assigned value in place of an immediate operand. For example, instead of referring in your code to keyboard scan codes with numbers such as 30 or 48, you can create more recognizable symbols:

```
SCAN_A  EQU    30
SCAN_B  EQU    48
```

then use the appropriate symbol in your program rather than the number. Using symbolic constants instead of undescriptive numbers makes your code more readable and easier to maintain. The assembler does not allocate data storage when you use either **EQU** or **=**. It simply replaces each occurrence of the symbol with the value of the expression.

The directives **EQU** and **=** have slightly different purposes. Integers defined with the **=** directive can be redefined with another value in your source code, but those defined with **EQU** cannot. Once you've defined a symbolic constant with the **EQU** directive, attempting to redefine it generates an error. The syntax is:

symbol **EQU** *expression*

The *symbol* is a unique name of your choice, except for words reserved by MASM. 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. Symbolic constants let you change a constant value used throughout your source code by merely altering *expression* in the definition. This removes the potential for error and saves you the inconvenience of having to find and replace each occurrence of the constant in your program.

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 6.1 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**. However, you can repeat the same directive in your source code as often as you wish. You can place the same directive in every include file, for example.

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` illustrates a valid use of the plus operator (+). It tells the assembler to add 2 to the constant value `Amount`, which might be a value or an address. Contrast this operation, which occurs at assembly time, with the processor's **ADD** instruction. **ADD** tells the processor at run time to add two numbers and store the result.

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

- Operations in parentheses are performed before 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.

Table 1.3 lists the order of precedence for all operators. Operators on the same line have equal precedence.

Table 1.3 Operator Precedence

Precedence	Operators
1	**( ), [ ]**
2	**LENGTH, SIZE, WIDTH, MASK, LENGTHOF, SIZEOF**
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**

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 6.1 are **BYTE**, **SBYTE**, **WORD**, **SWORD**, **DWORD**, **SDWORD**, **FWORD**, **QWORD**, and **TBYTE**. These types define integers and binary coded decimals (BCDs), as discussed in Chapter 6. The signed data types **SBYTE**, **SWORD**, and **SDWORD** work in conjunction with directives such as **INVOKE** (for calling procedures) and **.IF** (introduced in Chapter 7). The **REAL4**, **REAL8**, and **REAL10** directives define floating-point types. (See Chapter 6.)

Versions of MASM prior to 6.0 had separate directives for types and initializers. For example, **BYTE** is a type and **DB** is the corresponding initializer. The distinction does not apply in MASM 6.1. You can use any type (intrinsic or user-defined) as an initializer.

MASM does not have specific types for arrays and strings. However, you can treat a sequence of data units as arrays, and character or byte sequences as strings. (See "Arrays and Strings" in Chapter 5.)

Types can also have attributes such as *langtype* and *distance* (**NEAR** and **FAR**). For information on these attributes, see "Declaring Parameters with the PROC Directive" in Chapter 7.

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.")

You can define new types, including pointer types, with the **TYPEDEF** directive. **TYPEDEF** assigns a *qualifiedtype* (explained in the following) to a *typename* of your choice. This lets you build new types with descriptive names of your choosing, making your programs more readable. For example, the following statement makes the symbol `CHAR` a synonym for the intrinsic type **BYTE**:

```
CHAR    TYPEDEF BYTE
```

The *qualifiedtype* is any type or pointer to a type of the form:

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

where *distance* is **NEAR, FAR**, or any distance modifier. (For more information on *distance*, see "Declaring Parameters with the PROC Directive" in Chapter 7.)

The *qualifiedtype* can also be any type previously defined with **TYPEDEF**. For example, if you use **TYPEDEF** to create an alias for **BYTE**—say, `CHAR` as in the preceding example—you can use `CHAR` as a *qualifiedtype* when defining the pointer type `PCHAR`, like this:

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

The typename `CHAR` in the first line becomes a *qualifiedtype* in the second line. Use of the **TYPEDEF** directive to define pointers is explained in "Accessing Data with Pointers and Addresses" in Chapter 3.

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*. For more information about procedure prototypes, see "Declaring Procedure Prototypes" in Chapter 7.

These 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 you do not specify *distance*, the assembler assumes a distance that corresponds to the memory model. The assumed distance is **NEAR** for tiny, small, and medium models, and **FAR** for other models.
- If you do not specify a memory model with **.MODEL**, the assembler assumes **SMALL** model (and therefore **NEAR** pointers).

You can use a *qualifiedtype* 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` `EXTERNDEF MyProc:PROTO`
With the **COMM** directive	`COMM var1:WORD:3`
With the **TYPEDEF** directive	`PBYTE TYPEDEF PTR BYTE` `PFUNC TYPEDEF PROTO MyProc`

"Defining Pointer Types with TYPEDEF" in Chapter 3 shows ways to write a **TYPEDEF** type for a *qualifiedtype*. Attributes such as **NEAR** and **FAR** can also apply to a *qualifiedtype*.

You can 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*.

Registers

The 8086 family of processors have the same base set of 16-bit registers. Each processor can treat certain registers as two separate 8-bit registers. The 80386/486 processors have extended 32-bit registers. To maintain compatibility

with their predecessors, 80386/486 processors can access their registers as 16-bit or, where appropriate, as 8-bit values.

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 used by the rest of the 8086 family. 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

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 purposes include:

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)	Contains the program stack for use by **PUSH**, **POP**, **CALL**, and **RET**.

Register and Segment	Purpose
ES (Extra Segment)	References secondary data segment. Used by string instructions.
FS, GS	Provides extra segments on the 80386/486.

General-Purpose Registers

The AX, DX, CX, BX, BP, DI, and SI registers are 16-bit general-purpose registers, used for temporary data storage. Since the processor accesses registers more quickly than it accesses memory, you can make your programs run faster by keeping the most-frequently used data in registers.

The 8086-based processors do not perform memory-to-memory operations. For example, the processor cannot directly copy a variable from one location in memory to another. You must first copy from memory to a register, then from the register to the new memory location. Similarly, to add two variables in memory, you must first copy one variable to a register, then add the contents of the register to the other variable in memory.

The processor can access four of the general registers—AX, DX, CX, and BX—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.

The 80386/486 processors can extend all the general registers to 32 bits, though as Figure 1.4 shows, you cannot treat the upper 16 bits as a separate register as you can the lower 16 bits. To use EAX as an example, you can directly reference the low byte as AL, the next lowest byte as AH, and the low word as AX. To access the high word of EAX, however, you must first shift the upper 16 bits into the lower 16 bits.

Special-Purpose Registers

The 8086 family of processors has two additional registers, SP and IP, 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 two; popping from the stack increases the value of SP by two. Thirty-two–bit operands on 80386/486 processors increase or decrease SP by four instead of two. The **CALL** and **INT** instructions store the return address on the stack and reduce SP accordingly. Return instructions retrieve the stored address from the stack and reset SP to its value before the call. SP can also be adjusted with instructions such as **ADD**. The program stack is described in detail in Chapter 3.

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.

Flags Register

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. Earlier 8086-family processors use only the lower word. The unmarked bits are reserved for processor use, and should not be modified.

Figure 1.5 Flags for 8088-80486 Processors

In the following descriptions and throughout this book, "set" means a bit value of 1, and "cleared" means the bit value is 0. The nine flags common to all 8086-family processors, starting with the low-order flags, include:

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 4 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	If set, string operations process down from high addresses to low addresses. If cleared, 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.

Although all flags serve a purpose, most programs require only the carry, zero, sign, and direction flags.

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, as shown here:

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

The following list explains each field:

Field	Purpose
name	Labels the statement, so that instructions elsewhere in the program can refer to the statement by name. The *name* field can label a variable, type, segment, or code location.
operation	Defines the action of the statement. This field contains either an instruction or an assembler directive.
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   ; Load AX with the value 7
```

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.

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 text and code between the delimiters or on the same line as the delimiters. This example illustrates the use of **COMMENT**.

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

The Assembly Process

Creating and running an executable file involves four steps:

1. Assembling the source code into an object file
2. Linking the object file with other modules or libraries into an executable program
3. Loading the program into memory
4. Running the program

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

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.

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, such as using macros instead of procedures, or using the **OPTION** directive or conditional assembly. The other chapters in this book include specific programming methods; this section simply gives you an overview.

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.

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 segments.
- Places segments and segment attributes in 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.

For information about conditional assembly, see “Conditional Directives” in this chapter; for macros, see Chapter 9. Further details about segments and offsets are included in Chapters 2 and 3. Assembly listings are explained in Appendix C.

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. 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 the result as an executable program file.

Classes and groups are defined in "Defining Segment Groups" in Chapter 2. Segments and offsets are explained in Chapter 3, "Using Addresses and Pointers."

Loading

After loading the executable file into memory, the operating system:

- Creates the program segment prefix (PSP) header in 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.

For information about segment registers, the instruction pointer (IP), and the stack pointer (SP), see "Registers" earlier in this chapter. For more information on the PSP see Help or an MS-DOS reference.

Running

To run your program, MS-DOS jumps to the program's first instruction. Some program operations, such as resolving indirect memory operands, cannot be handled until the program runs. For a description of indirect references, see "Indirect Operands" in Chapter 7.

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 **OPTION** keyword.

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 6.1 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
```

The assembler disables M510 compatibility options for all code following this statement:

```
OPTION NOM510
```

The following lists explain each of the arguments for the **OPTION** directive. Where appropriate, an underline identifies the default argument. If you wish to place more than one **OPTION** statement on a line, separate them by commas.

Options for M510 compatibility include:

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.
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. For more information on using **OPTION M510**, see Appendix A.

Argument	Description
OLDMACROS | **NOOLDMACROS**	Enables the version 5.1 treatment of macros. MASM 6.1 treats macros differently.
OLDSTRUCTS | **NOOLDSTRUCTS**	Enables compatibility with MASM 5.1 for treatment of structure members. See Chapter 5 for information on structures.
SCOPED | **NOSCOPED**	Guarantees that all labels inside procedures are local to the procedure when **SCOPED** (the default) is enabled.
SETIF2: TRUE | **FALSE**	If TRUE, **.ERR2** statements and **IF2** and **ELSEIF2** conditional blocks are evaluated on every pass. If FALSE, they are not evaluated. If **SETIF2** is not specified (or implied), **.ERR2**, **IF2**, and **ELSEIF2** expressions cause an error. Both the /Zm command-line argument and **OPTION M510** imply **SETIF2:TRUE**.

Options for procedure use include:

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 Chapter 7.
PROLOGUE: *macroname*	Instructs the assembler to call *macroname* to generate a user-defined prologue instead of generating the standard prologue code. See Chapter 7.
PROC: *visibility*	Lets you explicitly set the default visibility as **PUBLIC**, **EXPORT**, or **PRIVATE.**

Other options include:

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.

Argument	Description
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. For information about conditional-jump lengthening, see Chapter 7.
NOKEYWORD:*<keywordlist>*	Disables the specified reserved words. For an example of the syntax for this argument, see "Reserved Words" in this chapter.
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.
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.
READONLY | **NOREADONLY**	Enables checking for instructions that modify code segments, thereby guaranteeing that read-only code segments are not modified. Same as the /p command-line option of MASM 5.1, except that it affects only segments with at least one assembly instruction, not all segments. The argument is useful for protected mode programs, where code segments must remain 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**.

Conditional Directives

MASM 6.1 provides conditional-assembly directives and conditional-error directives. Conditional-assembly directives let you test for a specified condition and assemble a block of statements if the condition is true. Conditional-error directives allow you 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. You can test only expressions that evaluate to constants during assembly. For a list of the predefined symbols often used in conditional assembly, see "Predefined Symbols," earlier in this chapter.

Conditional-Assembly Directives

The **IF** and **ENDIF** directives enclose the conditional statements. The optional **ELSEIF** and **ELSE** blocks follow the **IF** directive. There are many forms of the **IF** and **ELSE** directives. Help provides a complete list.

The following statements show the syntax for the **IF** directives. The syntax for other condition-assembly directives follow the same form.

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

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

MASM assembles the statements following the **IF** directive only if the corresponding condition is true. If the condition is not true and the block contains an **ELSEIF** directive, 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 assembler processes only the statements following the **ELSE** directive.

For example, you may want to assemble a line of code only if your program defines a particular variable. In this example,

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

the assembler allocates `buff` only if `buffer` has been previously defined.

MASM 6.1 provides the directives **IF1**, **IF2**, **ELSEIF1**, and **ELSIF2** to grant assembly only on pass one or pass two. To use these directives, you must either enable 5.1 compatibility (with the /Zm command-line switch or **OPTION M510**) or set **OPTION SETIF2:TRUE**, as described in the previous section.

The following list summarizes the conditional-assembly directives:

The Directive	Grants Assembly If
IF *expression*	*expression* is true (nonzero)
IFE *expression*	*expression* is false (zero)
IFDEF *name*	*name* has been previously defined
IFNDEF *name*	*name* has not been previously defined
IFB *argument**	*argument* is blank
IFNB *argument**	*argument* is not blank
IFIDN[I] *arg1, arg2**	*arg1* equals *arg2*
IFDIF[I] *arg1, arg2**	*arg1* does not equal *arg2*
	The optional **I** suffix (**IFIDNI** and **IFDIFI**) makes comparisons insensitive to differences in case.

* Used only in macros.

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 MASM encounters a severe error during assembly, it does not generate the object module.

For example, the **.ERRNDEF** directive produces an error if the program has not defined a given label. In the following example, **.ERRNDEF** makes sure a label called `publevel` actually exists.

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

The conditional-error directives use the syntax given in the previous section. The following list summarizes the conditional-error directives. Note their close correspondence with the previous list of conditional-assembly directives.

The Directive	Generates an Error
.ERR	Unconditionally where it occurs in the source file. Usually placed within a conditional-assembly block.
.ERRE *expression*	If *expression* is false (zero).
.ERRNZ *expression*	If *expression* is true (nonzero).
.ERRDEF *name*	If *name* has been defined.
.ERRNDEF *name*	If *name* has not been defined.
.ERRB *argument**	If *argument* is blank.
.ERRNB *argument**	If *argument* is not blank.
.ERRIDN[I] *arg1, arg2**	If *arg1* equals *arg2*.
.ERRDIF[I] *arg1, arg2**	If *arg1* does not equal *arg2*.
	The optional **I** suffix (**.ERRIDNI** and **.ERRDIFI**) makes comparisons insensitive to case.

* Used only in macros

Two special conditional-error directives, **.ERR1** and **.ERR2**, generate an error only on pass one or pass two. To use these directives, you must either enable 5.1 compatibility (with the /Zm command-line switch or **OPTION M510**) or set **OPTION SETIF2:TRUE**, as described in the previous section.

CHAPTER 2

Organizing Segments

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

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

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

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

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

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

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

Physical Memory Segments

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

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

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

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

Logical Segments

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

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

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

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

Using Simplified Segment Directives

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

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

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

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

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

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

        .STACK          ; Use default 1-kilobyte stack

        .DATA           ; Begin data segment

                        ; Place data declarations here

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

                        ; Place instructions here

        .EXIT           ; Generate exit code
        END
```

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

Defining Basic Attributes with .MODEL

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

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

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

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

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

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

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

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

The next four sections give more detail on each field.

Defining the Memory Model

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

Table 2.1 Attributes of Memory Models

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

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

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

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

Small, Medium, Compact, Large, and Huge Models

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

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

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

Tiny Model

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

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

Flat Model

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

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

Choosing the Language Convention

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

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

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

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

Setting the Stack Distance

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

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

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

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

Specifying a Processor and Coprocessor

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

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

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

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

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

Creating a Stack

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

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

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

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

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

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

Creating Data Segments

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

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

Near Data Segments

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

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

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

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

Far Data Segments

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

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

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

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

Creating Code Segments

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

Near Code Segments

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

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

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

Far Code Segments

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

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

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

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

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

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

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

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

Starting and Ending Code with .STARTUP and .EXIT

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Using Full Segment Definitions

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

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

Defining Segments with the SEGMENT Directive

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

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

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

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

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

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

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

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

Aligning Segments

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

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

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

Making Segments Read-Only

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

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

Combining Segments

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

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

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

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

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

Setting Segment Word Sizes (80386/486 Only)

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

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

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

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

Setting Segment Order with Class Type

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

Controlling the Segment Order

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

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

Segment Order Directives

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

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

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

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

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

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

Linker Control

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

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

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

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

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

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

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

Setting the ASSUME Directive for Segment Registers

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

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

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

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

```
ASSUME CS : MY_CODE
```

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

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

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

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

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

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

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

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

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

You can also prevent the use of a register with:

```
ASSUME  SegRegister : ERROR
```

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

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

Defining Segment Groups

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

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

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

The syntax of the group directive is:

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

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

```
MYGROUP GROUP   CSEG
```

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

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

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

CHAPTER 3

Using Addresses and Pointers

MASM applications running in real mode require segmented addresses to access code and data. The address of the code or data in a segment is relative to a segment address in a segment register. You can also use pointers to access data in assembly language programs. (A pointer is a variable that contains an address as its value.)

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 access code and data. It also describes related operators, syntax, and displacements. The discussion of memory operands lays the foundation for the third section, which describes the stack.

The fourth section of this chapter explains how to use the **TYPEDEF** directive to declare pointers 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.

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 Chapter 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.

Initializing Default Segment Registers

The segmented architecture of the 8086-family of processors does not require that you specify two addresses every time you access memory. As explained in Chapter 2, “Organizing Segments,” 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 the segment registers yourself. Follow these 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.

Informing the Assembler About Segment Values

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 automatically generates the appropriate **ASSUME** statements. 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 "Defining Register Types with ASSUME" later in this chapter.)

The **.STARTUP** directive generates startup code that sets DS equal to SS (unless you specify **FARSTACK**), allowing 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 MS-DOS and Windows dynamic-link libraries (see Chapter 10). The code generated for **.STARTUP** is shown in "Starting and Ending Code with .STARTUP and .EXIT" in Chapter 2. You can use similar code to set DS equal to SS in programs using full segment definitions.

Here is an example of **ASSUME** using full segment definitions:

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

This example is equivalent to the **ASSUME** statement generated with simplified segment directives in small model with **NEARSTACK**. Note that 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 here:

```
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 is the current segment. When you use the simplified segment directives .**DATA**, .**DATA?**, .**CONST**, .**FARDATA**, or **.FARDATA?**, the assembler automatically assumes CS is the **ERROR** segment. 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 simple or full segments, you can cancel the control of an **ASSUME** statement by assuming **NOTHING**. You can cancel the previous assumption for ES 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.

Informing the Processor About Segment Values

The second and final 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 the operating system and on your use of simplified segment directives or full segment definitions.

Specifying a Starting Address

A program's starting address determines where execution begins. After the operating system loads a program, it simply jumps to the starting address, giving processor control to the program. The true starting address is known only to the loader; the linker determines only the offset of the address within an undetermined code segment. That's why a normal application is often referred to as "relocatable code," because it runs regardless of where the loader places it in memory.

The offset of the starting address depends on the program type. Programs with an .EXE extension contain a header from which the loader reads the offset and combines it with a segment to form the starting address. Programs with a .COM extension (tiny model) have no such header, so by convention the loader jumps to the first byte of the program.

In either case, the **.STARTUP** directive identifies where execution begins, provided you use simplified segment directives. For an .EXE program, place **.STARTUP** immediately before the instruction where you want execution to start. In a .COM program, place **.STARTUP** before the first assembly instruction in your source code.

If you use full segment directives or prefer not to use **.STARTUP**, you must identify the starting instruction in two steps:

1. Label the starting instruction.
2. Provide the same label in the **END** directive.

These steps tell the linker where execution begins in the program. The following example illustrates the two steps for a tiny model program:

```
_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
```

Notice the **ORG** statement in this example. This statement is mandatory in a tiny model program without the **.STARTUP** directive. It places the first instruction at offset 100h in the code segment to create space for a 256-byte (100h) data area called the Program Segment Prefix (PSP). The operating system takes care of initializing the PSP, so you need only make sure the area exists. (For a description of what data resides in the PSP, refer to the "Tables" chapter in the *Reference*.)

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 Windows. If you do not use **.STARTUP** with MS-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 previous example loads DGROUP, but you can load any valid segment or group.

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," page 45.) For an .EXE file, the stack address is encoded into the executable header and resolved

at load time. For a .COM file, the loader sets SS equal to CS and initializes SP to 0FFFEh.

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

Near and Far Addresses

Addresses that have an implied segment name or segment registers associated with them are called "near addresses." Addresses that have an explicit segment associated with them are called "far addresses." The assembler handles near and far code automatically, as described in the following sections. 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 "Controlling the Segment Order" in Chapter 2.

The assembler cannot determine the address for some program components; these are said to be relocatable. The assembler generates a fixup record and the linker provides the address once it has determined the location of all segments. Usually a relocatable operand references a label, but there are exceptions. Examples in the next two sections include information about relocating 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** instruction.

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

A program can access near data directly, because 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, you must reload the register with DGROUP's address before referencing any DGROUP data.

The processor assumes all memory references are relative to the segment in the DS register, with the exception of references using BP or SP. The processor associates these registers with the SS register. (You can override these assumptions with the segment override operator, described in "Direct Memory Operands," on page 62.)

The following lines illustrate how the processor accesses either the DS or SS segments, depending on whether the pointer operand contains BP or SP. Note the distinction loses significance when DS and SS are equal.

```
nearvar WORD    0
        .
        .
        .
        mov     ax, nearvar ; Reads from DS:[nearvar]
        mov     di, [bx]    ; Reads from DS:[bx]
        mov     [di], cx    ; Writes to  DS:[di]
        mov     [bp+6], ax  ; Writes to  SS:[bp+6]
        mov     bx, [bp]    ; Reads from SS:[bp]
```

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.

Note Flat model does not require far addresses. By default, all addressing is relative to the initial values of the segment registers. Therefore, this section on far addressing does not apply to flat model programs.

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
        mov     es, ax          ,   far address into ES
        mov     ax, es:farvar   ; Provide an explicit segment
                                ;   override on the addressing
```

```
; Second method
        mov     ax, SEG farvar2 ; Load the segment of the
        mov     es, ax          ;   far address into ES
        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 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. (For more information on segment overrides, see "Direct Memory Operands" on page 62.)

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 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). However, some compilers require that you restore ES before returning to a module written in a high-level language.

To access far data, first set DS to the far segment and then restore the original DS when finished. Use the **ASSUME** directive to let the assembler know that DS no longer points to the default data segment, as shown here:

```
        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] ; Set DX:AX = dword variable
        mov     dx, fararray[2] ;   fararray
        .
        .
        .
        pop     ds              ; Restore segment
        ASSUME  ds:@DATA        ;   and default assumption
```

"Direct Memory Operands,"on page 62, describes an alternative method for accessing far data. The technique of resetting DS as shown in the previous example is best for a lengthy series of far data references. The segment override method described in "Direct Memory Operands" serves best when accessing only one or two far variables.

If your 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. Many compilers, including Microsoft compilers, use this convention.

Operands

With few exceptions, assembly language instructions work on sources of data called operands. In a listing of assembly code (such as the examples in this book), operands appear in the operand field immediately to the right of the instructions.

This section describes the four kinds of instruction operands: register, immediate, direct memory, and indirect memory. Some instructions, such as POPF and STI, have implied operands which do not appear in the operand field. Otherwise, an implied operand is just as real as one stated explicitly.

Certain other instructions such as NOP and WAIT deserve special mention. These instructions affect only processor control and do not require an operand.

The following four types of operands are described 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.

Instructions that take two or more operands always work right to left. The right operand is the source operand. It specifies data that will be read, but not changed, in the operation. The left operand is the destination operand. It specifies the data that will be acted on and possibly changed by the instruction.

Register Operands

Register operands refer to data stored in registers. The following examples show typical register operands:

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

An offset stored in a base or index register often serves as a pointer into memory. You can store an offset in one of the base or index registers, then use the register as an indirect memory operand. (See "Indirect Memory Operands," following.) 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 2 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.

Immediate Operands

An immediate operand is a constant or the result of a constant expression. The assembler encodes immediate values into the instruction at assembly time. Here are some typical examples showing immediate operands:

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

Immediate data is never permitted in the destination operand. If the source operand is immediate, the destination operand must be either a register or direct memory to provide a place to store the result of the operation.

Immediate expressions often involve the useful **OFFSET** and **SEG** operators, described in the following paragraphs.

The OFFSET Operator

An address constant is a special type of immediate operand that consists of an offset or segment value. The **OFFSET** operator returns the offset of a memory location, as shown here:

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

For information on differences between MASM 5.1 behavior and MASM 6.1 behavior related to **OFFSET**, see Appendix A.

Since data in different modules may belong to a single segment, the assembler cannot know for each module the true offsets within a segment. Thus, the offset for `var`, although an immediate value, is not determined until link time.

The SEG Operator

The **SEG** operator returns the segment of a memory location:

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

The actual value of a particular segment is not known until the program is loaded into memory. For .EXE programs, the linker makes a list in the program's header of all locations in which the **SEG** operator appears. The loader reads this list and fills in the required segment address at each location. Since .COM programs have no header, the assembler does not allow relocatable segment expressions in tiny model programs.

The **SEG** operator returns a variable's "frame" if it appears in the instruction. The frame is the value of the segment, group, or segment override of a nonexternal variable. For example, the instruction

```
mov     ax, SEG DGROUP:var
```

places in AX the value of DGROUP, where `var` is located. If you do not include a frame, **SEG** returns the value of the variable's group if one exists. If the variable is not defined in a group, **SEG** returns the variable's segment address.

This behavior can be changed with the /Zm command-line option or with the **OPTION OFFSET:SEGMENT** statement. (See Appendix A, "Differences between MASM 6.1 and 5.1.") "Using the OPTION Directive" in Chapter 1 introduces the **OPTION** directive.

Direct Memory Operands

A direct memory operand specifies the data at a given address. The instruction acts on the contents of the address, not the address itself. Except when size is implied by another operand, you must specify the size of a direct memory operand so the instruction accesses the correct amount of memory. The following example shows how to explicitly specify data size with the **BYTE** directive:

```
        .DATA?                  ; Segment for uninitialized data
var     BYTE   ?                ; Reserve one byte, labeled "var"
        .CODE
        .
        .
        .
        mov    var, al          ; Copy AL to byte at var
```

Any location in memory can be a direct memory operand as long as a size is specified (or implied) 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 in the following section.

Plus, Minus, and Index

The plus and index operators perform in exactly the same way when applied to direct memory operands. For example, both the following statements move the second word value from an array into the AX register:

```
        mov     ax, array[2]
        mov     ax, array+2
```

The index operator can contain any direct memory operand. The following statements 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.

The minus operator behaves as you would expect. Both the following instructions retrieve the value located at the word preceding `array`:

```
mov     ax, array[-2]
mov     ax, array-2
```

Structure Field

The structure operator (**.**) references a particular element of a structure or "field," to use C terminology:

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

The address of the structure operand is the sum of the offsets of `structvar` and `field1`. For more information about structures, see "Structures and Unions" in Chapter 5.

Segment Override

The segment override operator (**:**) 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 assemble in exactly the same way:

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

A segment name override or the segment override operator identifies the operand as 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.

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 you can use 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 compatible with earlier processors.

This section covers features of indirect operands in either mode. The specific 16-bit rules and 32-bit rules are then explained separately.

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.

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 into AX the word value at the address in DS:BX.

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

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

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

Specifying Displacements

You can specify an address displacement, which is a constant value added to the effective address. A direct memory specifier is the most common displacement:

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

In this relocatable expression, 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 executes.

Each displacement can be an address or numeric constant. If there is more than one displacement, the assembler totals them 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 `6` to `table` to get the total displacement. However, the statement

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

is not legal, because it attempts to use a single command to join the contents of two different addresses.

Specifying Operand Size

You must give the size of an indirect memory operand in one of three ways:

- By the variable's declared size
- With the **PTR** operator
- Implied by the size of the other operand

The following lines illustrate all three methods. Assume the size of the `table` array is **WORD**, as declared earlier.

```
        mov     table[bx], 0      ; 2 bytes - from size of table
        mov     BYTE PTR table, 0 ; 1 byte  - specified by BYTE
        mov     ax, [bx]          ; 2 bytes - implied by AX
```

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 assemble the same way:

```
        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.

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 by one place). When using 16-bit registers, you must scale with separate instructions, as shown here:

```
        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 "Indirect Memory Operands with 32-Bit Registers," following.

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 is initialized
        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
        mov     di, ax                  ;   elements to point DI
                                        ;   to current element
        mov     al, (STUDENT PTR[bx+di]).year
```

For more information on MASM structures, see "Structures and Unions" in Chapter 5.

Indirect Memory Operands with 16-Bit Registers

For 8086-based computers and MS-DOS, you must follow the strict indexing rules established for the 8086 processor. Only four registers are allowed—BP, BX, SI, and DI—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 register modes in which you can specify indirect memory operands.

Table 3.1 Indirect Addressing 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 *Reference*.

Indirect Memory Operands with 32-Bit Registers

You can write instructions for the 80386/486 processor using either 16-bit or 32-bit segments. Indirect memory operands are different in each case.

In 16-bit real mode, the 80386/486 operates the same way as earlier 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 with either 16-bit or 32-bit segments. Thirty-two–bit

registers eliminate many of the limitations of 16-bit indirect memory operands. You can use 80386/486 features to make your MS-DOS programs run faster and more efficiently if you are willing to sacrifice compatibility with earlier processors.

In 32-bit mode, an offset address can be up to 4 gigabytes. (Segments are still represented in 16 bits.) This effectively eliminates size restrictions on each segment, since few programs need 4 gigabytes of memory. Windows NT uses 32-bit mode and flat model, which spans all segments. XENIX 386 uses 32-bit mode with multiple segments.

80386/486 Enhancements

On the 80386/486, the processor allows you to use any general-purpose 32-bit register as a base or index register, except ESP, which can be a base but not an index. However, you cannot combine 16-bit and 32-bit registers. Several examples are shown here:

```
add     edx, [eax]              ; Add double
mov     dl, [esp+10]            ; Copy 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
```

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. To specify the scaling factor, use 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 "Indirect Memory Operands with 16-Bit Registers," previous.

The default segment register is SS if the base register is EBP or ESP. However, if EBP is scaled, the processor treats it as an index register with a value relative to DS, not SS.

All other base registers are relative to DS. 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]        ; EBP base (only - seg SS)
mov   eax, [ebp*2]      ; EBP*2 index (seg DS)
```

Mixing 16-Bit and 32-Bit Registers

Assembly statements can mix 16-bit and 32-bit registers. For example, the following statement is legal for 16-bit and 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 you cannot use the CX register as a 16-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 16-bit or 32-bit mode:

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

This statement moves the 16-bit value pointed to by EAX into the BX register. This works 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, generating 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.

The Program Stack

The preceding discussion on memory operands lays the groundwork for understanding the important data area known as the "stack."

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 starting from high memory. Data is always pushed onto, or "popped" from the top of the stack.

The stack gets its name from its similarity to the spring-loaded plate holders in cafeterias. You add and remove plates from only the top of the stack. To retrieve the third plate, you must remove—that is, "pop"—the first two plates. Stacks are often referred to as LIFO buffers, from their last-in-first-out operation.

A stack is an essential part of any nontrivial program. A program continually uses its stack to temporarily store return addresses, procedure arguments, memory data, flags, or registers.

The SP register serves as an indirect memory operand to the top of the stack. At first, the stack is an uninitialized segment of a finite size. As your program adds data to the stack, the stack grows downward from high memory to low memory. When you remove items from the stack, it shrinks upward from low to high memory.

Saving Operands on the Stack

The **PUSH** instruction stores a 2-byte operand on the stack. The **POP** instruction retrieves the most recent 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. Since the stack always contains word values, the SP register 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 4-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 3.1 illustrates how pushes and pops change the SP register.

Figure 3.1 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
```

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 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, remember 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 word in AX
mov     bx, [bp-4]      ; Put second word in BX
mov     cx, [bp-2]      ; Put first word in CX
.
.
.
add     sp, 6           ; Restore stack pointer
                        ;   (two bytes per push)
pop     bp              ; Restore BP
```

If you often use these stack values 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 previous example with `mov ax, count`. For more information about the **TEXTEQU** directive, see "Text Macros" in Chapter 9.

Saving Flags on the Stack

Your program can push and pop flags onto the stack with the **PUSHF** and **POPF** instructions. These instructions save and then restore the status of the flags. 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 flags 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. **SAHF** restores the value.

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 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**.

Accessing Data with Pointers and Addresses

A pointer is simply a variable that contains an address of some other variable. The address in the pointer "points" to the other object. Pointers are useful when transferring a large data object (such as an array) to a procedure. The caller places only the pointer on the stack, which the called procedure uses to locate the array. This eliminates the impractical step of having to pass the entire array back and forth through the stack.

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 contains the segment address and offset of some other data. Like any other variable, a pointer 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 with the name of another variable, the initialized variable is a pointer, as shown in this example. 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 **TYPEDEF** directive and enhanced capabilities of **ASSUME** (introduced in MASM 6.0) make it easier to manage pointers in registers and variables. The rest of this chapter describes these directives and how they apply to basic pointer operations.

Defining Pointer Types with TYPEDEF

The **TYPEDEF** directive can define types for pointer variables. A type so defined is considered the same as the intrinsic types provided by the assembler and can be used in the same contexts. When used to define pointers, the syntax for **TYPEDEF** 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**. (For a full definition of *qualifiedtype*, see "Data Types" in Chapter 1.)

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

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

The distance of a pointer can 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 2 bytes that contain the offset of the object pointed to. A far pointer requires 4 bytes, and contains both the segment and offset. In 32-bit mode, a near pointer is 4 bytes and a far pointer is 6 bytes, since segments are

still word values in 32-bit mode. If you specify the distance with **NEAR** or **FAR**, the processor uses the default distance of the current segment size. 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.

You can declare pointer variables with a pointer type created with **TYPEDEF**. Here are some examples using these pointer types.

```
; 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.

Defining Register Types with ASSUME

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     ; Assume BX is now a word pointer
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 assumption
```

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

Without the assumption to **PTR WORD**, many instructions need a size specifier. The **INC** and **MOV** statements from the previous examples 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 this example, 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     al, 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
```

For information on using **ASSUME** with segment registers, refer to "Setting the ASSUME Directive for Segment Registers" in Chapter 2.

Basic Pointer and Address Operations

A program can perform the following 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 pointers and addresses. The examples are used with the assumption 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
```

Initializing Pointer Variables

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 shows how to do this, placing the address of `msg` in the pointer `pmsg`.

```
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 here:

```
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 in a register, you can load it directly:

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

Dynamic Addresses

Often a pointer must point to a dynamic address, meaning the address depends on a run-time condition. Typical situations include memory allocated by MS-DOS (see "Interrupt 21h Function 48h" in Help) and addresses found by the **SCAS** or **CMPS** instructions (see "Processing Strings" in Chapter 5). The following illustrates the technique for saving dynamic addresses:

```
; 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
```

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     ax, WORD PTR fpBuf1[0]
        mov     WORD PTR fpBuf2[0], ax
        mov     ax, WORD PTR fpBuf1[2]
        mov     WORD PTR fpBuf2[2], ax

; 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 as Arguments

Most high-level-language procedures and library functions accept arguments passed on the stack. “Passing Arguments on the Stack” in Chapter 7 covers this subject in detail. A pointer is passed in the same way as any other variable, as this fragment shows:

```
; 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 has the same result as pushing a pointer to the address:

```
; 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 push a constant in one step:

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

Loading Addresses into Registers

Loading a near address into a register (or a far address into a pair of registers) is a common task in assembly-language programming. To reference data pointed to by a pointer, your program must first place the pointer into a register or pair of registers.

Load far addresses as *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 certain DOS functions
ES:BX	Output from certain DOS functions

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 into DS:BX from a near data segment:

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

Far data can be loaded like this:

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

You can also read a far address from a pointer in one step, using the **LES** and **LDS** instructions described next.

Far Pointers

The **LES** and **LDS** instructions load a far pointer into a segment pair. The instructions copy the pointer's low word into either ES or DS, and the high word into a given register. The following example shows how to load a far pointer into ES:DI:

```
OutBuf  BYTE    20 DUP (0)

fpOut   FPBYTE  OutBuf
        .
        .
        .
        les     di, fpOut      ; Load far pointer into ES:DI
```

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
```

The local variable in this case actually evaluates to SS:[BP-4]. This is an offset from the stack frame (described in "Passing Arguments on the Stack," Chapter 7). 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.

Direct Memory Operands

To get the address of a direct memory operand, use either the **LEA** instruction or the **MOV** instruction with **OFFSET**. Though both methods have the same effect, the **MOV** instruction produces smaller and faster code, as shown in this example:

```
        lea     si, Msg        ; Four byte instruction
        mov     si, OFFSET Msg ; Three byte equivalent
```

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 copying methods are shown here. 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
```

Model-Independent Techniques

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.

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                       ; @DataSize > 0 for far
        mov     WORD PTR pMsg[0], OFFSET Msg1   ; Load far offset
        mov     WORD PTR pMsg[2], SEG Msg1      ; Load far segment
        ELSE                                    ; @DataSize = 0 for near
        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 automatically generate the proper instructions and segment overrides.

```
; 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 these examples can be defined once in an include file and used whenever you need to handle pointers.

CHAPTER 4

Defining and Using Simple Data Types

This chapter covers the concepts essential for working with simple data types in assembly-language programs. The first section shows how to declare integer variables. The second section describes basic operations including moving, loading, and sign-extending numbers, as well as calculating. The last section describes how to do various operations with numbers 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 operations illustrated in this chapter. 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."

Declaring Integer Variables

An integer is a whole number, such as 4 or 4,444. Integers have no fractional part, as do the real numbers discussed in Chapter 6. You can initialize integer variables in several ways with the data allocation directives. This section 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 "Integer Constants and Constant Expressions" in Chapter 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*

The following directives indicate the integer's size and value range:

Directive	Description of Initializers
BYTE, **DB** (byte)	Allocates unsigned numbers from 0 to 255.
SBYTE (signed byte)	Allocates signed numbers from –128 to +127.
WORD, DW (word = 2 bytes)	Allocates unsigned numbers from 0 to 65,535 (64K).
SWORD (signed word)	Allocates signed numbers from –32,768 to +32,767.
DWORD, DD (doubleword = 4 bytes),	Allocates unsigned numbers from 0 to 4,294,967,295 (4 megabytes).
SDWORD (signed doubleword)	Allocates signed numbers from –2,147,483,648 to +2,147,483,647.
FWORD, DF (farword = 6 bytes)	Allocates 6-byte (48-bit) integers. These values are normally used only as pointer variables on the 80386/486 processors.
QWORD, DQ (quadword = 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 **SIZEOF** and **TYPE** operators, when applied to a type, return the size of an integer of that type. The size attribute associated with each data type is:

Data Type	Bytes
BYTE, SBYTE	1
WORD, SWORD	2
DWORD, SDWORD	4
FWORD	6
QWORD	8
TBYTE	10

The data types **SBYTE**, **SWORD**, and **SDWORD** tell 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**, and with **PROTO** and **INVOKE** directives. For descriptions of these directives, see the sections "Loop-Generating Directives," "Declaring Procedure Prototypes," and "Calling Procedures with INVOKE" in Chapter 7.

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

Although the **TYPEDEF** directive's primary purpose is to define pointer variables (see "Defining Pointer Types with TYPEDEF" in Chapter 3), you can also use **TYPEDEF** to create an alias for any integer type. For example, these declarations

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

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

Data Initialization

You can initialize variables when you declare them with constants or expressions that evaluate to constants. The assembler generates an error if you specify an initial value too large for the variable type.

A **?** in place of an initializer indicates you do not require the assembler to initialize the variable. The assembler allocates the space but does not write in it. Use **?** for buffer areas or variables your program will initialize at run time.

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 int
                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
```

For information on arrays and on using the **DUP** operator to allocate initializer lists, see "Arrays and Strings" in Chapter 5.

Working with Simple Variables

Once you have declared integer variables in your program, you can use them to copy, move, 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 numbers and do bit-level manipulations with logical, shift, and rotate instructions.

Since MASM instructions require operands to be the same size, you may need to operate on data in a size other than that originally declared. You can do this with the **PTR** operator. 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
```

Copying Data

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

Moving Data

The most common method of moving data, the **MOV** instruction, is essentially a copy instruction, since it always copies the source operand to the destination operand without affecting the source. After a **MOV** instruction, the source and destination operands contain the same value.

The following example illustrates the **MOV** instruction. As explained in "General-Purpose Registers," Chapter 1, you cannot move a value from one location in memory to another in a single 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
```

The following example shows several common types of moves that require two instructions.

```
; Move immediate to segment register
        mov     ax, DGROUP  ; Load AX with immediate value
        mov     ds, ax      ; Copy AX to segment register

; Move memory to memory
        mov     ax, mem1    ; Load AX with memory value
        mov     mem2, ax    ; Copy AX to other memory

; Move segment register to segment register
        mov     ax, ds      ; Load AX with segment register
        mov     es, ax      ; Copy AX to segment register
```

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

Exchanging Integers

The **XCHG** (Exchange) instruction exchanges the data in the source and destination operands. You can exchange data 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 memory locations
```

Extending Signed and Unsigned Integers

Since moving data between registers of different sizes is illegal, you must "sign-extend" integers to convert signed data to a larger size. Sign-extending means copying the sign bit of the unextended operand to all bits of the operand's next larger size. This widens the operand while maintaining its sign and value.

8086-based processors provide four instructions specifically for sign-extending. The four instructions act only on the accumulator register (AL, AX, or EAX), as shown in the following list.

Instruction	Sign-extend
CBW (convert byte to word)	AL to AX
CWD (convert word to doubleword)	AX to DX:AX
CWDE (convert word to doubleword extended)*	AX to EAX
CDQ (convert doubleword to quadword)*	EAX to EDX:EAX

*Requires an extended register and applies only to 80386/486 processors.

On the 80386/486 processors, 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
        cwd                 ; Convert to 32-bit +5 (0000:0005h) in DX:AX
        mov     ax, mem16   ; Load 16-bit +5
        cwde                ; Convert to 32-bit +5 (00000005h) in EAX
        mov     eax, mem32  ; Load 32-bit -5 (FFFFFFFBh)
        cdq                 ; Convert to 64-bit -5
                            ;   (FFFFFFFF:FFFFFFFBh) in EDX:EAX
```

These four instructions efficiently convert unsigned values as well, provided the sign bit is zero. This example, for instance, correctly widens `mem16` whether you treat the variable as signed or unsigned.

The processor does not differentiate between signed and unsigned values. For instance, the value of `mem8` in the previous example is literally 251 (0FBh) to the processor. It ignores the human convention of treating the highest bit as an indicator of sign. The processor can ignore the distinction between signed and unsigned numbers because binary arithmetic works the same in either case.

If you add 7 to `mem8`, for example, the result is 258 (102h), a value too large to fit into a single byte. The byte-sized `mem8` can accommodate only the least-significant digits of the result (02h), and so receives the value of 2. The result is the same whether we treat `mem8` as a signed value (-5) or unsigned value (251).

This overview illustrates how the programmer, not the processor, must keep track of which values are signed or unsigned, and treat them accordingly. If AL=127 (01111111y), the instruction **CBW** sets AX=127 because the sign bit is zero. If AL=128 (10000000y), however, the sign bit is 1. **CBW** thus sets AX=65,280

(FF00h), which may not be what you had in mind if you assumed AL originally held an unsigned value.To widen unsigned values, explicitly set the higher register to zero, as shown in the following example:

```
        .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)

        sub     eax, eax  ; Zero entire extended register (EAX)
        mov     ax, mem16 ; Load 251 (FBh) from 16-bit memory
```

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 80386/486 instructions usually execute much faster than the equivalent 8086/286 instructions.

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.

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 "Adding and Subtracting in Multiple Registers," following.)

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.

To meet the second requirement, you can use the **PTR** operator to force an operand to the size required. (See "Working with Simple Variables," previous.) 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] ; Add word from byte array
```

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.

When the sum of 8-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 previous subtraction example, 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. Jump instructions are described in the "Jumps" section in Chapter 7.

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 processors, 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 **SBB**, be sure to turn off the carry flag with the **CLC** (Clear Carry Flag) instruction or use **ADD** or **SUB** for the first calculation, when appropriate.

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.

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 unsigned 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 signed 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 additional 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*

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. The syntax is:

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**.

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 "Extending Signed and Unsigned Integers," earlier in this chapter.

Manipulating Numbers at the Bit Level

The instructions introduced so far in this chapter access numbers at the byte or word level. The logical, shift, and rotate instructions described in this section access individual bits in a number. 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 reviews some applications of these bit-level operations.

Logical Instructions

The logical instructions **AND**, **OR**, and **XOR** compare bits in two operands. Based on the results of the comparisons, the instructions alter bits in the first (destination) operand. The logical instruction **NOT** also changes bits, but operates on a single operand.

The following list summarizes these four logical instructions. The list makes reference to the "destination bit," meaning the bit in the destination operand. The terms "both bits" and "either bit" refer to the corresponding bits in the source and destination operands. These instructions include:

Instruction	Sets Destination Bit If	Clears Destination Bit If
AND	Both bits set	Either or both bits clear
OR	Either or both bits set	Both bits clear
XOR	Either bit (but not both) set	Both bits set or both clear
NOT	Destination bit clear	Destination bit set

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.

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 any number with a pattern of bits set for an 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
```

The **AND** instruction clears unmasked bits—that is, bits not protected by 1 in the mask. To mask off certain bits in an operand and clear the others, use an appropriate masking value in the source operand. 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.

The **OR** instruction forces 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.

The **XOR** instruction toggles the value of specific bits on and off—that is, reverses 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.

```
;AND example - 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:    .

;OR example - compares operand to 0
        or      bx, bx          ; Compare to 0
        jg      positive        ; BX is positive
        jl      negative        ; BX is negative
                                ; else BX is zero

;XOR example - 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 80386/486 processors, the **BSF** (Bit Scan Forward) and the **BSR** (Bit Scan Reverse) instructions perform operations like 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.

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 4.2 illustrates the eight variations of shift and rotate instructions for 8-bit operands. Notice that **SHL** and **SAL** are identical.

Figure 4.2 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.

Starting with the 80186 processor, you can use 8-bit immediate values larger than 1 as the source operand for shift or rotate instructions, as shown here:

```
        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 total
```

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
```

Multiplying and Dividing with Shift Instructions

You can use the shift and rotate instructions (**SHR**, **SHL**, **SAR**, and **SAL**) for multiplication and division. Shifting a value right by one bit has the effect of dividing by two; shifting left by 1 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 negative numbers down—**IDIV** always rounds negative numbers up (toward 0). 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**.

Multiply and divide instructions are relatively slow, particularly on the 8088 and 8086 processors. When multiplying or dividing by a power of two, use shifts to speed operations by a factor of 10 or more. For example, these statements take only four clocks on an 8088 or 8086 processor:

```
        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 processors. The same statements take 15 clocks on the 80286 and between 11 and 16 clocks on the 80386. (For a discussion about instruction timings, see “A Word on Instruction Timings” in the Introduction.)

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

As the following macro shows, it’s possible to multiply by any number—in this case, 10—without resorting to the **MUL** instruction. However, such a procedure is no more than an interesting arithmetic exercise, since the additional code almost certainly takes more time to execute than a single **MUL**. You should consider using shifts in your program only when multiplying or dividing by a power of two.

```
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
```

Here’s another macro that divides by 512. In contrast to the previous example, this macro uses little code and operates faster than an equivalent **DIV** instruction.

```
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)
```

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 9-bit or 17-bit operand), the flag value before the operation is crucial. The carry flag can be adjusted by a previous instruction, but you can also set or clear the flag directly with the **CLC** (Clear Carry Flag), **CMC** (Complement Carry Flag), and **STC** (Set Carry Flag) instructions.

On the 80386 and 80486 processors, 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
```

Scaling of 80386 indirect memory operands is reviewed in "Indirect Memory Operands with 32-Bit Registers" in Chapter 3. **LEA** is introduced in "Loading Addresses into Registers" in Chapter 3.

The next chapter deals with 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 covered in Chapter 5, “Defining and Using Complex Data Types.”

CHAPTER 5

Defining and Using Complex Data Types

With the complex data types available in MASM 6.1—arrays, strings, records, structures, and unions—you can access data as a unit or as individual elements that make up a unit. The individual elements of complex data types are often the integer types discussed in Chapter 4, "Defining and Using Simple Data Types."

"Arrays and Strings" reviews 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.

"Structures and Unions" 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.

"Records" explains how to declare record types, define record variables, and use record operators.

Arrays and Strings

An array is a sequential collection of variables, all of the same size and type, called "elements." A string is an array of characters. For example, in the string "ABC," each letter is an element. You can access the elements in an array or string relative to the first element. This section explains how to handle arrays and strings in your programs.

Declaring and Referencing Arrays

Array elements occupy memory contiguously, so a program references each element relative to the start of the array. To declare an array, supply a label name, the element type, and a series of initializing values or **?** placeholders. The following examples declare the arrays `warray` and `xarray`:

```
warray  WORD    1, 2, 3, 4
xarray  DWORD   0FFFFFFFFh, 789ABCDEh
```

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 use the **LENGTHOF** and **SIZEOF** operators discussed later in this section, an array may span more than one logical line, although a separate type declaration is needed on each logical line:

```
var1    BYTE    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 works with any of the data allocation directives described in "Allocating Memory for Integer Variables" in Chapter 4. In the syntax

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

the *count* value sets the number of times to repeat all values within the parentheses. The *initialvalue* can be an integer, character constant, or another **DUP** operator, and must always appear within parentheses. For example, the statement

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

allocates the integer 1 five times for a total of 5 bytes.

The following examples show various ways to allocate data elements with the **DUP** operator:

```
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

Each element in an array is referenced with an index number, beginning with zero. The array index appears in brackets after the array name, as in

```
array[9]
```

Assembly-language indexes differ from indexes in high-level languages, where the index number always corresponds to the element's position. In C, for example, `array[9]` references the array's tenth element, regardless of whether each element is 1 byte or 8 bytes in size.

In assembly language, an element's index refers to the number of bytes between the element and the start of the array. This distinction can be ignored for arrays of byte-sized elements, since an element's position number matches its index. For example, defining the array

```
prime  BYTE 1, 3, 5, 7, 11, 13, 17
```

gives a value of 1 to `prime[0]`, a value of 3 to `prime[1]`, and so forth.

However, in arrays with elements larger than 1 byte, index numbers (except zero) do not correspond to an element's position. You must multiply an element's position by its size to determine the element's index. Thus, for the array

```
wprime  WORD 1, 3, 5, 7, 11, 13, 17
```

`wprime[4]` represents the third element (5), which is 4 bytes from the beginning of the array. Similarly, the expression `wprime[6]` represents the fourth element (7) and `wprime[10]` represents the sixth element (13).

The following example determines an index at run time. It multiplies the position by two (the size of a word element) by shifting it left:

```
mov     si, cx          ; CX holds position number
shl     si, 1           ; Scale for word referencing
mov     ax, wprime[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]

Referencing an array element by distance rather than position is not difficult to master, and is actually very consistent with how assembly language works. Recall that a variable name is a symbol that represents the contents of a particular address in memory. Thus, if the array `wprime` begins at address DS:2400h, the reference `wprime[6]` means to the processor "the word value contained in the DS segment at offset 2400h-plus-6-bytes."

As described in "Direct Memory Operands," Chapter 3, you can substitute the plus operator (+) for brackets, as in:

```
wprime[9]
wprime+9
```

Since brackets simply add a number to an address, you don't need them when referencing the first element. Thus, `wprime` and `wprime[0]` both refer to the first element of the array `wprime`.

If your program runs only on an 80186 processor or higher, you can use the **BOUND** instruction to verify that an index value is within the bounds of an array. For a description of **BOUND**, see the *Reference*.

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 elements in the array. The **SIZEOF** operator returns the number of bytes used by the initializers in the array definition. **TYPE** returns the size of the elements of the array. The following 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
```

Declaring and Initializing Strings

A string is an array of characters. Initializing a string like `"Hello, there"` allocates and initializes 1 byte for each character in the string. An initialized string can be no longer than 255 characters.

For data directives other than **BYTE**, a string may initialize only the first element. The initializer value must fit into the specified size and conform to the expression word size in effect (see “Integer Constants and Constant Expressions” in Chapter 1), as shown in these examples:

```
wstr    WORD    "OK"
dstr    DWORD   "DATA"  ; 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.

```
PBYTE   TYPEDEF PTR BYTE
        .DATA
msg1    BYTE    "Operation completed successfully."
msg2    BYTE    "Unknown command"
msg3    BYTE    "File not found"
pmsg    PBYTE   msg1            ; pmsg is an array
        PBBYTE  msg2            ;   of pointers to
        PBYTE   msg3            ;   above messages
```

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

You do not have to initialize an array. The **?** operator lets you allocate space for the array without placing specific values in it. Object files contain records for initialized data. Unspecified space 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. If the **?** initializer does not appear in a **DUP** statement, or if the **DUP** statement contains only **?** initializers, the assembler leaves the allocated space unspecified.

LENGTHOF, SIZEOF, and TYPE for Strings

Because strings are simply arrays of byte elements, the **LENGTHOF**, **SIZEOF**, and **TYPE** operators behave as you would expect, as illustrated in this example:

```
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
```

Processing 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 processors, and on arrays of bytes, words, and doublewords on the 80386/486 processors. Processing larger elements must be done indirectly with loops.

The following list gives capsule descriptions of the five instructions discussed in this section.

Instruction	Description
MOVS	Copies a string from one location to another
STOS	Stores contents of the accumulator register to a string
CMPS	Compares one string with 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.

Overview of String Instructions

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:

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 MS-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 100 elements in a 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. For information on changing segment registers, see "Programming Segmented Addresses" in Chapter 3.

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 the 80386/486 processors.

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 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 the next array element. 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 1, 2, or 4 bytes each time.

Each prefix governs the number of repetitions as follows:

Prefix	Description
REP	Repeats instruction CX times
REPE, **REPZ**	Repeats instruction maximum CX times while values are equal
REPNE, **REPNZ**	Repeats instruction maximum CX times while values are not equal

The prefixes apply to only one string instruction at a time. To repeat a block of instructions, use a loop construction. (See "Loops" in Chapter 7.)

At run time, if a string instruction is preceded by a repeat sequence, the processor:

1. Checks the CX register and exits if CX is 0.
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.
4. Decrements CX without modifying the flags.

5. Checks the zero flag (for **SCAS** or **CMPS**) if the **REPE** or **REPNE** prefix is used. If the repeat condition holds, loops back to step 1. Otherwise, the loop ends and execution proceeds to the next instruction.

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 last match occurred.

Although string instructions (except **LODS**) are used most often with repeat prefixes, they can also be used by themselves. In these cases, the SI and/or DI registers are adjusted as specified by the direction flag and the size of operands.

Using String Instructions

To use the 8086-family string instructions, follow 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 the section "Structures and Unions," following.)

Moving Array Data

The **MOVS** instruction copies data from one area of memory to another. To move data, first load the count, 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
```

Filling 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.

The next example stores the character `'a'` in each byte of a 100-byte string, filling the entire string with "aaaa...." Notice how the code stores 50 words rather than

100 bytes. This makes the fill operation faster by reducing the number of iterations. To fill an odd number of bytes, you need 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 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
        je      allmatch            ; Jump if all match
        .
        .
        .
allmatch:                           ; Special case for all match
```

Loading Data from Arrays

The **LODS** instruction loads a value from a string into the accumulator register. This instruction is not used with a repeat instruction prefix, since continually reloading the accumulator serves no purpose.

The code in this example loads, processes, and displays each byte in a string.

```
        .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 compares the value pointed to by ES:DI with the value in the accumulator. If both values are the same, it sets the zero flag.

A repeat prefix lets **SCAS** work on an entire string, scanning (from which **SCAS** gets its name) for a particular value called the target. **REPNE SCAS** sets the zero flag if it finds the target value in the array. **REPE SCAS** sets the zero flag if the scanned array contains nothing but the target value.

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
        jne     notfound           ; Jump if not found
        .                          ; ES:DI points to character
        .                          ;   after first 'z'
        .
notfound:                          ; Special case for not found
```

Translating Data in Byte Arrays

The **XLAT** (Translate) instruction copies a byte from an array of bytes into the AL register. The instruction takes its name from its ability to translate an element's number into the element itself. For example, given the number 7, **XLAT** returns byte #7 from the array. The array may hold byte-sized integers or, very often, a table or list of characters. The syntax for **XLAT** is:

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

The optional **B** suffix (for "byte") reflects the size of data the instruction handles. Both **XLAT** and **XLATB** assemble to exactly the same machine code.

To use **XLAT**, place the offset of the start of the array in the BX register and the desired index value in AL. Array indexes always begin with 0 in assembly language. To retrieve the first byte of the array, set AL to 0; to retrieve the second byte, set AL to 1, and so forth. **XLAT** returns the byte element in AL, overwriting the index number.

By default, the DS register contains the segment of the table, but you can use a segment override to specify a different segment. You need not give an operand except when specifying a segment override. (For information about the segment override operator, see "Direct Memory Operands" in Chapter 3.)

This example illustrates **XLAT** by looking up hexadecimal characters in a list. The code converts 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 char 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
```

Although AL cannot contain an index value greater than 255, you can use **XLAT** with arrays containing more than 256 elements. Simply treat each 256-byte block of the array as a smaller sub-array. For example, to retrieve the 260th element of an array, add 256 to BX and set AL=3 (260-256-1).

Structures and Unions

A structure is a group of possibly dissimilar data types and variables 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 also can 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; 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 6.1 extends the functionality of structures and also makes some changes to MASM 5.1 behavior. If you prefer, you can retain MASM 5.1 behavior by specifying **OPTION OLDSTRUCTS** in your program.

Declaring Structure and Union Types

When you declare a structure or union type, you create a template for data. The template states the sizes and, optionally, the initial values in the structure or union, but allocates no memory.

The **STRUCT** keyword marks the beginning of a type declaration for a structure. (**STRUCT** and **STRUC** are synonyms.) The format for **STRUCT** and **UNION** type declarations is:

name {**STRUCT** | **UNION**} [[*alignment*]] [[,**NONUNIQUE**]]
fielddeclarations
name **ENDS**

The *fielddeclarations* is a series of one or more variable declarations. You can declare default initial values individually or with the **DUP** operator. (See "Defining Structure and Union Variables," following.) "Referencing Structures, Unions, and Fields," later in this chapter, explains the **NONUNIQUE** keyword. You can nest structures and unions, as explained in "Nested Structures and Unions," also later in this chapter.

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. "Defining Structure and Union Variables," following, 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 must be unique within a nesting level because they represent the offset from the beginning of the structure to the corresponding field.

A label elsewhere in the code may have the same name as a structure field, but a text macro cannot. Also, field names between structures need not be unique. Field names must 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 and 32-bit processors but makes no difference in programs 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 /Zp*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*, each field is aligned to either the field's size or the *alignment* value, whichever is less. If the field size in bytes is greater than the alignment value, the field is padded so that its offset is evenly divisible by the alignment value. Otherwise, the field is padded so that its offset is evenly divisible by the field size.

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 aligned field. 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 the assembler determines offsets:

```
STUDENT2    STRUCT  2   ; Alignment value is 2
  score     WORD    1   ; Offset = 0
  id        BYTE    2   ; Offset = 2 (1 byte padding added)
  year      DWORD   3   ; Offset = 4
  sname     BYTE    4   ; Offset = 8 (1 byte padding added)
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, 1 byte of padding is added at the end of `student2`.

```
STUDENT4    STRUCT  4            ; Alignment value is 4
  sname     BYTE    1            ; Offset =  0 (1 byte padding added)
  score     WORD    10 DUP (100) ; Offset =  2
  year      BYTE    2            ; Offset = 22 (1 byte padding
                                 ;   added so offset of next field
                                 ;   is divisible by 4)
  id        DWORD   3            ; Offset = 24
STUDENT4    ENDS
```

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. MASM 6.1 provides an improved H2INC utility, which C programmers can use to translate C structures to assembly. (See *Environment and Tools,* Chapter 20.)

The **ALIGN**, **EVEN**, and **ORG** directives can modify how field offsets are placed during structure definition. The **EVEN** and **ALIGN** directives insert padding bytes to round the field offset up to the specified alignment boundary. The **ORG** directive changes the offset of the next field to a given value, either positive or negative. If you use **ORG** when declaring a structure, you cannot define a structure of that type. **ORG** is useful when accessing existing data structures, such as a stack frame created by a high-level language.

Defining Structure and Union Variables

Once you have declared a structure or union type, you can define variables of that type. 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 you do not provide a name, 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.

You can give an *initializer* for each field. Each initializer must correspond in type with the 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 use the **DUP** operator.

The list of initializers can be broken only after a comma unless you end the line with a continuation character (\). 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 that follows. Angle brackets and curly braces can be intermixed in an initialization as long as they match. This example illustrates the options for initializing lists in structures of type `ITEMS`:

```
ITEMS       STRUCT
  Iname     BYTE      'Item Name'
  Inum      WORD      ?
  UNION     ITYPE                ; UNION keyword appears first
    oldtype BYTE      0          ;   when nested in structure.
    newtype WORD      ?          ;   (See "Nested Structures
  ENDS                           ;   and Unions," following ).
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 example defines—that is, allocates space for—four structures of the `ITEMS` type. The structures are named `Item1` through `Item4`. Each definition requires the angle brackets or curly braces even when not initialized. If you initialize more than one field, separate the values with commas, as shown in `Item3` and `Item4`.

You need not initialize all fields in a structure. If a field is blank, the assembler uses the structure's initial value given for that field in the declaration. If there is no default value, the field value is left unspecified.

For nested structures or unions, 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 20 unions
```

Arrays as Field Initializers

The size of the initializer determines the length of the array that can override the contents of a field in a variable definition. 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 can override any member of type **BYTE** (or **SBYTE**). You need not enclose the string in angle brackets or curly braces unless mixed with other override methods.

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` previously shown. 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' }

; Next line illegal since name in query field is too long:
; info2  INFO    {"TESTFILE", , "DirectoryName"}

lotsof  INFO    { , , 'file1', , {0,0,0} },
                { , , 'file2', , {0,0,1} },
                { , , 'file3', , {0,0,2} }
```

The following diagram 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 because `"DirectoryName"` is longer than the initial string for that field.

Arrays of Structures and Unions

You can define an array of structures using the **DUP** operator (see "Declaring and Referencing Arrays," earlier in this chapter) 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 the following example.

```
Item8   ITEMS    {'Bolts', 126, 10},
                 {'Pliers',139, 10},
                 {'Saws',  414, 10}
```

Redeclaring a Structure

The assembler generates an error when you declare a structure more than once unless the following are the same:

- Field names
- Offsets of named fields
- Initialization lists
- Field alignment value

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. (For information about alignment, see “Declaring Structure and Union Types,” earlier in this chapter.)

This example, using the preceding data declarations, 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
```

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

References to fields must always be fully qualified, with the structure or union names and the dot operator preceding the field name. The assembler requires that you use the dot operator only with structure fields, not as an alternative to the plus operator; nor can you use the plus operator as an alternative to the dot operator.

The following example shows several ways to reference the fields of a structure of type DATE.

```
DATE    STRUCT                          ; Defines structure type
  month BYTE    ?
  day   BYTE    ?
  year  WORD    ?
DATE    ENDS

yesterday       DATE    {1, 20, 1993}     ; 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 only if
                                          ;   month is already a
                                          ;   field in a different
                                          ;   structure
```

Under **OPTION M510** or **OPTION OLDSTRUCTS**, unique structure names do not need to be qualified. However, 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, you must qualify all references to a field. (For information on the **OPTION** directive, see Chapter 1.)

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 unions.

```
WB      UNION
  w     WORD    ?
  b     BYTE    ?
WB      ENDS

array   WB      (100 / SIZEOF WB) DUP ({0})

        mov     array[12].w, 40h
        mov     array[32].b,  2
```

As the preceding code illustrates, you can use unions to access the same data in more than one form. One application of structures and unions is to simplify the task of reinitializing a far pointer. For a far pointer declared as

```
FPWORD  TYPEDEF FAR PTR WORD

        .DATA
WordPtr FPWORD ?
```

you must follow these steps to point `WordPtr` to a word value named `ThisWord` in the current data segment.

```
        mov     WORD PTR WordPtr[2], ds
        mov     WORD PTR WordPtr, OFFSET ThisWord
```

The preceding method requires that you remember whether the segment or the offset is stored first. However, if your program declares a union like this:

```
uptr      UNION
  dwptr   FPWORD   0
  STRUCT
    offs  WORD     0
    segm  WORD     0
  ENDS
uptr      ENDS
```

You can initialize a far pointer with these steps:

```
        .DATA
WrdPtr2 uptr    <>
        .
        .
        .
        mov     WrdPtr2.segm, ds
        mov     WrdPtr2.offs, OFFSET ThisWord
```

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.

Nested Structures and Unions

You can nest structures and unions in several ways. This section explains how to refer to the fields in a nested structure or union. The following example illustrates the four techniques for nesting, and how to reference the fields. Note the syntax for nested structures. The techniques are reviewed following the example.

```
ITEMS           STRUCT
  Inum          WORD    ?
  Iname         BYTE    'Item Name'
ITEMS           ENDS

INVENTORY       STRUCT
  UpDate        WORD    ?
  oldItem       ITEMS   { \
                        100,
                        '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. Because `INVENTORY` contains two structures of type `ITEMS` , the field names in `oldItem` are not unique. Therefore, you must use the full field names when referencing those fields, as 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:

  ```
  mov     yearly.ups.shipmode, 'A'
  ```

- As shown in the `Items` field of `Inventory`, you also can use unnamed variables of existing structures or unions inside another structure or union. In these cases, you can reference fields directly:

  ```
  mov     yearly.Inum, 'C'
  mov     ax, yearly.f1
  ```

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.

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 using 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 it is defined, you can use the record variable 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 some applications of record variables and constants.

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. 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 Chapter 1. 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. *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 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. It allocates no space in the data segment.

```
COLOR   RECORD  blink:1, back:3, intense:1, fore:3
```

The next example creates a record type `CW` that has six fields. Each record declared with this type occupies 16 bits of memory. Initial (default) values are given for each field. You can use them when declaring data 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
```

Defining Record Variables

Once you have declared a record type, you can define record variables of that type. For each variable, the assembler allocates memory 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 previously declared with 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. You must include curly braces or angle brackets even when you do not specify an initial value.

If you use the **DUP** operator (see "Declaring and Referencing Arrays," earlier in this chapter) to initialize multiple record variables, only the angle brackets and

any initial values 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 following example 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 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** returns the number of bytes a record variable occupies. You cannot use **LENGTHOF** with a record declaration, but you can use it with defined record variables. **LENGTHOF** returns the number of records in an array of records, or 1 for a single record variable. The following 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 00100101y

        mov     ax, RGBCOLOR2         ; Equivalent to
                                          ;   "mov ax, 00FFh"
        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"
```

Record Operators

The **WIDTH** operator (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 following example 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
wintens EQU     WIDTH intense       ; "wintens"    = 1
wfore   EQU     WIDTH fore          ; "wfore"      = 3
wcolor  EQU     WIDTH COLOR         ; "wcolor"     = 8
        .CODE
        .
        .
        .
        mov     ah, message         ; Load initial   1101 1001
        and     ah, NOT MASK back   ; Turn off   AND 1000 1111
                                    ; "back"         ---------
                                    ;                1000 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) GT 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
```

The example continues by illustrating several ways in which record fields can serve as operands and expressions:

```
; Rotate "back" of "message" without changing other values

mov     al, message      ; 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 message     ---------
                         ;                           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 message      ---------
                         ;                           0110 0000 ah
or      ah, al           ; Combine old and new    OR 1000 1001 al
                         ;                           ---------
mov     message, 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.

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 80486DX processor as well. It does not apply to the 80486SX, which does not provide an on-chip coprocessor.

This chapter begins with a summary of the directives and formats of floating-point data that you need 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:

- 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. The emulation routines provided with all Microsoft high-level languages enable you to 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 includes the instruction set for these numbers.

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 covers floating-point declarations and floating-point data formats.

Declaring Floating-Point Variables and Constants

You can allocate real constants using the **REAL4**, **REAL8**, and **REAL10** directives. These directives allocate the following floating-point numbers:

Directive	Size
REAL4	Short (32-bit) real numbers
REAL8	Long (64-bit) real numbers
REAL10	10-byte (80-bit) real numbers and BCD numbers

Table 6.1 lists the possible ranges for floating-point variables. The number of significant digits can vary in an arithmetic operation as the least-significant digit may be lost through rounding errors. This occurs regularly for short and long real numbers, so you should assume the lesser value of significant digits shown in Table 6.1. Ten-byte real numbers are much less susceptible to rounding errors for reasons described in the next section. However, under certain circumstances, 10-byte real operations can have a precision of only 18 digits.

Table 6.1 Ranges of Floating-Point Variables

Data Type	Bits	Significant Digits	Approximate Range
Short real	32	6–7	1.18×10^{-38} to 3.40×10^{38}
Long real	64	15–16	2.23×10^{-308} to 1.79×10^{308}
10-byte real	80	19	3.37×10^{-4932} to 1.18×10^{4932}

With versions of MASM prior to 6.0, the **DD**, **DQ**, and **DT** directives could allocate real constants. MASM 6.1 still supports these directives, but the variables are integers rather than floating-point values. Although this makes no difference in the assembly code, CodeView displays the values incorrectly.

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. You can use these numbers as initializers for real-number variables.

The assembler always evaluates digits of real numbers as base 10. It converts real-number constants given in decimal format to a binary format. The sign, exponent, and decimal part 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 end with the real-number designator (R). It cannot be signed. For example, the hexadecimal number `3F800000r` can serve 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
```

The section “Storing Numbers in Floating-Point Format,” following, 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**, see "Defining Pointer Types with TYPEDEF," page 75.

Storing Numbers in Floating-Point Format

The assembler stores floating-point variables in the IEEE format. MASM 6.1 does not support **.MSFLOAT** and Microsoft binary format, which are available in version 5.1 and earlier. Figure 6.1 illustrates the IEEE format for encoding short (4-byte), long (8-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

The following list explains how the parts of a real number are stored in the IEEE format. Each item in the list refers to an item in Figure 6.1.

- Sign bit (0 for positive or 1 for negative) in the upper bit of the first byte.
- 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).

- The integer part of the significand in bit 63 for the 10-byte real format. By absorbing carry values, this bit allows 10-byte real operations to preserve precision to 19 digits. The integer part is always 1 in short and long real numbers; consequently, these formats do not provide a bit for the integer, since there is no point in storing it.
- Decimal part of the significand in the remaining bits. The length is 23 bits for short real numbers, 52 bits for long real numbers, and 63 bits for 10-byte real numbers.

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 real numbers 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 real numbers is 1,023. The bias for 10-byte real numbers is 16,383.

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.

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 can be one of the four formats for using operands explained in "Instruction and Operand Formats," later in this section.

This section describes how the coprocessor transfers data to and from the coprocessor, coordinates processor and coprocessor operations, and controls program flow.

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 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:

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.

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***instruction*	ST, ST(1)	`fadd`
Memory	**F***instruction memory*	ST	`fadd memloc`
Register	**F***instruction* **ST(***num***), ST** **F***instruction* **ST, ST(***num***)**	—	`fadd st(5), st` `fadd st, st(3)`
Register pop	**F***instruction***P ST(***num***), ST**	—	`faddp st(4), st`

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.

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.

ST (the top of the stack) is the source operand in coprocessor arithmetic operations. ST(1), the second register, is the destination. The result of the operation replaces the destination operand, and the source is popped off the stack. This leaves the result at the top of the stack.

The following example illustrates the classical-stack format; 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

Memory Format

Instructions that use 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 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 “Using Coprocessor Instructions,” later in this section). 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 in 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

Register Format

Instructions that use 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 positions of the operands do not change.

The only instructions that use the register operand format are the **FXCH** instruction and arithmetic instructions for calculations on two values. With the **FXCH** instruction, the stack top is implied and need not be specified, as shown in this example and in 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

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 top pops off the stack. The register-pop format is used only for instructions for calculations on two values, as in this example and in 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

Coordinating Memory Access

The math coprocessor and main processor work simultaneously. However, since the coprocessor cannot handle device input or output, data originates in the main processor.

The main processor and the coprocessor have their own registers, which are separate and inaccessible to each other. They 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. The two potential timing conflicts that can occur are handled in different ways.

One timing conflict results from a coprocessor instruction following 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 case shown in the first example that follows.

Another conflict results from a processor instruction that accesses memory following 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 **WAIT** or **FWAIT** (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 smaller and more efficient by using the **.286** or **.386** directive.

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 these operations. For general syntax information, see “Instruction and Operand Formats,” earlier in this section.

Loading and Storing Data

Data-transfer instructions copy 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 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.

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 in the following list.

The math coprocessor offers a few special instructions 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.

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
Instruction(s)	**Description**
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

Doing Arithmetic Calculations

Most of the coprocessor instructions for 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. In most of the arithmetic instructions listed here, the result replaces the destination register. The instructions include:

Instruction	Description
FADD	Adds the source and destination
FSUB	Subtracts the source from the destination
FSUBR	Subtracts the destination from the source
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 illustrates several arithmetic instructions. The code solves quadratic equations, but does no error checking and fails for some values because it attempts to find the square root of a negative number. Both Help and the MATH.ASM sample file show a complete version of this procedure. The complete form uses the **FTST** (Test for Zero) instruction to check for a negative number or 0 before calculating the square root.

```
        .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(b2 - 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       ; = b2
        fsubr               ; = b2 - 4ac
                            ; Negative value here produces error
        fsqrt               ; = square root(b2 - 4ac)
        fld     b           ; Load b
        fchs                ; Make it negative
        fxch                ; Exchange

        fld     st          ; Copy square root
        fadd    st,st(2)    ; Plus version = -b + root(b2 - 4ac)
        fxch                ; Exchange
        fsubp   st(2),st    ; Minus version = -b - root(b2 - 4ac)

        fdiv    st,st(2)    ; Divide plus version
        fstp    posx        ; Store it
        fdivr               ; Divide minus version
        fstp    negx        ; Store it
```

Controlling Program Flow

The math coprocessor has 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 this example 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 “Control Registers,” following.

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’ settings 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. Each instruction in the list has implied operands. Some instructions have a wait version and a no-wait version. The no-wait versions have **N** as the second letter. The instructions for comparing and testing flags include:

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**	Compares the source to ST and sets 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**.

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

The status word register is the only commonly used control register. (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

Using An Emulator Library

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 an emulator from a high-level language. All Microsoft high-level languages come with emulator libraries for all memory models.

To use emulator functions, first write your assembly-language procedure using coprocessor instructions. Then assemble the module with the /FPi option and link it with your high-level–language modules. 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 following example tells the assembler to use emulation mode. This option (introduced in Chapter 1) can be defined only once in a module.

```
OPTION EMULATOR
```

You can use emulator functions in a stand-alone assembler program by assembling with the /Cx command-line option and linking with the appropriate emulator library. The following fragment outlines a small-model program that contains floating-point instructions served by an emulator:

```
        .MODEL  small, c
        OPTION  EMULATOR
        .
        .
        .
        PUBLIC  main
        .CODE
main:                           ; Program entry point must
        .STARTUP                ;   have name 'main'
        .
        fadd    st, st          ; Floating-point instructions
        fldpi                   ;   emulated
```

Emulator libraries do not allow for all of the coprocessor instructions. The following floating-point instructions are not emulated:

FBLD	**FLDENV**	**FSAVE**	**FSTENV**
FBSTP	**FNOP**	**FSAVEW**	**FUCOM**
FCOS	**FPREM1**	**FSAVED**	**FUCOMP**
FDECSTP	**FRSTOR**	**FSETPM**	**FUCOMPP**
FINCSTP	**FRSTORW**	**FSIN**	**FXTRACT**
FINIT	**FRSTORD**	**FSINCOS**	

For information about writing assembly-language procedures for high-level languages, see Chapter 12, "Mixed-Language Programming."

Using Binary Coded Decimal Numbers

Binary coded decimal (BCD) numbers allow calculations on large numbers without rounding errors. This characteristic makes BCD numbers a common choice for monetary calculations. Although BCDs can represent integers of any precision, the 8087-based coprocessors accommodate BCD numbers only in the range ±999,999,999,999,999,999.

This section explains how to define BCD numbers, how to access them with a math coprocessor or emulator, and how to perform simple BCD calculations on the main processor.

Defining BCD Constants and Variables

Unpacked BCD numbers are made up of bytes containing a single decimal digit in the lower 4 bits of each byte. Packed BCD numbers are made up of bytes

containing two decimal digits: one in the upper 4 bits and one in the lower 4 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.

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 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
4 bits. They can be defined using the **BYTE** directive. For example, an unpacked BCD number could be defined and initialized as follows:

```
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
```

As these two lines show, you can arrange digits backward or forward, depending on how you write the calculation routines that handle the numbers.

BCD Calculations on a Coprocessor

As the previous section explains, BCDs differ from other numbers only in the way a program stores them in memory. Internally, a math coprocessor does not distinguish BCD integers from any other type. The coprocessor can load, calculate, and store packed BCD integers up to 18 digits long.

The coprocessor instruction

```
        fbld            bcd1
```

pushes the packed BCD number at `bcd1` onto the coprocessor stack. When your code completes calculations on the number, place the result back into memory in BCD format with the instruction

```
        fbstp           bcd1
```

which discards the variable from the stack top.

BCD Calculations on the Main Processor

The 8086-family of processors can perform simple arithmetic operations on BCD integers, but only one digit at a time. The main processor, like the coprocessor, operates internally on the number's binary value. It requires additional code to translate the binary result back into BCD format.

The main processor provides instructions specifically designed to translate to and from BCD format. These instructions are called "ASCII-adjust" and "decimal-adjust" instructions. They 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.

Unpacked BCD Numbers

When a calculation using two one-digit values produces a two-digit result, the instructions **AAA**, **AAS**, **AAM**, and **AAD** place 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. The four ASCII-adjust instructions for unpacked BCDs are:

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.

For processor arithmetic on unpacked BCD numbers, you must do the 8-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 and always work on the value in the AL register.

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.

Packed BCD Numbers

Packed BCD numbers are made up of bytes containing two decimal digits: one in the upper 4 bits and one in the lower 4 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.

For processor calculations on packed BCD numbers, you must do the 8-bit arithmetic calculations on each byte separately, placing the result in the AL register. After each operation, use the corresponding decimal-adjust instruction to adjust the result. The decimal-adjust instructions do not take an operand and always work on the value in the AL register.

The 8086-family processors provide the instructions **DAA** (Decimal Adjust after Addition) and **DAS** (Decimal Adjust after Subtraction) for adjusting packed BCD numbers after addition and subtraction.

These examples use **DAA** and **DAS** to add and subtract 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 4 bits carries to or borrows from the digit in the upper 4 bits, and it sets the carry flag if the digit in the upper 4 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.

CHAPTER 7

Controlling Program Flow

Very few programs 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 and from procedures. This chapter describes various ways 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 6.1 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 structures that repeat actions or evaluate conditions. It discusses MASM directives, such as **.WHILE** and **.REPEAT**, that generate appropriate compare, loop, and jump instructions for you, and the **.IF**, **.ELSE**, and **.ELSEIF** directives that generate jump instructions.

The "Procedures" section in this chapter explains how to write an assembly-language procedure. It covers the 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 options for the stack-frame setup inside procedures.

The last section explains how to pass program control to an interrupt routine.

Jumps

Jumps are the most direct way to change program control from one location to another. At the processor level, jumps work by changing the value of the IP (Instruction Pointer) register to a target offset and, for far jumps, by changing the CS register to a new segment address. Jump instructions fall into only two categories: conditional and unconditional.

Unconditional Jumps

The **JMP** instruction transfers control unconditionally to another instruction. **JMP**'s single operand contains the address of the target instruction.

Unconditional jumps skip over code that should not be executed, as shown here:

```
; 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 longer the distance, the more bytes the assembler uses to code the instruction. In versions of MASM prior to 6.0, unconditional **NEAR** jumps sometimes generated inefficient code, but MASM can now optimize unconditional jumps.

Jump Optimizing

The assembler determines the smallest encoding possible for the direct unconditional jump. MASM does not require a distance operator, so you do not have to determine the correct distance of the jump. If you specify a distance, it overrides any assembler optimization. If the specified distance falls short of the target address, the assembler generates an error. If the specified distance is longer than the jump requires, the assembler encodes the given distance and does not optimize it.

The assembler optimizes jumps when 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 though it were external.

If these two conditions are met, MASM uses the instruction, distance, and size of the operand to determine how 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.

Indirect Operands

An indirect operand provides a pointer to the target address, rather than the address itself. A pointer is a variable that contains an address. The processor distinguishes indirect (pointer) operands from direct (address) operands by the instruction's context.

You can specify the pointer's size with the **WORD**, **DWORD**, or **FWORD** attributes. Default sizes are based on **.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 **NEAR32** for a 32-bit register:

```
        jmp     bx            ; NEAR16 jump
        jmp     ebx           ; NEAR32  jump
```

A **DWORD** indirect operand, however, is ambiguous to the assembler.

```
jmp     DWORD PTR [var]       ; A NEAR32 jump in a 32-bit segment;
                              ; a FAR16 jump in a 16-bit segment
```

In this case, your code must clear the ambiguity with the **NEAR32** or **FAR16** keywords. The following example shows how to use **TYPEDEF** to define **NEAR32** and **FAR16** pointer types.

```
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 the following example.

```
NPVOID  TYPEDEF NEAR PTR
        .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 instance, the indirect memory operands point to addresses of routines for handling different keystrokes.

Conditional Jumps

The most common way to transfer control in assembly language is to use a conditional jump. This is a two-step process:

1. First test the condition.
2. Then jump if the condition is true or continue if it is false.

All conditional jumps except two (**JCXZ** and **JECXZ**) use the processor flags for their criteria. Thus, any statement that sets or clears a flag can serve as a test basis for a conditional jump. The jump statement can be any one of 30 conditional-jump instructions. A conditional-jump instruction takes a single operand containing the target address. You cannot use a pointer value as a target as you can with unconditional jumps.

Jumping Based on the CX Register

JCXZ and **JECXZ** are special conditional jumps that do not consult the processor flags. Instead, as their names imply, these instructions cause a jump only if the CX or ECX register is zero. The use of **JCXZ** and **JECXZ** with program loops is covered in the next section, "Loops."

Jumping Based on the Processor Flags

The remaining conditional jumps in the processor's repertoire all depend on the status of the flags register. As the following list shows, several conditional jumps have two or three names—**JE** (Jump if Equal) and **JZ** (Jump if Zero), for example. Shared names assemble to exactly the same machine instruction, so you may choose whichever mnemonic seems more appropriate. Jumps that depend on the status of the flags register include:

Instruction	Jumps if
JC/JB/JNAE	Carry flag is set
JNC/JNB/JAE	Carry flag is clear
JBE/JNA	Either carry or zero flag is set
JA/JNBE	Carry and zero flag are clear
JE/JZ	Zero flag is set
JNE/JNZ	Zero flag is clear
JL/JNGE	Sign flag ≠ overflow flag
JGE/JNL	Sign flag = overflow flag
JLE/JNG	Zero flag is set or sign ≠ overflow
JG/JNLE	Zero flag is clear and sign = overflow
JS	Sign flag is set
JNS	Sign flag is clear
JO	Overflow flag is set
JNO	Overflow flag is clear
JP/JPE	Parity flag is set (even parity)
JNP/JPO	Parity flag is clear (odd parity)

The last two jumps in the list, **JPE** (Jump if Parity Even) and **JPO** (Jump if Parity Odd), are useful only for communications programs. The processor sets

the parity flag if an operation produces a result with an even number of set bits. A communications program can compare the flag against the parity bit received through the serial port to test for transmission errors.

The conditional jumps in the preceding list can follow any instruction that changes the processor flags, as these examples show:

```
; 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
        mov     cx, Count       ; First, initialize CX
        jnz     skip            ; If the result is not zero, continue
        call    zhandler        ; Else do special case
```

As the second example shows, the jump does not have to immediately follow the instruction that alters the flags. Since **MOV** does not change the flags, it can appear between the **SUB** instruction and the dependent jump.

There are three categories of conditional jumps:

- Comparison of two values
- Individual bit settings in a value
- Whether a value is zero or nonzero

Jumps Based on Comparison of Two Values

The **CMP** instruction is the most common way to test for conditional jumps. It compares two values without changing either, then sets or clears the processor flags according to the results of the comparison.

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 of the subtraction.

You can compare signed or unsigned values, but you must choose the subsequent conditional jump to reflect the correct value type. For example, **JL** (Jump if Less Than) and **JB** (Jump if Below) may seem conceptually similar, but a failure to understand the difference between them can result in program bugs. Table 7.1 shows the correct conditional jumps for comparisons of signed and unsigned values. The table shows the zero, carry, sign, and overflow flags as ZF, CF, SF, and OF, respectively.

Table 7.1 Conditional Jumps Based on Comparisons of Two Values

Signed Comparisons		Unsigned Comparisons	
Instruction	**Jump if True**	**Instruction**	**Jump if True**
JE	ZF = 1	**JE**	ZF = 1
JNE	ZF = 0	**JNE**	ZF = 0
JG/JNLE	ZF = 0 and SF = OF	**JA/JNBE**	CF = 0 and ZF = 0
JLE/JNG	ZF = 1 or SF ≠ OF	**JBE/JNA**	CF = 1 or ZF = 1
JL/JNGE	SF ≠ OF	**JB/JNAE**	CF = 1
JGE/JNL	SF = OF	**JAE/JNB**	CF = 0

The mnemonic names of jumps always refer to the comparison of **CMP**'s first operand (destination) with the second operand (source). For instance, in this example, **JG** tests whether the first operand is greater than the second.

```
cmp     ax, bx ; Compare AX and BX
jg      next1  ; Equivalent to: If ( AX > BX ) goto next1
jl      next2  ; Equivalent to: If ( AX < BX ) goto next2
```

Jumps Based on Bit Settings

The individual bit settings in a single value can also serve as the criteria for a conditional jump. The **TEST** instruction tests whether specific bits in an operand are on or off (set or clear), and sets the zero flag accordingly.

The **TEST** instruction is the same as the **AND** instruction, except that **TEST** changes neither 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
```

The source operand for **TEST** is often a mask in which the test bits are the only bits set. The destination operand contains the value to be tested. If all the bits set in the mask are clear in the destination operand, **TEST** sets the zero flag. If any of the flags set in the mask are also set in the destination operand, **TEST** clears the zero flag.

The 80386/486 processors provide additional bit-testing instructions. The **BT** (Bit Test) series of instructions copy a specified bit from the destination operand to the carry flag. A **JC** or **JNC** can then route program flow depending on the result. For variations on the **BT** instruction, see the *Reference*.

Jumps Based on a Value of Zero

A program often needs to jump based on whether a particular register contains a value of zero. We've seen how the **JCXZ** instruction jumps depending on the value in the CX register. You can test for zero in other data registers nearly as efficiently with the **OR** instruction. A program can **OR** a register with itself without changing the register's contents, then act on the resulting flags status. For example, the following example tests whether BX is zero:

```
        or      bx, bx          ; Is BX = 0?
        jz      is_zero         ; Jump if so
```

This code is functionally equivalent to:

```
        cmp     bx, 0           ; Is BX = 0?
        je      is_zero         ; Jump if so
```

but produces smaller and faster code, since it does not use an immediate number as an operand. The same technique also lets you test a register's sign bit:

```
        or      dx, dx          ; Is DX sign bit set?
        js      sign_set        ; Jump if so
```

Jump Extending

Unlike an unconditional jump, a conditional jump cannot reference a label more than 128 bytes away. For example, the following statement is valid as long as `target` is within a distance of 128 bytes:

```
; Jump to target less than 128 bytes away
        jz      target          ; If previous operation resulted
                                ;   in zero, jump to target
```

However, if `target` is too distant, the following sequence is necessary to enable a longer jump. Note this sequence is logically equivalent to the preceding example:

```
; 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:
```

MASM can automate jump-extending for you. If you target a conditional jump to a label farther than 128 bytes away, MASM rewrites the instruction with an unconditional jump, which ensures that the jump can reach its target. If `target` lies within a 128-byte range, the assembler encodes the instruction `jz target` as is. Otherwise, MASM generates two substitute instructions:

```
        jne $ + 2 + (length in bytes of the next instruction)
        jmp NEAR PTR target
```

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 less than 128 bytes away.

MASM enables automatic jump expansion by default, but you can turn it off with the **NOLJMP** form of the **OPTION** directive. For information about the **OPTION** directive, see page 24.

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 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 away, so jump extension occurs only for targets greater than that distance.

Anonymous Labels

When you code jumps in assembly language, you must invent many label names. One alternative to continually thinking up new label names is to use anonymous labels, which you can use anywhere in your program. But because anonymous labels do not provide meaningful names, they are best used for jumping over only 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 following example targets an anonymous label:

```
        jge     @F
        .
        .
        .
@@:
```

The items `@B` and `@F` always refer to the nearest occurrences of @@:, so there is never any conflict between different anonymous labels.

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 inside the alternative part of the original **.IF** statement to be evaluated. 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:
```

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 6.1 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 both CX and the state of the zero flag. **LOOPZ** ends when either CX=0 or the zero flag is clear, whichever occurs first. **LOOPNZ** ends when either CX=0 or the zero flag is set, whichever occurs first. **LOOPE** and **LOOPZ** assemble to the same machine instruction, as do **LOOPNE** and **LOOPNZ**. Use whichever mnemonic best fits the context of your loop. 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 or ECX = 0. Unlike other conditional-jump instructions, which can jump to either a near or a short label under the 80386 or 80486, **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 "Conditional Jumps," page 164.

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
```

The **JCXZ** and **JECXZ** instructions provide an efficient way to avoid executing loops when the loop counter CX is empty. For example, consider the following loops:

```
mov     cx, LoopCount           ; Load loop counter
next:   .                       ; Iterate loop CX times
        .
        .
        loop    next            ; Do again
```

If `LoopCount` is zero, CX decrements to -1 on the first pass. It then must decrement 65,535 more times before reaching 0. Use a **JCXZ** to avoid this problem:

```
mov     cx, LoopCount           ; Load loop counter
        jcxz    done            ; Skip loop if count is 0
next:   .                       ; Else iterate loop CX times
        .
        .
        loop    next            ; Do again
done:                           ; Continue after loop
```

Loop-Generating Directives

The high-level control structures generate loop structures for you. These directives are similar to the **while** and **repeat** loops of C or Pascal, and can make your assembly programs easier to code and to read. The assembler generates the appropriate assembly code. These directives are summarized as follows:

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.
.BREAK	End a **.REPEAT** or a **.WHILE** loop unconditionally.
.CONTINUE	Jump unconditionally past any remaining code to bottom of loop.

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 signed or unsigned value, an expression using the binary operators in C (**&&**, ||, or **!**), or the state of a flag. For more information about expression operators, see page 178.

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**.

.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 the contents of 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
```

.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*]]

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 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
```

.BREAK and .CONTINUE Directives

The **.BREAK** and **.CONTINUE** directives 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 ending the loop execution 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 you press ENTER.

```
.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 preceding source code with the /Fl and /Sg command-line options and then view the results in the listing file, you will 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 encodes the jump using the jump extension and jump optimization techniques described in "Unconditional Jumps," page 162, and "Conditional Jumps," page 164.

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.

Expression Operators

The binary relational operators in MASM 6.1 are the same binary operators used in C. These operators generate MASM compare, test, and conditional jump instructions. High-level control instructions include:

Operator	Meaning
==	Equal
!=	Not equal
>	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)`.

You can also use the flag names (**ZERO?**, **CARRY?**, **OVERFLOW?**, **SIGN?**, and **PARITY?**) as operands in conditions with the high-level control structures. For example, in `.WHILE (CARRY?)`, the value of the carry flag determines the outcome of the condition.

Signed and Unsigned Operands

Expression operators generate unsigned jumps by default. However, if either side of the operation is signed, the assembler considers the entire operation signed.

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
```

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 priority. Like expressions in high-level languages, precedence is evaluated left to right.

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 **&&** requires both expressions to be true for the entire condition to be true.

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 similar to functions, subroutines, and procedures in high-level languages such as C, FORTRAN, and Pascal. 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.

Defining Procedures

Procedures require a label at the start of the procedure and a **RET** instruction 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 normally is placed immediately before the **ENDP** directive. The assembler makes sure 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 example code on page 164.

Calls can be near or far. Near calls push only the offset portion of the calling address and therefore must target a procedure within the same segment or group. You can specify the type for the target operand. 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. The assembler then encodes the call appropriately, as it does with unconditional jumps. (See previous "Unconditional Jumps" and "Conditional Jumps.")

MASM optimizes a call to a far non-external 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* **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 "Using Local Variables," following.

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, the popped value is meaningless, and the stack status may cause the execution to return to a random memory location, resulting in program failure.

An extended **PROC** syntax automates many of the details of accessing arguments and saving registers. See "Declaring Parameters with the PROC Directive," later in this chapter.

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 to the procedure in various ways. Although you can pass arguments to a procedure in registers or in variables, the most common method is the stack. Microsoft languages have specific conventions for passing arguments. These conventions for assembly-language modules shared with modules from high-level languages are explained in Chapter 12, "Mixed-Language Programming."

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
        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.

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 following diagram 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 parts of the **PROC** directive include:

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 on the following page.
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 following 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:

Attributes

The syntax for the attributes field is:

[[*distance*]] [[*langtype*]] [[*visibility*]] [[*<prologuearg>*]]

The explanations for these options include:

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. **NEAR** distance is assumed for **TINY, SMALL**, **COMPACT**, and **FLAT**. The assembler assumes **FAR** distance for **MEDIUM**, **LARGE**, and **HUGE**. For 80386/486 programming with 16- and 32-bit segments, you can specify **NEAR16**, **NEAR32**, **FAR16**, or **FAR32**.
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** *num* instruction to end the procedure. The **RET** adjusts the stack upward by *num*, which represents the number of bytes in the argument list. This step, called "cleaning the stack," returns the stack pointer SP to the value it had before the caller pushed any arguments. 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 12).
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. For more information, see Chapter 1, "Using the Option Directive."
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). For an explanation of prologue and epilogue code, see "Generating Prologue and Epilogue Code," later in this chapter.

Parameters

The comma that separates *parameters* from *reglist* is optional, if both fields appear on the same line. If *parameters* appears on a separate line, you must end the *reglist* field with a comma. In the syntax:

parmname [[**:***tag*]]

parmname is the name of the parameter. The *tag* can be 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 "Data Types," Chapter 1. An example showing how to reference **VARARG** parameters appears later in this section. You can nest procedures if they do not have parameters or **USES** register lists. This diagram shows a procedure definition with one parameter definition.

The procedure presented in "Passing Arguments on the Stack," page 182, is here rewritten using the extended **PROC** functionality. Prior to the procedure call, you must push the arguments onto the stack unless you use **INVOKE.** (See "Calling Procedures with INVOKE," later in this chapter.)

```
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. (For more information about using pointers, see Chapter 3, "Using Addresses and Pointers.")

In the following example, even though the procedure declares the parameters as near pointers, you must code two **MOV** instructions to get the values of the param-
eters. 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
```

You can use conditional-assembly directives to make sure 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
```

Using VARARG

In the **PROC** statement, you can append the **:VARARG** keyword to the last parameter to indicate that the procedure accepts a variable number of arguments. However, **:VARARG** applies only to the **C**, **SYSCALL**, or **STDCALL** calling conventions (see Chapter 12). A symbol must precede **:VARARG** so the procedure can access arguments as offsets from the given variable name, as this example illustrates:

```
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     argcount      ; Point to next argument
        inc     si
        inc     si
        .ENDW

        ret                   ; Total is in AX
addup3  ENDP
```

You can pass non-default-sized pointers in the **VARARG** portion of the parameter list by separately passing the segment portion and the offset portion of the address.

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.

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 demonstrated 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. Use local (stack) data when you have a large amount of temporary data for the procedure.

To use a local variable, you must save stack space for it at the start of the procedure. A procedure can then reference the variable by its position in the stack. At the end of the procedure, you must clean the stack by restoring the stack pointer. This effectively throws away all local variables and regains the stack space they occupied.

This example subtracts 2 bytes from the SP register to make room for a local word variable, then accesses the variable 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 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 changes 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 "Passing Arguments on the Stack," page 182, in which the calling code adjusts the stack for the argument.

Figure 7.2 shows the stack at key points in the process.

Figure 7.2 Local Variables on the Stack

Creating Local Variables Automatically

MASM's **LOCAL** directive automates the process for creating local variables on the stack. **LOCAL** frees you from having to count stack words, and it makes your code easier to write and maintain. This section illustrates the advantages of creating temporary data with the **LOCAL** directive.

To use the **LOCAL** directive, 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 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. If you have programmed before in a high-level language that allows scoping, local variables will seem familiar. For example, a C compiler sets up variables with automatic storage class in the same way as the **LOCAL** directive.

We can simplify the procedure in the previous section 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 with 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.
qualifiedtype	A simple MASM type or a type defined with other types and attributes. For more information, see "Data Types" in Chapter 1.

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. Alternatively, you can use several consecutive **LOCAL** directives.

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 of BP, and they are not visible outside the procedure. In the following 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.

Declaring Procedure Prototypes

MASM provides the **INVOKE** directive to handle many of the details important to procedure calls, such as pushing parameters according to the correct calling conventions. To use **INVOKE**, the procedure called must have been declared previously 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.

Declaring procedure prototypes is good programming practice, but is optional. Prototypes in MASM perform the same function as prototypes in C 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 usually are placed at the beginning of an assembly program or in a separate include file so the assembler encounters the prototype before the actual procedure.

Prototypes enable the assembler to check for unmatched parameters and are especially useful for procedures called from other modules and other languages. 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. For more information about using include files, see Chapter 8, "Sharing Data and Procedures among Modules and Libraries."

The **PROTO** directive provides one way to define a procedure prototype. The syntax for a prototype definition is the same as for a procedure declaration (see "Declaring Parameters with the PROC Directive," earlier in this chapter), 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. However, if they are 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** is to write your 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 following example illustrates how to define and then declare two typical procedures. In both prototype and declaration, the comma before the argument list is optional only when the list does not appear on a separate line:

```
; 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 reports an error or converts the type to the expected type. These conversions are explained in the next section.

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 are changed. **INVOKE** generates procedure calls and automatically:

- Converts arguments to the expected types.
- Pushes arguments on the stack in the correct order.
- Cleans 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 the procedure uses **VARARG**, **INVOKE** can pass a number of arguments different from the number in the parameter list without generating an error or warning. Any additional arguments must be at the end of the **INVOKE** argument list. All other arguments must match those in the prototype parameter list.

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 later in this chapter.)

Procedures with these 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
```

can be called with **INVOKE** statements 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.

Widening Arguments

For **INVOKE** to correctly handle type conversions, you must use the signed data types for any signed assignments. MASM widens an argument to match the type expected by a procedure's parameters in these cases:

Type Passed	Type Expected
BYTE, **SBYTE**	**WORD**, **SWORD**, **DWORD**, **SDWORD**
WORD, **SWORD**	**DWORD**, **SDWORD**

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.

Detecting Errors

If the assembler needs to widen an argument, it first copies the value to AL or AX. It widens an unsigned value by placing a zero in the higher register area, and widens a signed value with a **CBW**, **CWD**, or **CWDE** instruction as required. Similarly, the assembler copies a constant argument value into AL or AX when the **.8086** directive is in effect. You can see these generated instructions in the listing file when you include the /Sg command-line option.

Using the accumulator register to widen or copy an argument may lead to an error if you attempt to pass AX as another argument. For example, consider the following **INVOKE** statement for a procedure with the C calling convention

```
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 `arg` like this:

```
mov     al, DGROUP:arg
xor     ah, ah
push    ax
```

The generated code thus overwrites the last argument (AX) passed to the procedure. The assembler generates an error in this case, requiring you to rewrite the **INVOKE** statement.

To summarize, the **INVOKE** directive overwrites AX and perhaps DX when widening arguments. It also uses AX to push constants on the 8088 and 8086. If you use these registers (or EAX and EDX on an 80386/486) to pass arguments, they may be overwritten. The assembler's error detection prevents this from ever becoming a run-time bug, but AX and DX should remain your last choice for holding arguments.

Invoking Far Addresses

You can pass a **FAR** pointer in a *segment*::*offset* pair, as shown in the following. Note the use of double colons to separate the register pair. The registers could be any other register pair, including a pair that an MS-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, **INVOKE** cannot combine into a single address one argument for the segment and one for the offset.

Passing an Address

You can use the **ADDR** operator to pass the address of an expression to a procedure that expects 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
```

For information on defining pointers with **TYPEDEF,** see "Defining Pointer Types with TYPEDEF" in Chapter 3.

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
FUNCPTR         TYPEDEF PTR FUNCPROTO

        .DATA
pfunc   FUNCPTR OFFSET proc1, OFFSET proc2

        .CODE
        .
        .
        .
        mov     bx, OFFSET pfunc          ; BX points to table
        mov     si, Num                   ; Num contains 0 or 2
        INVOKE  FUNCPTR PTR [bx+si], arg1 ; Call proc1 if Num=0
                                          ;   or proc2 if Num=2
```

You can also use **ASSUME** to accomplish the same task. The following **ASSUME** statement associates the type `FUNCPTR` with the BX register.

```
        ASSUME  BX:FUNCPTR
        mov     bx, OFFSET pfunc
        mov     si, Num
        INVOKE  [bx+si], arg1
```

Checking the Code Generated

Code generated by the **INVOKE** directive 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.

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 or label after the **PROC** directive. This means you cannot label the prologue for the purpose of jumping to it. The assembler generates the epilogue code when it encounters a **RET** or **IRET** instruction. Using the assembler-generated prologue and epilogue code saves 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 to generate the prologue or epilogue code. 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.

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.

The prologue code consists of three steps:

1. Point BP to top of stack.
2. Make space on stack for local variables.
3. Save registers the procedure must preserve.

The epilogue cancels these three steps in reverse order, then cleans the stack, if necessary, with a **RET** *num* instruction. For example, the procedure declaration

```
myproc  PROC NEAR PASCAL USES di si,
        arg1:WORD, arg2:WORD, arg3:WORD
        LOCAL local1:WORD, local2:WORD
```

generates the following prologue code:

```
        push    bp              ; Step 1:
        mov     bp, sp          ;   point BP to stack top
        sub     sp, 4           ; Step 2: space for 2 local words
        push    di              ; Step 3:
        push    si              ;   save registers listed in USES
```

The corresponding epilogue code looks like this:

```
        pop     si              ; Undo Step 3
        pop     di
        mov     sp, bp          ; Undo Step 2
        pop     bp              ; Undo Step 1
        ret     6               ; Clean stack of pushed arguments
```

Notice the **RET 6** instruction cleans the stack of the three word-sized arguments. The instruction appears in the epilogue because the procedure does not use the C calling convention. If `myproc` used C conventions, the epilogue would end with a **RET** instruction without an operand.

The assembler generates standard epilogue code when it encounters a **RET** instruction without an operand. It does not generate an epilogue if **RET** has a nonzero operand. To suppress generation of a standard epilogue, use **RETN** or **RETF** with or without an operand, or use **RET 0**.

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**.

For example, adding **LOADDS** to the argument list for `myproc` creates this prologue:

```
push    bp              ; Step 1:
mov     bp, sp          ;   point BP to stack top
sub     sp, 4           ; Step 2: space for 2 locals
push    ds              ; Save DS and point it
mov     ax, DGROUP      ;   to DGROUP, as
mov     ds, ax          ;   instructed by LOADDS
push    di              ; Step 3:
push    si              ;   save registers listed in USES
```

The epilogue code restores DS:

```
pop     si              ; Undo Step 3
pop     di
pop     ds              ; Restore DS
mov     sp, bp          ; Undo Step 2
pop     bp              ; Undo Step 1
ret     6               ; Clean stack of pushed arguments
```

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 run in place 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 Windows. 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

Your macro must have formal parameters to match all the actual arguments passed. The arguments passed to your macro include:

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 is generated as a result of an **IRET** instruction and cleared if the epilogue is generated as a result of a **RET** instruction.
	Bits 9–15	Undefined (not necessarily zero).
parmbytes	The accumulated count in bytes of all 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. Enclose this list with angle brackets (< >) and separate each item with commas. Reverse the list 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 calling any procedures that use the prologue:

```
OPTION PROLOGUE:ProfilePro
```

If you define either a prologue or an epilogue macro, the assembler uses the standard prologue or epilogue code 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.

For additional information about writing macros, see Chapter 9, “Using Macros.” The PROLOGUE.INC file provided in the MASM 6.1 distribution disks can create the prologue and epilogue sequences for the Microsoft C professional development system.

MS-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. For example, you may want to replace an MS-DOS interrupt handler, such as the Critical Error (Interrup 24h) and CONTROL+C (Interrupt 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. For additional information about MS-DOS and BIOS interrupts, see Chapter 11, “Writing Memory-Resident Software.”

Calling MS-DOS and ROM-BIOS Interrupts

Interrupts provide a way to access MS-DOS and ROM-BIOS from assembly language. They are called with the **INT** instruction, which takes an immediate value between 0 and 255 as its only operand.

MS-DOS and ROM-BIOS interrupt routines accept data through registers. For instance, most MS-DOS routines (and many BIOS routines) require a function number in the AH register. Many handler routines also return values in registers. To use an interrupt, you must know what data the handler routine expects and what data, if any, it returns. For information, consult Help or one of the other references mentioned in the Introduction.

The following fragment illustrates a simple call to MS-DOS Function 9, which displays the string `msg` on the screen:

```
        .DATA
msg     BYTE    "This writes to the screen$"
        .CODE
        mov     ax, SEG msg     ; Necessary only if DS does not
        mov     ds, ax          ;   already point to data segment
        mov     dx, offset msg  ; DS:DX points to msg
        mov     ah, 09h         ; Request Function 9
        int     21h
```

When the **INT** instruction executes, the processor:

1. Looks up the address of the interrupt routine in the Interrupt Vector Table. This table starts at the lowest point in memory (segment 0, offset 0) and consists of a series of far pointers called vectors. Each vector comprises a 4-byte address (segment:offset) pointing to an interrupt handler routine. The table sequence implies the number of the interrupt the vector references: the first vector points to the Interrupt 0 handler, the second vector to the Interrupt 1 handler, and so forth. Thus, the vector at 0000:*i**4 holds the address of the handler routine for Interrupt *i*.
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), in that order. (The current instruction is the one following the **INT** statement.) As with a **CALL**, this ensures control returns to the next logical position in the program.
4. Jumps to the address of the interrupt routine, as specified in the Interrupt Vector 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

Replacing an Interrupt Routine

To replace an existing interrupt routine, your program must:

- Provide a new routine to handle the interrupt.
- Replace the old routine's address in the Interrupt Vector Table with the address of your new routine.
- Replace the old address back into the vector table before your program ends.

You can write an interrupt routine as 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 You can use the full extended **PROC** syntax (described in "Declaring Parameters with the PROC Directive," earlier in this chapter) 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 restore a register list in interrupt procedures.

The **IRET** instruction in MASM 6.1 has two forms that suppress epilogue code. 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 shows how to replace the handler for Interrupt 4. Once registered in the Interrupt Vector Table, the new routine takes control when the processor encounters either an **INT 4** instruction or its special variation **INTO** (Interrupt on Overflow). **INTO** is a conditional instruction that acts only when the overflow flag is set. With **INTO** after a numerical calculation, your code can automatically route control to a handler routine if the calculation results in a numerical overflow. 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 another set of instructions.

The following example program first executes **INT 21h** to invoke MS-DOS Function 35h (Get Interrupt Vector). This function returns the existing vector for Interrupt 4. The program stores the vector, then invokes MS-DOS Function 25h (Set Interrupt Vector) to place the address of the `ovrflow` procedure in the Interrupt Vector Table. From this point on, `ovrflow` gains control whenever the processor executes **INTO** while the overflow flag is set. The new routine displays a message and returns with AX and DX set to 0.

```
        .MODEL LARGE, C
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
        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
        pop     ds                      ; Restore
        .
        .
        .
        add     ax, bx                  ; Do arithmetic
        into                            ; Call Interrupt 4 if overflow
        .
        .
        .
        lds     dx, vector              ; Load original address
        mov     ax, 2504h               ; Restore it to vector table
        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
        cwd                             ; Set DX to 0
        iret                            ; Return
ovrflow ENDP
        END
```

Before the program ends, it again uses MS-DOS Function 25h to reset the original Interrupt 4 vector back into the Interrupt Vector Table. This reestablishes the original routine as the handler for Interrupt 4.

The first instruction of the `ovrflow` routine warrants further discussion. When the processor encounters an **INT** instruction, it clears the interrupt flag before branching to the specified interrupt handler routine. The interrupt flag serves a crucial role in smoothing the processor's tasks, but must not be abused. When clear, the flag inhibits hardware interrupts such as the keyboard or system timer. It should be left clear only briefly and only when absolutely necessary. Unless you have a

compelling reason to leave the flag clear, always include an **STI** (Set Interrupt Flag) instruction at the beginning of your interrupt handler routine to reenable hardware interrupts.

CLI (Clear Interrupt Flag) and its corollary **STI** are designed to protect small sections of time-dependent code from interruptions by the hardware. If you use **CLI** in your program, be sure to include a matching **STI** instruction as well. The sample interrupt handlers in Chapter 11, "Writing Memory-Resident Software," illustrate how to use these important 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 provides 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.

This chapter explains the two data-sharing methods MASM 6.1 offers. The first method simplifies data sharing between modules with include files. The second does not involve include files. Instead, this method allows modules to share procedures and data items using the **PUBLIC** and **EXTERN** directives.

The last section of this chapter explains how to create program libraries and access their routines.

Selecting Data-Sharing Methods

If data defined in one module is to be used in other modules of a program, you must declare the data public and external. MASM provides several ways to do this:

- Declare a symbol public with the **PUBLIC** directive in the module where it is defined. This makes the symbol available to other modules. You must also place an **EXTERN** statement for that symbol in all other modules that refer to the public symbol. This statement informs the assembler that the symbol is external—that is, defined in another module.
- Declare the data communal with the **COMM** directive. However, communal variables have limitations. You cannot depend on their location in memory because they are allocated by the linker, and they cannot be initialized.

The **EXTERNDEF** directive 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. For more information on **PUBLIC** and **EXTERN**, see "Using Alternatives to Include Files," page 219.

Sharing Symbols with 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 are optional, but can simplify project organization by eliminating the need to insert common declarations into all modules of a program. An alternative to using include files is described in "Using Alternatives to Include Files," page 219.

This section explains how to organize symbol definitions and the declarations that make them global (available to all modules); how to make both variables and procedures public with **EXTERNDEF**, **PROTO**, and **COMM**.; and where to place these directives in the modules and include files.

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. This ensures consistency and saves time when you update programs. Include files should contain only symbol declarations and any other declarations that are resolved at assembly time. (For a list of assembly-time operations, see "Generating and Running Executable Programs" in Chapter 1.)

If more than one module accesses the include file, the file cannot contain statements that define and allocate memory for symbols. Otherwise, the assembler would attempt to allocate the same symbol more than once.

Note An include file used in two or more modules should not allocate data variables.

Modules

An **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 module and continues the assembly.

The **INCLUDE** directive takes the form:

INCLUDE *filename*

where *filename* is the full name of the include file. For example, the following declaration inserts the contents of the include file SCREEN.INC in your program:

```
INCLUDE SCREEN.INC
```

The filename in the **INCLUDE** directive must be fully specified; no extensions are assumed. If a full pathname is not given, the assembler first searches 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*

You can include more than one /I option on the command line. The assembler then searches for include files within each specified path in the order given. If none of these directories contains the 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 /x command-line option tells the assembler to ignore the INCLUDE environment variable when searching for include files.)

An include file may specify another include file. The assembler processes the second include file before returning to the first. Your program can nest include files this way as deeply as the amount of free memory allows.

Include Files or Modules

You can use the **EQU** directive to create named constants that cannot be redefined in your program. (For information about the **EQU** directive, see "Integer Constants and Constant Expressions," page 11.) 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 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 "Controlling the Segment Order,"
page 47.

Declaring Symbols Public and External

It is sometimes useful to make certain procedures and variables (such as 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. This section describes how to make variables and procedures global using the **EXTERNDEF**, **PROTO**, or **COMM** declarations within include files.

When a procedure is defined in one module and called in another module, it must be declared public in the defining module and external in the calling module(s). MASM offers three ways to declare a procedure public and external:

- Use the **PUBLIC** directive in the defining module and **EXTERN** in all other modules that reference the procedure. The **PUBLIC** and **EXTERN** directives are explained on page 220.
- Declare the procedure with **EXTERNDEF**.
- Prototype the procedure with the **PROTO** directive.

Using EXTERNDEF

MASM treats **EXTERNDEF** as a public declaration in the defining module, and as an external declaration in the referencing module(s). You can use the **EXTERNDEF** statement in your include file to make a variable common to 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 referencing module. It is ignored in modules that neither define nor reference 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 "Data Types," page 14.

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. For information on selecting the appropriate *langtype* type, see "Naming and Calling Conventions," page 308.

The following diagram 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. For more information, see "Positioning External Declarations," following.

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. Use **EXTERNDEF** with procedures in the same way as variables:

```
EXTERNDEF MyProc:FAR            ; Declare far procedure external
```

You can also use **EXTERNDEF** to make a code label global between modules so that one module can reference a label in another module. Give the label global scope with the double colon operator, like this:

```
EXTERNDEF codelabel:NEAR
.
.
.
codelabel::
```

Another module can reference `codelabel` like this:

```
EXTERNDEF codelabel:NEAR
.
.
.
        jmp     codelabel
```

Using PROTO

This section describes how to prototype a procedure with the **PROTO** directive. **PROTO** automatically issues an **EXTERNDEF** for the procedure unless the **PROC** statement declares the procedure **PRIVATE**. Defining a prototype enables type-checking for the procedure arguments.

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** in one module. The **PROC** directive declares the procedure **PUBLIC** by default.
3. Call the procedure with the **INVOKE** statement (or with **CALL**). Make sure that all calling modules access the include file.

For descriptions, syntax, and examples of **PROTO**, **PROC**, and **INVOKE,** see Chapter 7, "Controlling Program Flow."

The following example illustrates these three steps. In the example, a **PROTO** statement defines 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

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.

COMM declares a data variable external and instructs the linker to allocate 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**). If you do not provide a memory model with the .**MODEL** directive, you must specify a distance when accessing a communal variable, like this:

```
mov     ax, NEAR PTR CommNear
mov     bx, FAR PTR CommFar
```

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 on 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 (explained in the previous section) is more flexible than **COMM** because you can initialize variables defined with it, and your code can rely on the position and sequence of the defined data.

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, 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 any segments opened in include files so that external declarations following an include statement are not incorrectly placed inside a segment. If you want to be certain an external definition lies 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
```

For information about predefined symbols such as **@CurSeg**, see "Predefined Symbols," page 10.

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 referencing 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.

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 "Positioning External Declarations," previous.

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**. For more information on specifying *langtype* types, see "Declaring Parameters with the PROC Directive," page 184, and "Naming and Calling Conventions," page 308.

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*}

"Data Types," page 14, describes *qualifiedtype*. You can use the **ABS** (absolute) keyword 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:

Figure 8.3 Using PUBLIC and EXTERN

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.

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 file, recognizable by its .LIB extension, usually contains related routines. For example, you might place string-manipulation functions in one library, matrix calculations in another, and port communications in another. Do not place communal variables (defined with the **COMM** directive) in a library.

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, the linker adds the entire module to your program, but not the entire library.

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.

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 you place this statement in the source file DRAW.ASM and GRAPHICS.LIB is in the same directory, you can assemble and link the program with the following command:

```
ML DRAW.ASM
```

Without the **INCLUDELIB** directive, you must link the program DRAW.ASM with either of the following commands:

```
ML DRAW.ASM GRAPHICS.LIB
ML DRAW /link GRAPHICS
```

If you want to assemble and link separately, type

```
ML /c DRAW.ASM
LINK DRAW,,,GRAPHICS
```

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 specified by the LIB environment variable.

The LIB.EXE utility helps you create, organize, and maintain run-time libraries. Refer to *Environment and Tools* for instructions on LIB.EXE.

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 linker 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):PROC
        .
        .
        .
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 that force linkage with FLOAT.OBJ, the linker will use the alternate name for `init(dummy)`.

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.

This chapter explains how to use macros for simple code substitutions and 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.

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 **TEXTEQU** directive defines a text macro, as these examples show:

name **TEXTEQU** *<text>*
name **TEXTEQU** *macroId | textmacro*
name **TEXTEQU** *%constExpr*

In the previous lines, *text* is a sequence of characters enclosed in angle brackets, *macroId* is a previously defined macro function, *textmacro* is a previously defined text macro, and *%constExpr* is an expression that evaluates to text.

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
```

The first line assigns text to the symbol `msg`. The second line equates the text of the `msg` text macro with a new text macro called `string`. The third line assigns new text to `msg`. Although `msg` has new text, `string` retains its 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, as discussed on page 248), the resulting text macro will expand recursively.

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
```

Macro Procedures

If your program must perform the same task many times, you can avoid repeatedly typing the same statements each time by writing a macro procedure. Think of macro procedures (commonly called macros) as text-processing mechanisms that automatically generate repeated text.

This section uses the term “macro procedure” rather than “macro” when necessary to distinguish between a macro procedure and a macro function. Macro functions are described in “Returning Values with Macro Functions.”

Conforming to common usage, this chapter occasionally speaks of “calling” a macro, a term that deserves further scrutiny. It’s natural to think of a program calling a macro procedure in the same way it calls a normal subroutine procedure, because they seem to perform identically. However, a macro is simply a representative for real code. Wherever a macro name appears in your program, so in reality does all the code the macro represents. A macro does not cause the processor to vector off to a new location as does a normal procedure. Thus, the expression “calling a macro” may imply the effect, but does not accurately describe what actually occurs.

Creating Macro Procedures

You can define a macro procedure without parameters by placing 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. You could define a `beep` macro as follows:

```
beep    MACRO
    mov  ah, 2          ;; Select DOS Print Char function
    mov  dl, 7          ;; Select ASCII 7 (bell)
    int  21h            ;; Call DOS
ENDM
```

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 the macro is referenced and expanded. Listings are usually easier to read if the comments aren’t repeatedly expanded. However, regular comments (those with a single semicolon) are listed in macro expansions. See Appendix C for listing files and examples of how macros are expanded in listings.

Once you define a macro, 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
```

During assembly, the instructions in the macro replace the macro reference. The listing file shows:

```
                          .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 with the **CALL** instruction.

Many such tasks can be handled as either a macro or a procedure. In deciding which method to use, you must choose between speed and size. For repetitive tasks, a procedure produces smaller code, because the instructions physically appear only once in the assembled program. However, each call to the procedure involves the additional overhead of a **CALL** and **RET** instruction. Macros do not require a change in program flow and so execute faster, but generate the same code multiple times rather than just once.

Passing Arguments to Macros

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. You cannot use reserved words as parameter names unless you disable the keyword with **OPTION NOKEYWORD**. You must also set the compatibility mode with **OPTION M510** or the /Zm command-line option.

To pass arguments to a macro, place the arguments after the macro name when you call the macro:

macroname arglist

The assembler treats as one item all text between matching quotation marks in an *arglist*.

The `beep` macro introduced in the previous section used the MS-DOS interrupt to write only the bell character (ASCII 7). We can rewrite the macro with a parameter that accepts any character:

```
writechar MACRO char
    mov  ah, 2                  ;; Select DOS Print Char function
    mov  dl, char               ;; Select ASCII char
    int  21h                    ;; Call DOS
ENDM
```

Whenever it expands the macro, the assembler replaces each instance of `char` with the given argument value. The rewritten macro now writes any character to the screen, not just ASCII 7:

```
        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 page 242). If you pass fewer arguments than the macro procedure expects, the assembler assigns empty strings to the remaining parameters (unless you have specified default values). This may cause errors. For example, a reference to the `writechar` macro with no argument results in the following line:

```
        mov     dl,
```

The assembler generates an error for the expanded statement but not for the macro definition or the macro call.

You can make macros more flexible by leaving off arguments or adding additional arguments. The next section tells some of the ways your macros can handle missing or extra arguments.

Specifying Required and Default Parameters

Macro parameters can have special attributes to make them more flexible and improve error handling. You can make parameters required, give them default values, or vary their number. Variable parameters are used almost exclusively with the **FOR** directive, so are covered in "FOR Loops and Variable-Length Parameters," later in this chapter.

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 reference. **REQ** can thus improve error reporting.

You can also accommodate missing parameters by specifying a default value, like this:

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
```

If a reference to this macro does not include the argument `char`, the assembler fills in the blank with the default value of 7 and the macro beeps when called.

Enclose the default parameter value in angle brackets so the assembler recognizes the supplied value as a text value. This is explained in detail in "Text Delimiters and the Literal-Character Operator," later in this chapter.

Missing arguments can also be handled with the **IFB**, **IFNB**, **.ERRB**, and **.ERRNB** directives. They are described in the section "Conditional Directives" in chapter 1 and in Help. Here is a slightly more complex macro that uses some of these techniques:

```
Scroll MACRO distance:REQ, attrib:=<7>, 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 7 (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 shows 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.

Defining Local Symbols in Macros

You can make a symbol local to a macro by identifying 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 outside the macro.

You must declare as local any labels within a macro, since a label can occur only once in the source. The **LOCAL** directive makes a special instance of the label each time the macro appears. This prevents redefinition of the label when expanding the macro. It also allows you to reuse the label elsewhere in your code.

You must declare all local symbols immediately following the **MACRO** statement (although blank lines and comments may precede the local symbol). Separate each symbol with a comma. You can attach comments to the **LOCAL** statement and list multiple **LOCAL** statements in the macro. 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.

You should avoid using anonymous labels in macros (see "Anonymous Labels" in Chapter 7). Although legal, they can produce unwanted results if you expand a macro near another anonymous label. For example, consider what happens in the following:

```
Update  MACRO arg1
@@: .
    .
    .
    loop @B
ENDM
        .
        .
        .
        jcxz    @F
        Update  ax
@@:
```

Expanding `Update` places another anonymous label between the jump and its target. The line

```
        jcxz    @F
```

consequently jumps to the start of the loop rather than over the loop—exactly the opposite of what the programmer intended.

Assembly-Time Variables and Macro Operators

In writing macros, you will often assign and modify values assigned to symbols. Think of these symbols 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 the assembler expands a macro, it processes the symbols in the order shown here. MASM first replaces macro parameters with the text of their actual arguments, then expands text macros.

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.

Text Delimiters and the Literal-Character Operator

The angle brackets (< >) are text delimiters. A text value is usually delimited when assigning a text macro. You can do this with **TEXTEQU**, as previously shown, or with the **SUBSTR** and **CATSTR** directives discussed in "String Directives and Predefined Functions," later in this chapter.

By delimiting the text of macro arguments, you can pass text that includes spaces, commas, semicolons, and other special characters. The following example expands a macro called `work` in two different ways:

```
work    <1, 2, 3, 4, 5> ; Passes one argument with 13 chars,
                        ;   including commas and spaces
work    1, 2, 3, 4, 5   ; Passes five arguments, each
                        ;   with 1 character
```

The literal-character operator (**!**) lets you include angle brackets as part of a delimited text value, so the assembler does not interpret them as delimiters. The assembler treats the character following **!** literally rather than as a special character, like this:

```
errstr  TEXTEQU <Expression !> 255>  ; errstr = "Expression > 255"
```

Text delimiters also have a special use with the **FOR** directive, as explained in "FOR Loops and Variable-Length Parameters," later in this chapter.

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 in the following.

Converting Numeric Expressions to Text

The expansion operator can convert numbers to text. The 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, as the following lines show. Notice how you can enclose expressions with parentheses to make them more readable:

```
a       TEXTEQU <3 + 4>         ; a = "3 + 4"
b       TEXTEQU %3 + 4          ; b = "7"
c       TEXTEQU %(3 + 4)        ; c = "7"
```

When assigning text macros, you can use numeric equates in the constant expressions, but not text macros:

```
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 gives you flexibility when passing arguments to macros. It lets you pass a computed value rather than the literal text of an expression. The following example illustrates by defining a macro

```
work    MACRO   arg
    mov ax, arg * 4
ENDM
```

which accepts different arguments:

```
        work    2 + 3           ; Passes "2 + 3"
                                ; Code: mov ax, 2 + (3 * 4)
        work    %2 + 3          ; Passes 5
                                ; Code: mov ax, 5 * 4
        work    2 + num         ; Passes "2 + num"
        work    %2 + num        ; Passes "6"
        work    2 + numstr      ; Passes "2 + numstr"
        work    %2 + numstr     ; Passes "6"
```

You must consider operator precedence when using the expansion operator. Parentheses inside the macro can force evaluation in a desired order:

```
work    MACRO   arg
    mov ax, (arg) * 4
ENDM

        work    2 + 3           ; Code: mov ax, (2 + 3) * 4
        work    %2 + 3          ; Code: mov ax, (5) * 4
```

Several other uses for the expansion operator are reviewed in "Returning Values with Macro Functions," later in this chapter.

Expansion Operator as 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**, which 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. Consider this example:

```
    ECHO    Bytes per element: %(SIZEOF array / LENGTHOF array)
```

Instead of evaluating the expression, this line 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 simply by 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 expansion operator at the start of a line:

```
; Assume memmod, lang, and os specified 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>
```

Substitution Operator

References to a parameter within a macro can sometimes be ambiguous. In such cases, the assembler may not expand the argument as you intend. The substitution operator (**&**) lets you identify unambiguously any parameter within a macro.

As an example, consider the following macro:

```
errgen  MACRO   num, msg
    PUBLIC  errnum
    errnum BYTE    "Error num: msg"
ENDM
```

This macro is open to several interpretations:

- Is `errnum` a distinct word or the word `err` next to the parameter `num`?
- Should `num` and `msg` within the string be treated literally as part of the string or as arguments?

In each case, the assembler chooses the most literal interpretation. That is, it treats `errnum` as a distinct word, and `num` and `msg` as literal parts of the string.

The substitution operator can force different interpretations. If we rewrite the macro with the **&** operator, it looks like this:

```
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 now generates this code:

```
        PUBLIC  err5
err5    BYTE    "Error 5: Unreadable disk"
```

When it encounters the **&** operator, the assembler interprets subsequent text as a parameter name until the next **&** or until the next separator character (such as a space, tab, or comma). Thus, the assembler correctly parses the expression `err&num` because `num` is delimited by **&** and a space. The expression could also be written as `err&num&`, which again unambiguously identifies `num` as a parameter.

The rule also works in reverse. You can delimit a parameter reference with **&** at the end rather than at the beginning. For example, if `num` is 5, the expression `num&12` resolves to "512."

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 to substitute a text macro 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”
```

You can also use the substitution operator in lines beginning with the expansion operator (**%**) symbol, even outside macros (see page 236). It may be necessary to use the substitution operator to paste text macro names to adjacent characters or symbol names, as shown here:

```
text    TEXTEQU <var>
value   TEXTEQU %5
%       ECHO    textvalue is text&&value
```

This echoes the message

```
textvalue is var5
```

Macro substitution always occurs before evaluation of the high-level control structures. The assembler may therefore mistake a bit-test operator (**&**) in your macro for a substitution operator. You can guarantee the assembler correctly recognizes a bit-test operator by enclosing its operands in parentheses, as shown here:

```
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
```

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 MASM 5.1 or earlier, you can specify the old behavior by using **OLDMACROS** or **M510** with the **OPTION** directive (see page 24).

Defining Repeat Blocks with Loop Directives

A “repeat block” is an unnamed macro defined with a loop directive. The loop directive generates the statements inside the repeat block a specified number of times or until a given condition becomes true.

MASM provides several loop directives, which let you specify the number of loop iterations in different ways. Some loop directives can also accept arguments for each iteration. Although the number of iterations is usually specified in the directive, you can use the **EXITM** directive to exit 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. Since repeat blocks are macros themselves, they end with the **ENDM** directive.

This section explains the following four loop directives: **REPEAT**, **WHILE**, **FOR**, and **FORC**. In versions of MASM prior to 6.0, **REPEAT** was called **REPT**, **FOR** was called **IRP**, and **FORC** was called **IRPC**. MASM 6.1 recognizes the old names.

The assembler evaluates repeat blocks on the first pass only. You should therefore avoid using address spans as loop counters, as in this example:

```
REPEAT  (OFFSET label1 - OFFSET label2) ; Don't do this!
```

Since the distance between two labels may change on subsequent assembly passes as the assembler optimizes code, you should not assume that address spans remain constant between passes.

Note The **REPEAT** and **WHILE** directives should not be confused with the **REPEAT** and **WHILE** directives (see "Loop-Generating Directives" in Chapter 7), which generate loop and jump instructions for run-time program control.

REPEAT Loops

REPEAT 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 expands 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
```

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.

The following repeat block uses the **WHILE** directive to allocate variables initialized to calculated values. This is a common technique for generating lookup tables. (A lookup table is any list of precalculated results, such as a table of interest payments or trigonometric values or logarithms. Programs optimized for speed often use lookup tables, since calculating a value often takes more time than looking it up in a table.)

```
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
```

FOR Loops and Variable-Length Parameters

With the **FOR** directive you can iterate through a list of arguments, working on each of them in turn. It has the following syntax:

FOR *parameter*, *<argumentlist>*
statements
ENDM

The *parameter* is a placeholder that represents the name of each argument inside the **FOR** block. The argument list must contain comma-separated arguments and must always be enclosed in angle brackets. Here's an example of a **FOR** block:

```
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 2 bytes initialized to 2, the next 3 bytes initialized to 3, and so on.

The argument list is given specifically in this example, 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 identify angle brackets as characters, not delimiters. See "Text Delimiters (<>) and the Literal-Character Operator," earlier in this chapter.

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 be last. If this macro is called with the statement

```
work 4, , 6, 7, a, b
```

the first argument is received as the value 4, 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. You can modify the `show` macro 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 calling the macro with

```
        show  ‘O’,, ‘K’, 13, 10
```

inserts the default character, a space, for the blank argument.

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 versions of MASM earlier than 6.0, **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. The new predefined macro functions and string processing directives discussed in the following section are usually more efficient for these tasks.

String Directives and Predefined Functions

The assembler provides four directives for manipulating text:

Directive	Description
SUBSTR	Assigns part of string to a new symbol.
INSTR	Searches for one string within another.
SIZESTR	Determines the size of a string.
CATSTR	Concatenates one or more strings to a single string.

These directives assign a processed value to a text macro or numeric equate. For example, the following lines

```
num     =       7
newstr  CATSTR  <3 + >, %num, < = > , %3 + num ; "3 + 7 = 10"
```

assign the string `"3 + 7 = 10"` to `newstr`. **CATSTR** and **SUBSTR** assign text in the same way as the **TEXTEQU** directive. **SIZESTR** and **INSTR** assign a number in the same way as the **=** operator. The four string directives take only text values as arguments. Use the expansion operator (**%**) when you need to make sure that constants and numeric equates expand to text, as shown in the preceding lines.

Each of the string directives has a corresponding predefined macro function version: **@SubStr**, **@InStr**, **@SizeStr**, and **@CatStr**. Macro functions are similar to the string directives, but you must enclose their arguments in parentheses. Macro functions return text values and can appear in any context where text is expected. The following section, "Returning Values with Macro Functions," tells how to write your own macro functions. The following example is equivalent to the previous **CATSTR** example:

```
num     =       7
newstr  TEXTEQU @CatStr( <3 + >, %num, < = > , %3 + num )
```

Macro functions are often more convenient than their directive counterparts because you can use a macro function as an argument to a string directive or to another macro function. Unlike string directives, predefined macro function names are case sensitive when you use the /Cp command-line option.

Each string directive and predefined function acts on a string, which can be any *textItem*. The *textItem* can be text enclosed in angle brackets (< >), the name of a text macro, or a constant expression preceded by **%** (as in **%constExpr**). Refer to Appendix B, "BNF Grammar," for a list of types that *textItem* can represent.

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 the symbol *name*. The *start* parameter specifies the position in *string*, beginning with 1, to start the substring. The *length* gives the length of the substring. If you do not specify *length*, **SUBSTR** returns the remainder of the string, including the *start* character.

INSTR

name **INSTR** [[*start*,]] *string*, *substring*
@InStr([[*start*]], *string*, *substring* **)**

The **INSTR** directive searches a specified *string* for an occurrence of *substring* and assigns its position number to *name*. The search is case sensitive. The *start* parameter is the position in *string* to start the search for *substring*. If you do not specify *start*, it is assumed to be position 1, the start of the string. If **INSTR** does not find *substring*, it assigns position 0 to *name*.

The **INSTR** directive assigns the position value *name* as if it were a numeric equate. In contrast, the **@InStr** returns the value 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> )
```

You can also assign the return value to a text macro, like this:

```
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 returns a length of zero. The **SIZESTR** directive assigns the size value to a name as if it were a numeric equate. The **@SizeStr** function returns the value 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 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 pushes and pops one set of registers, illustrating 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
    LOCAL reg
    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 reg
    %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 `reg` text macro is used only in `RestoreRegs`. It is declared **LOCAL** so it won't take the name `reg` from the global name space. The MACROS.INC file provided with MASM 6.1 includes expanded versions of these same two macros.

Returning Values with Macro Functions

A macro function is a named group of statements that returns a value. When calling a macro function, you must enclose its argument list in parentheses, even if the list is empty. The function always returns text.

MASM 6.1 provides several predefined macro functions for common tasks. The predefined macros include **@Environ** (see page 10) and the string functions **@SizeStr**, **@CatStr**, **@SubStr**, and **@InStr** (discussed in the preceding section).

You define macro functions in exactly the same way as macro procedures, except that a macro function always returns a value through 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 here:

```
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 assembler sees the value returned by the macro function 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. A macro function must first convert a numeric value—such as a constant, a numeric equate, or the result of a numeric expression—before returning it. The macro function can use angle brackets or the expansion operator (**%**) to convert numbers to text. The `DEFINED` macro, for instance, could have returned its value as

```
EXITM   %-1
```

Here is another example of a macro function that 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 define data, as shown here:

```
var     WORD    factorial( 4 )
```

This statement initializes `var` with the number 24 (the factorial of 4).

Using Macro Functions with Variable-Length Parameter Lists

You can use the **FOR** directive to handle macro parameters with the **VARARG** attribute. "FOR Loops and Variable-Length Parameters," page 242, 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.

For example, the following macro function determines the number of arguments in a **VARARG** parameter:

```
@ArgCount MACRO arglist:VARARG
    LOCAL count
    count = 0
    FOR arg, <arglist>
        count = count + 1       ;; Count the arguments
    ENDM
    EXITM %count
ENDM
```

You can use `@ArgCount` inside a macro that has a **VARARG** parameter, as shown here:

```
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 the 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
```

You can use `@ArgI` like this:

```
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
```

Here is an example showing `@ArgRev` in use:

```
work    MACRO   args:VARARG
%   FOR  arg, @ArgRev( <args> )   ;; Process in reverse order
        ECHO    arg
    ENDM
ENDM
```

These three macro functions appear in the MACROS.INC include file, located on one of the MASM distribution disks.

Expansion Operator in Macro Functions

This list summarizes the behavior of the expansion operator (**%**) with macro functions.

- If a macro function is preceded by a **%**, it will be expanded. However, if it expands to a text macro or a macro function call, it will not expand further.
- If you use a macro function call as an argument for another macro function call, a **%** is not needed.
- If a macro function is called inside angle brackets and is preceded by **%**, it will be expanded.

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.

Defining Macros within Macros

Macros can define other macros, a technique called “nesting macros.” MASM expands macros as it encounters them, so nested macros are always processed in nesting order. You cannot reference a nested macro directly in your program, since the assembler begins expansion from the outer macro. In effect, a nested macro is local to the macro that defines it. Only the amount of available memory limits the number of macros a program can nest.

The following example demonstrates how one macro can define another. The macro takes as an argument the name of a shift or rotate instruction, then creates another macro that simplifies the instruction for 8088/86 processors.

```
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
```

Recall that the 8086 processor allows only 1 or CL as an operand for shift and rotate instructions. Expanding `shifts` generates a macro for the shift instruction that uses whichever operand is more efficient. You create the entire series of macros, one for each shift instruction, like this:

```
         ; Call macro repeatedly 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
```

Then use the new macros as replacements for shift instructions, like this:

```
        shrs    ax, 5
        rols    bx, 3
```

Testing for Argument Type and Environment

Macros can expand conditional blocks of code by testing for argument type with the **OPATTR** operator. **OPATTR** returns a single word constant that indicates the type and scope of an expression, like this:

OPATTR *expression*

If *expression* is not valid or is forward-referenced, **OPATTR** returns a 0. Otherwise, the return value incorporates the bit flags shown in the table below.

OPATTR serves as an enhanced version of the **.TYPE** operator, which returns only the low byte (bits 0–7) shown in the table. Bits 11–15 of the return value are undefined.

Bit	Set If expression
0	References a code label
1	Is a memory variable or has a relocatable data label
2	Is an immediate value
3	Uses direct memory addressing
4	Is a register value
5	References no undefined symbols and is without error
6	Is relative to SS
7	References an external label
8–10	Has the following language type: • 000—No language type • 001—C • 010—SYSCALL • 011—STDCALL • 100—Pascal • 101—FORTRAN • 110—Basic

A macro can use **OPATTR** to determine if an argument is a constant, a register, or a memory operand. With this information, the macro can conditionally generate the most efficient code depending on argument type.

For example, given a constant argument, a macro can test it for 0. Depending on the argument's value, the code can select the most effective method to load the value into a register:

```
IF CONST
mov     bx, CONST    ; If CONST > 0, move into BX
ELSE
sub     bx, bx       ; More efficient if CONST = 0
ENDIF
```

The second method is faster than the first, yet has the same result (with the byproduct of changing the processor flags).

The following macro illustrates some techniques using **OPATTR** by loading an address into a specified offset register:

```
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 also can generate different code depending on the assembly environment. The predefined text macro **@Cpu** returns a flag 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
```

Another macro can now use `pushc` rather than conditionally testing for processor type itself. Although either case produces the same code, using `pushc` assembles faster because the environment is checked only once.

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
ASSUMES	Saves segment register information
RADIX	Saves current default radix
LISTING	Saves listing and CREF information
CPU	Saves current CPU and processor
ALL	All of the above

Using Recursive Macros

Macros can call themselves. In MASM 5.1 and earlier, recursion is an important technique for handling variable arguments. MASM 6.1 handles variable arguments much more cleanly with the **FOR** directive and the **VARARG** attribute, as described in “FOR Loops and Variable-Length Parameters,” earlier in this chapter. However, recursion is still available and may be useful for some macros.

CHAPTER 10

Writing a Dynamic-Link Library For Windows

The Windows operating system relies heavily on service routines and data contained in special libraries called "dynamic-link libraries," or DLLs for short. Most of what Windows comprises, from the collections of screen fonts to the routines that handle the graphical interface, is provided by DLLs. MASM 6.1 contains tools that you can use to write DLLs in assembly language. This chapter shows you how.

DLLs do not run under MS-DOS. The information in this chapter applies only to Windows, drawing in part on the chapter "Writing a Module-Definition File" in *Environment and Tools*. The acronym API, which appears throughout this chapter, refers to the application programming interface that Windows provides for programs. For documentation of API functions, see the *Programmer's Reference, Volume 2* of the Windows Software Development Kit (SDK).

The first section of this chapter gives an overview of DLLs and their similarities to normal libraries. The next section explores the parts of a DLL and the rules you must follow to create one. The third section applies this information to an example DLL.

Overview of DLLs

A dynamic-link library is similar to a normal run-time library. Both types of libraries contain a collection of compiled procedures, which serve one or more calling modules. To link a normal library, the linker copies the required functions from the library file (which usually has a .LIB extension) and combines them with other modules to form an executable program in .EXE format. This process is called static linking.

In dynamic linking, the library functions are not copied to an .EXE file. Instead, they reside in a separate file in executable form, ready to serve any calling program, called a "client." When the first client requires the library, Windows takes care of loading the functions into memory and establishing linkage. If

subsequent clients also need the library, Windows dynamically links them with the proper library functions already in memory.

Loading a DLL

How Windows loads a DLL affects the client rather than the DLL itself. Accordingly, this section focuses on how to set up a client program to use a DLL. Since the client can itself be a DLL, this is information a DLL programmer should know. However, MASM 6.1 does not provide all the tools required to create a stand-alone program for Windows. To create such a program, called an "application," you must use tools in the Windows SDK.

Windows provides two methods for loading a dynamic-link library into memory:

Method	Description
Implicit loading	Windows loads the DLL along with the first client program and links it before the client begins execution.
Explicit loading	Windows does not load the DLL until the first client explicitly requests it during execution.

When you write a DLL, you do not need to know beforehand which of the two methods will be used to load the library. The loading method is determined by how the client is written, not the DLL.

Implicit Loading

The implicit method of loading a DLL offers the advantage of simplicity. The client requires no extra programming effort and can call the library functions as if they were normal run-time functions. However, implicit loading carries two constraints:

- The name of the library file must have a .DLL extension.
- You must either list all DLL functions the client calls in the IMPORTS section of the client's module-definition file, or link the client with an import library.

An import library contains no executable code. It consists of only the names and locations of exported functions in a DLL. The linker uses the locations in the import library to resolve references to DLL functions in the client and to build an executable header. For example, the file LIBW.LIB provided with MASM 6.1 is the import library for the DLL files that contain the Windows API functions.

The IMPLIB utility described in *Environment and Tools* creates an import library. Run IMPLIB from the MS-DOS command line like this:

IMPLIB *implibfile dllfile*

where *implibfile* is the name of the import library you want to create from the DLL file *dllfile*. Once you have created an import library from a DLL, link it with a client program that relies on implicit loading, but does not list imported functions in its module-definition file. Continuing the preceding example, here's the link step for a client program that calls library procedures in the DLL *dllfile*:

LINK *client*.OBJ, *client*.EXE, , *implibfile*, *client*.DEF

This simplified example creates the client program *client*.EXE, linking it with the import library *implibfile*, which in turn was created from the DLL file *dllfile*.

To summarize implicit loading, a client program must either

- List DLL functions in the IMPORTS section of its module-definition file, or
- Link with an import library created from the DLL.

Implicit loading is best when a client always requires at least one procedure in the library, since Windows automatically loads the library with the client. If the client does not always require the library service, or if the client must choose at run time between several libraries, you should use explicit loading, discussed next.

Explicit Loading

To explicitly load a DLL, the client does not require linking with an import library, nor must the DLL file have an extension of .DLL. Explicit loading involves three steps in which the client calls Windows API functions:

1. The client calls **LoadLibrary** to load the DLL.
2. The client calls **GetProcAddress** to obtain the address of each DLL function it requires.
3. When finished with the DLL, the client calls **FreeLibrary** to unload the DLL from memory.

The following example fragment shows how a client written in assembly language explicitly loads a DLL called SYSINFO.DLL and calls the DLL function **`GetSysDate`**.

```
        INCLUDE windows.inc
        .DATA
hInstance       HINSTANCE 0
szDLL           BYTE    'SYSINFO.DLL', 0
szDate          BYTE    'GetSysDate', 0
lpProc          DWORD   0
```

```
.CODE
.
.
.
INVOKE  LoadLibrary, ADDR szDLL         ; Load SYSINFO.DLL
mov     hInstance, ax                   ; Save instance count
INVOKE  GetProcAddress, ax, ADDR szDate ; Get and save
mov     lpProc, ax                      ;   far address of
mov     lpProc[2], dx                   ;   GetSysDate
call    lpProc                          ; Call GetSysDate
.
.
.
INVOKE  FreeLibrary, hInstance  ; Unload SYSINFO.DLL
```

For simplicity, the above example contains no error-checking code. An actual program should check all values returned from the API functions.

The explicit method of loading a DLL requires more programming effort in the client program. However, the method allows the client to control which (if any) dynamic-link libraries to load at run time.

Searching for a DLL File

To load a DLL, whether implicitly or explicitly, Windows searches for the DLL file in the following directories in the order shown:

1. The current directory
2. The Windows directory, which contains WIN.COM
3. The Windows system directory, which contains system files such as GDI.EXE
4. The directory where the client program resides (except Windows 3.0 and earlier)
5. Directories listed in the PATH environment string
6. Directories mapped in a network

If Windows does not locate the DLL in any of these directories, it prompts the user with a message box.

Building a DLL

A DLL has additional programming requirements beyond those for a normal run-time library. This section describes the requirements pertaining to the library's code, data, and stack. It also discusses the effects of the library's extension name.

DLL Code

The code in a DLL consists of exported and nonexported functions. Exported functions, listed in the **EXPORTS** section of the module-definition file, are public routines serving clients. Nonexported functions provide private, internal support for the exported procedures. They are not visible to a client.

Under Windows, an exported library routine must appear to the caller as a far procedure. Your DLL routines can use any calling convention you wish, provided the caller assumes the same convention. You can think of dynamic-link code as code for a normal run-time library with the following additions:

- An entry procedure
- A termination procedure
- Special prologue and epilogue code

Entry Procedure

A DLL, like any Windows-based program, must have an entry procedure. Windows calls the entry procedure only once when it first loads the DLL, passing the following information in registers:

- DS contains the library's data segment address.
- DI holds the library's instance handle.
- CX holds the library's heap size in bytes.

Note Windows API functions destroy all registers except DI, SI, BP, DS, and the stack pointer. To preserve the contents of other registers, your program must save the registers before an API call and restore them afterwards.

This information corresponds to the data provided to an application. Since a DLL has only one occurrence in memory, called an "instance," the value in DI is not usually important. However, a DLL can use its instance handle to obtain resources from its own executable file.

The entry procedure does not need to record the address of the data segment. Windows automatically ensures that each exported routine in the DLL has access to the library's data segment, as explained in "Prologue and Epilogue Code," on page 264.

The heap size contained in CX reflects the value provided in the **HEAPSIZE** statement of the module-definition file. You need not make an accurate guess in the **HEAPSIZE** statement about the library's heap requirements, provided you specify a moveable data segment. With a moveable segment, Windows automatically allocates more heap when needed. However, Windows can

provide no more heap in a fixed data segment than the amount specified in the **HEAPSIZE** statement. In any case, a library's total heap cannot exceed 64K, less the amount of static data. Static data and heap reside in the same segment.

Windows does not automatically deallocate unneeded heap while the DLL is in memory. Therefore, you should not set an unnecessarily large value in the **HEAPSIZE** statement, since doing so wastes memory.

The entry procedure calls the Windows API function **LocalInit** to allocate the heap. The library must create a heap before its routines call any heap functions, such as **LocalAlloc**. The following example illustrates these steps:

```
DLLEntry PROC FAR PASCAL PUBLIC         ; Entry point for DLL

        jcxz    @F                      ; If no heap, skip
        INVOKE  LocalInit, ds, 0, cx    ; Else set up the heap
        .IF     ( ax )                  ; If successful,
        INVOKE  UnlockSegment, -1       ;   unlock the data segment
@@:     call    LibMain                 ; Call DLL's data init routine
        mov     ax, TRUE                ; Return AX = 1 if okay,
        .ENDIF                          ;   else if LocalInit error,
        ret                             ;   return AX = 0

DLLEntry ENDP
```

This example code is taken from the DLLENTRY.ASM module, contained in the LIB subdirectory on one of the MASM 6.1 distribution disks. After allocating the heap, the procedure calls the library's initialization procedure—called **LibMain** in this case. **LibMain** initializes the library's static data (if required), then returns to **DLLEntry**, which returns to Windows. If Windows receives a return value of 0 (FALSE) from **DLLEntry**, it unloads the library and displays an error message.

The process is similar to the way MS-DOS loads a terminate-and-stay-resident program (TSR), described in the next chapter. Both the DLL and TSR return control immediately to the operating system, then wait passively in memory to be called.

The following section explains how a DLL gains control when Windows unloads it from memory.

Termination Procedure

Windows maintains a DLL in memory until the last client program terminates or explicitly unloads the library. When unloading a DLL, Windows first calls the library's termination procedure. This allows the DLL to return resources and do any necessary cleanup operations before Windows unloads the library from memory.

Libraries that have registered window procedures with **RegisterClass** need not call **UnregisterClass** to remove the class registration. Windows does this automatically when it unloads the library.

You must name the library's termination procedure `WEP` (for Windows Exit Procedure) and list it in the **EXPORTS** section of the library's module-definition file. To ensure immediate operation, provide an ordinal number and use the **RESIDENTNAME** keyword, as described in the chapter "Creating Module-Definition Files" in *Environment and Tools.* This keeps the name "WEP" in the Windows-resident name table at all times.

Besides its name, the code for `WEP` should also remain constantly in memory. To ensure this, place `WEP` in its own code segment and set the segment's attributes as **PRELOAD FIXED** in the **SEGMENTS** statement of the module-definition file. Thus, your DLL code should use a memory model that allows multiple code segments, such as medium model. Since a termination procedure is usually short, keeping it resident in memory does not burden the operating system.

The termination procedure accepts a single parameter, which can have one of two values. These values are assigned to the following symbolic constants in the WINDOWS.INC file located in the LIB subdirectory:

- `WEP_SYSTEM_EXIT` (value 1) indicates Windows is shutting down.
- `WEP_FREE_DLL` (value 0) indicates the library's last client has terminated or has called **FreeLibrary**, and Windows is unloading the DLL.

The following fragment provides an outline for a typical termination procedure:

```
WEP     PROC FAR PASCAL EXPORT
        wExitCode:WORD

        Prolog                              ; Prologue macro,
        .IF     wExitCode == WEP_FREE_DLL   ;   discussed below
        .                                   ; Get ready to
        .                                   ;   unload
        .
        ELSEIF  wExitCode == WEP_SYSTEM_EXIT
        .                                   ; Windows is
        .                                   ;   shutting down
        .
        . ENDIF                             ; If neither value,
                                            ;   take no action
        mov     ax, TRUE                    ; Always return AX = 1
        Epilog                              ; Epilogue code,
        ret                                 ;   discussed below

WEP     ENDP
```

Usually, the WEP procedure takes the same actions regardless of the parameter value, since in either case Windows will unload the DLL.

Under Windows 3.0, the WEP procedure receives stack space of about 256 bytes. This allows the procedure to unhook interrupts, but little else. Any other action, such as calling an API function, usually results in an unrecoverable application error because of stack overflow. Later versions of Windows provide at least 4K of stack to the WEP procedure, allowing it to call many API functions.

However, WEP should not send or post a message to a client, because the client may already be terminated. The WEP procedure should also not attempt file I/O, since only application processes—not DLLs—can own files. When control reaches WEP, the client may no longer exist and its files are closed.

Prologue and Epilogue Code

Exported procedures in a Windows-based program require special epilogue and prologue code. (For a definition of these terms, see "Generating Prologue and Epilogue Code" in Chapter 7.) The SAMPLES subdirectory on one of the MASM 6.1 distribution disks contains macros you can use for far procedures in your Windows-based programs. Here's a listing of the prologue macro:

```
Prolog  MACRO
        mov     ax, ds          ; Must be 1st, since Windows overwrites
        nop                     ; Placeholder for 3rd byte
        inc     bp              ; Push odd BP.  Not required, but
        push    bp              ;   allows CodeView to recognize frame
        mov     bp, sp          ; Set up stack frame to access params
        push    ds              ; Save DS
        mov     ds, ax          ; Point DS to DLL's data segment
        ENDM
```

The instruction

```
        inc     bp
```

marks the beginning of the stack frame with an odd number. This allows real-mode Windows to locate segment addresses on the stack and update the addresses when it moves or discards the corresponding segments. In protected mode, selector values do not change when segments are moved, so marking the stack frame is not required. However, certain debugging applications, such as Microsoft Codeview for Windows and the Microsoft Windows 80386 Debugger (both documented in *Programming Tools* of the SDK), search for a marked frame to determine if the frame belongs to a far procedure. Without the mark, these debuggers give meaningless information when backtracing through the stack. Therefore, you should include the **INC BP** instruction for Windows-

based programs that may run in real mode or that require debugging with a Microsoft debugger.

Another characteristic of the prologue macro may seem puzzling at first glance. The macro moves DS into AX, then AX back into DS. This sequence of instructions lets Windows selectively overwrite the prologue code in far procedures. When Windows loads a program, it compares the names of far procedures with the list of exported procedures in the module-definition file. For procedures that do not appear on the list, Windows leaves their prologue code untouched. However, Windows overwrites the first 3 bytes of all exported procedures with

```
        mov     ax, DGROUP
```

where DGROUP represents the selector value for the library's data segment. This explains why the prologue macro reserves the third byte with a **NOP** instruction. The 1-byte instruction serves as padding to provide a 3-byte area for Windows to overwrite.

The epilogue code returns BP to normal, like this:

```
Epilog  MACRO
        pop     ds              ; Recover original DS
        pop     bp              ;   and BP+1
        dec     bp              ; Reset to original BP
        ENDM
```

DLL Data

A DLL can have its own local data segment up to 64K. Besides static data, the segment contains the heap from which a procedure can allocate memory through the **LocalAlloc** API function. You should minimize static data in a DLL to reserve as much memory as possible for temporary allocations. Furthermore, all procedures in the DLL draw from the same heap space. If more than one procedure in the library accesses the heap, a procedure should not hold allocated space unnecessarily at the expense of the other procedures.

A Windows-based program must reserve a "task header" in the first 16 bytes of its data segment. If you link your program with a C run-time function, the C startup code automatically allocates the task header. Otherwise, you must explicitly reserve and initialize the header with zeros. The sample program described in "Example of a DLL:SYSINFO," page 267, shows how to allocate a task header.

DLL Stack

A DLL does not declare a stack segment and does not allocate stack space. A client program calls a library's exported procedure through a simple far call, and the stack does not change. The procedure is, in effect, part of the calling program, and therefore uses the caller's stack.

This simple arrangement differs from that used in small and medium models, in which many C run-time functions accept near pointers as arguments. Such functions assume the pointer is relative to the current data segment. In applications, the call works even if the argument points to a local variable on the stack, since DS and SS contain the same segment address.

However, in a DLL, DS and SS point to different segments. Under small and medium models, a library procedure must always pass pointers to static variables located in the data segment, not to local variables on the stack.

When you write a DLL, include the **FARSTACK** keyword with the **.MODEL** directive, like this:

```
.MODEL  small, pascal, farstack
```

This informs the assembler that SS points to a segment other than DGROUP. With full segment definitions, also add the line:

```
ASSUME  DS:DGROUP, SS:NOTHING
```

DLL Extension Names

You can name an explicitly-loaded DLL file with any extension. The many files in your Windows directory with extensions such as .DRV and .FON are almost certainly DLLs. Many DLLs have an .EXE extension, though they are not true executable files.

A library with an .EXE extension should always include stub code, specified by the **STUB** statement in the module-definition file. The stub code activates when run under MS-DOS, usually displaying a message to inform the user that the program requires Windows. Without the stub code, the system hangs if a user attempts to run a DLL with an .EXE extension.

Do not name a DLL with a .COM extension, since MS-DOS will give control to the first byte of the program header. The header does not contain executable instructions, and the system will hang even if the DLL has stub code.

Summary

Following is a summary of the previous information in this chapter.

- A dynamic-link library has only one instance—that is, it can load only once during a Windows session.
- A single DLL can service calls from many client programs. Windows takes care of linkage between the DLL and each client.
- Windows loads a DLL either implicitly (along with the first client) or explicitly (when the first client calls **LoadLibrary**). It unloads the DLL when the last client either terminates or calls **FreeLibrary**.
- A client calls a DLL routine as a simple far procedure. The routine can use any calling convention.
- Windows ensures that the first instruction in a DLL procedure moves the address of the library's data segment into AX. You must provide the proper prologue code to allow space for this 3-byte instruction and to copy AX to DS.
- All procedures in a DLL have access to a single common data segment. The segment contains both static variables and heap space, and cannot exceed 64K.
- A DLL procedure uses the caller's stack.
- All exported procedures in a DLL must appear in the **EXPORTS** list in the library's module-definition file.

Example of a DLL: SYSINFO

Like any library, a DLL should be as small and fast as possible—a good argument for writing it in assembly language. This section describes an example library called SYSINFO, written entirely in assembly language. The following text applies previous information in this chapter to an actual DLL.

SYSINFO contains three callable procedures. The acronym ASCIIZ refers to a string of ASCII characters terminated with a zero. The callable procedures are:

Procedure	Description
GetSysTime	Returns a far pointer to a 12-byte ASCIIZ string containing the current time in *hh:mm:ss* format.
GetSysDate	Returns a far pointer to an ASCIIZ string containing the current date in any of six languages.
GetSysInfo	Returns a far pointer to a structure containing the following system data: • ASCIIZ string of Windows version • ASCIIZ string of MS-DOS version • Current keyboard status • Current video mode

- Math coprocessor flag
- Processor type
- ASCIIZ string of ROM-BIOS release date

To see SYSINFO in action, follow the steps below. The file SYSDATA.EXE resides in the SAMPLES\WINDLL subdirectory of MASM if you requested example files when installing MASM. Otherwise, you must first install the file with the MASM 6.1 SETUP utility.

- Create SYSINFO.DLL as described in the following section and place it in the SAMPLES\WINDLL subdirectory for MASM 6.1.
- From the Windows File Manager, make the SAMPLES\WINDLL subdirectory the current directory.
- In the Program Manager, choose Run from the File menu and type

  SYSDATA

  to run the example program SYSDATA.EXE. This program calls the routines in SYSINFO.DLL and displays the returned data.

Entry Routine for SYSINFO

SYSINFO links with the DLLENTRY module, which serves as the library's entry point when Windows first loads the program. For a listing and description of DLLENTRY.ASM, see the previous section, "Entry Procedure."

DLLENTRY replaces the LIBENTRY module provided with the Windows SDK, but unlocks the data segment after calling the API function **InitTask**. LIBENTRY does not unlock the segment. DLLENTRY saves some space over LIBENTRY, because it does not pass any arguments to `LibMain`.

The `LibMain` procedure handles the library's initialization tasks. You can name the procedure whatever you want, provided you make the same change in

DLLENTRY.ASM and reassemble both modules. You can even combine DLLENTRY with `LibMain` to form one procedure, like this:

```
DLLInit PROC FAR PASCAL PUBLIC          ; Entry point for DLL

        jcxz    @F                      ; If no heap, skip
        INVOKE  LocalInit, ds, 0, cx    ; Else set up the heap
        .IF     ( ax )                  ; If successful,
        INVOKE  UnlockSegment, -1       ;   unlock the data segment
@@:     .                               ; Initialize DLL data. This
        .                               ;   replaces the call to the
        .                               ;   LibMain procedure.
        mov     ax, TRUE                ; Return AX = 1 if okay,
        .ENDIF                          ;   else if LocalInit error,
        ret                             ;   return AX = 0

DLLInit ENDP
END    DLLInit
```

Whatever you call your combined procedure (`DLLInit` in the preceding example), place the name on the **END** statement as shown. This identifies the procedure as the one that first executes when Windows loads the DLL.

SYSINFO accommodates several international languages. Currently, SYSINFO recognizes English, French, Spanish, German, Italian, and Swedish, but you can easily extend the code to include other languages. `LibMain` calls **GetProfileString** to determine the current language, then initializes the variable `indx` accordingly. The variable indirectly points to an array of strings containing days and months in different languages. The `GetSysDate` procedure uses these strings to create a full date in the correct language.

Static Data

SYSINFO stores the strings in its static data segment. This data remains in memory along with the library's code. All procedures have equal access to the data segment.

Because the library does not call any C run-time functions, it explicitly allocates the low paragraph of the data segment with the variable `TaskHead`. This 16-byte area serves as the required Windows task header, described in "DLL Data," earlier in this chapter.

Module-Definition File

The library's module-definition file, named SYSINFO.DEF, looks like this:

```
LIBRARY         SYSINFO
DESCRIPTION     'Sample assembly-language DLL'
EXETYPE         WINDOWS
CODE            PRELOAD MOVEABLE DISCARDABLE
DATA            PRELOAD MOVEABLE SINGLE
SEGMENTS CODE2  PRELOAD FIXED
EXPORTS         WEP             @1      RESIDENTNAME
                GetSysTime      @2
                GetSysDate      @3
                GetSysInfo      @4
```

Note the following points about the module-definition file:

- The **LIBRARY** statement identifies SYSINFO as a dynamic-link library.
- SYSINFO places its termination procedure `WEP` in a separate code segment, called CODE2, which the **SEGMENTS** statement declares as **FIXED**. This keeps the `WEP` routine fixed in memory, while all other code remains moveable.
- The **EXPORTS** section lists all procedures the library exports, including `WEP`.
- None of the library's procedures require heap space, so SYSINFO.DEF includes no **HEAPSIZE** statement.

Assembling and Linking SYSINFO

The following listing shows the description file for SYSINFO:

```
sysinfo.obj:  sysinfo.asm dll.inc
  ML /c /W3 sysinfo.asm
dllentry.obj:  dllentry.asm dll.inc.
  ML /c /W3 dllentry.asm
sysinfo.dll: dllentry.obj sysinfo.obj
  LINK dllentry sysinfo, sysinfo.dll,, libw.lib mnocrtdw.lib,
sysinfo.def
```

To create SYSINFO.DLL, run the **NMAKE** utility described in Chapter 16 of *Environments and Tools*. Or assemble and link SYSINFO with the three command lines shown in the preceding listing. This does not require running **NMAKE**.

SYSINFO links with the library modules MNOCRTDW.LIB and LIBW.LIB. The first supplies the required Windows startup code for a medium-model DLL that does not use any C run-time functions. LIBW.LIB is the Windows import library, which contains no executable code. The import library provides linkage information for the Windows API functions referenced in the DLL. Windows establishes the final links when it loads the program.

Expanding SYSINFO

SYSINFO is an example of how to write an assembly-language DLL without overwhelming detail. It has plenty of room for expansion and improvements. The following list may give you some ideas:

- To create a heap area for the library, add the line

  HEAPSIZE *value*

  to the module-definition file, where value is an approximate guess for the amount of heap required in bytes. The `DLLEntry` procedure automatically allocates the indicated amount of heap. Keep the data segment moveable, because Windows then provides more heap space if required by the DLL procedures.
- If you want to add a procedure that calls C run-time functions, you must replace MNOCRTDW.LIB with MDLLCW.LIB, which is supplied with the Windows SDK. The MDLLCW.LIB library contains the run-time functions for medium-model DLLs.
- Each time the `GetSysInfo` procedure is called, it retrieves the version number of MS-DOS and Windows, gets the processor type, checks for a coprocessor, and reads the ROM-BIOS release date. Since this information does not change throughout a Windows session, it would be handled more efficiently in the LibMain procedure, which executes only once. The code is currently placed in `GetSysInfo` for the sake of clarity at the expense of efficiency.
- SYSINFO is not a true international program. You can easily add more languages, extending the `days` and `months` arrays accordingly. Moreover, for the sake of simplicity, the `GetSysDate` procedure arranges the date with an American bias. For example, in many parts of the world, the date numeral appears before the month rather than after. If you use SYSINFO in your own applications, you should include code in `LibMain` to determine the correct date format with additional calls to **GetProfileString**. You can find more information on how to do this in Chapter 18 of the Microsoft Windows *Programmer's Reference, Volume 1*, supplied with the Windows SDK.

CHAPTER 11

Writing Memory-Resident Software

Through its memory-management system, MS-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 MS-DOS.

This chapter explains the techniques of writing memory-resident software. The first two sections present introductory material. Following sections describe important MS-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. The MASM 6.1 disks contain complete source code for the two example TSR programs.

Terminate-and-Stay-Resident Programs

MS-DOS maintains a pointer to the beginning of unused memory. Programs load into memory at this position and terminate execution by returning control to MS-DOS. Normally, the pointer remains unchanged, allowing MS-DOS to reuse the same memory when loading other programs.

A terminating program can, however, prevent other programs from loading on top of it. These programs exit to MS-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 MS-DOS services invoked through Interrupt 21h. The following fragment shows how a TSR program terminates through 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 MS-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.

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 MS-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.

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.

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 "MS-DOS Interrupts" in Chapter 7). 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. Such a risk does exist with active TSRs.

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 "active" because it must continually search for its startup 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.

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. (See the section "Replacing an Interrupt Routine" in Chapter 7 for information on how to write an interrupt handler.)

Collectively, interrupt handlers ensure that a TSR operates compatibly with the rest of the system. Individually, each handler fulfills one or more 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

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 MCA (IBM PS/2 and compatible), 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 = key scan code
        call    CheckHotKey             ; Check for hot key
        .IF     carry?                  ; If hot key pressed,
        mov     cs:TsrRequestFlag, TRUE ;   raise flag and
        .                               ;   set up for exit
        .
        .
```

A TSR running on a PS/2 computer cannot reliably read key scan codes using this 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:
        call    CheckHotKey             ; Check for hot key
        .IF     carry?                  ; If hot key pressed,
        mov     cs:TsrRequestFlag, TRUE ;   raise flag and
        .                               ;   set up for exit
        .
        .
```

The example program on page 293 shows 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, which occurs 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.

```
NewTimer PROC FAR
        .
        .
        .
        cmp     TsrRequestFlag, FALSE   ; Has TSR been requested?
        .IF     !zero?                  ; If so, can system be
        call    CheckSystem             ;   interrupted safely?
        .IF     carry?                  ; If so,
        call    ActivateTsr             ;   activate TSR
        .
        .
        .
```

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 simply sets a flag to indicate the device is in use, and then clears the flag when the interrupt finishes.

The following shows a typical monitor handler:

```
NewHandler PROC    FAR
        mov     cs:ActiveFlag, TRUE  ; Set active flag
        pushf                        ; Simulate interrupt by
                                     ;   pushing flags, then
        call    OldHandler           ;   far-calling original routine
        mov     cs: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 it does not interrupt such nonreentrant operations.

The activities of the operating system also affect the system status. With few exceptions, MS-DOS functions are not reentrant and must not be interrupted.

However, monitoring MS-DOS is somewhat more complicated than monitoring hardware. This subject is discussed in "Using MS-DOS in Active TSRs," later in this chapter.

Figure 11.1 illustrates the process described so far. 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.

Figure 11.1 Time Line of Interactions Between Interrupt Handlers for a Typical TSR

The following comments describe the chain of events depicted in Figure 11.1. 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 the user has not requested the TSR. 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, signaling a request for the TSR. The keyboard 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 activating 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 activating the TSR.
5. At time = t + 3/18 second, the timer handler repeats the process. This time it finds all active flags clear, indicating the TSR can safely execute. The timer handler calls the TSR, which sets its own active flag to ensure it will not interrupt itself if requested again.
6. The timer and other interrupts continue to function normally while the TSR executes.

The timer itself can serve as the startup 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.

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 11.1 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 11.1. A handler must check this flag along with the other active flags when determining whether the TSR can safely execute.

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 simplicity, the argument must consist of four digits, including leading zeros. To set the alarm at 7:45 A.M., for example, enter the command:

```
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 MS-DOS Function 35h and stores it in the far pointer `OldTimer`. 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.

        .MODEL tiny                     ; Create ALARM.COM
        .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--inc flag
        .ENDIF                          ;   again so it remains set
        .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 terminating TSR to include the installation code with the rest of the memory it returns to MS-DOS. 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 vector for Interrupt 08 after the alarm sounds. In effect, the multiple installations remain 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 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 "Example of an Advanced TSR: SNAP," later in this chapter, demonstrates this safety measure.

Using MS-DOS in Active TSRs

This section explains how to write active TSRs that can safely call MS-DOS functions. The material explores the problems imposed by the nonreentrant nature of MS-DOS and explains how a TSR can resolve those problems. The solution consists of four parts:

- Understanding how MS-DOS uses stacks
- Determining when MS-DOS is active
- Determining whether a TSR can safely interrupt an active MS-DOS function
- Monitoring the Critical Error flag

Understanding MS-DOS Stacks

MS-DOS functions set up their own stacks, which makes them nonreentrant. If a TSR interrupts an MS-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 an MS-DOS function must not interrupt any function that uses the same stack.

MS-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 MS-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. MS-DOS normally uses the auxiliary stack only when it executes Interrupt 24h (Critical Error Handler).

Determining MS-DOS Activity

A TSR's handlers can determine when MS-DOS is active by consulting a 1-byte flag called the InDos flag. Every MS-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 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 MS-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 MS-DOS returned, it too would have to wait for the terminating event.

The solution lies in determining when the low-order functions 01 through 0Ch are active. MS-DOS provides another service for this purpose: Interrupt 28h, the Idle Interrupt.

Interrupting MS-DOS Functions

MS-DOS continually calls Interrupt 28h from the low-order polling functions as they wait for keyboard input. This signal says that MS-DOS is idle and that a TSR may interrupt provided it does not overwrite the I/O stack. Put another way, a TSR can safely interrupt MS-DOS Functions 01 through 0Ch provided it does not call them.

An active TSR that calls MS-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 MS-DOS function which has stored data on the I/O stack, the TSR in this case must not call any MS-DOS function that also uses the I/O stack. Table 11.1 shows which functions set up the I/O stack for various versions of MS-DOS.

Table 11.1 MS-DOS Internal Stacks

Function	Critical Error flag	MS-DOS 2.x	MS-DOS 3.0	MS-DOS 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.

Monitoring the Critical Error Flag

MS-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. MS-DOS functions higher than 0Ch are illegal during critical error processing. Therefore, a TSR that calls MS-DOS must not execute while the Critical Error flag is set.

MS-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 MS-DOS or for the compatibility version of MS-DOS in OS/2 1.*x*, 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 DOS compatibility version for OS/2 1.x
        test    [CriticalErrorFlag], 0FFh
        jnz     @F
        push    ss:[ ? ]
        int     28h
```

The question mark inside brackets in the preceding **PUSH** statement indicates that the operand for the **PUSH** instruction can be any legal operand.

In either version of MS-DOS, the operand field in the first instruction gives the flag's offset. The value in ES determines the segment address. "Example of an Advanced TSR: SNAP," later in the chapter, presents a program that shows how to locate the Critical Error flag with this technique.

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 relatively harmless, such as moving the cursor, to serious, such as overrunning a stack.

Although a TSR can 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 described on page 293 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

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 MS-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 MS-DOS should also trap critical errors through Interrupt 24h (Critical Error Handler). MS-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 MS-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 startup, the TSR copies the handler's address to the interrupt vector table; it then restores the original vector before returning.
2. It provides complete service for the interrupt; it does not pass control on to the original routine.

Error-trapping handlers often set a flag to let the TSR know 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     cs: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
```

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.)

Preserving Existing Data

A TSR requires its own disk transfer area (DTA) if it calls MS-DOS functions that access the DTA. These include file control block functions and 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.

MS-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 an MS-DOS error. The TSR retrieves the current extended error data by calling MS-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. MS-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 = 0Ah and DS:DX pointing to the data structure. This call restores the extended error data to their original state.

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. MS-DOS version 2.*x* uses Interrupt 2Fh only as an interface for the resident print spooler utility PRINT.COM. Later MS-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.

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 MS-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 this chapter 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 "Deinstalling a TSR," following.

Using the Multiplex Interrupt Under MS-DOS Version 2.*x*

A TSR can use the Multiplex interrupt under MS-DOS version 2.*x*, with certain limitations. Under version 2.*x*, only MS-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 MS-DOS version 2.*x* does not pass control to the previous handler. If you intend 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.

Deinstalling a TSR

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 that may require additional memory space.
- Deinstallation restores the system to a normal state. Thus, sensitive programs that may be incompatible with TSRs can 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.

You can deinstall the TSR with these 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 MS-DOS copies into memory when it loads a program. Place the block's address in the ES register and call MS-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.

Example of an Advanced TSR: SNAP

This section presents SNAP, a memory-resident program that demonstrates most of the techniques discussed in this 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 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 filename, accept the previous filename by pressing ENTER, or cancel the request by pressing ESC.

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 the "Tables" section of the *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
```

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; MS-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 earlier in this chapter. 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 an MS-DOS function that sets up the I/O stack (see "Interrupting MS-DOS Functions," earlier in this chapter). Code

in Microsoft C, for example, must not call **getche** or **kbhit**, since these functions in turn call MS-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.

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 startup signal refers to the section "Auditing Hardware Events for TSR Requests," earlier in this chapter.

Figures 11.2 through 11.4 are flowcharts of the SNAP program. Each chart illustrates a separate phase of SNAP's operation, from installation through memory-residency to deinstallation.

Figure 11.2 Flowchart for SNAP.EXE: Installation Phase

Figure 11.3 Flowchart for SNAP.EXE: Resident Phase

Figure 11.4 Flowchart for SNAP.EXE: Deinstallation Phase

Refer to the flowcharts as you read the following outline. They will help you maintain perspective while exploring the details of SNAP's operation. Text in the outline cross-references the charts.

Note that information in both the outline and the flowcharts is generic. Except for references to the SNAP procedure, all descriptions in the outline and the

flowcharts apply to any TSR created with the HANDLERS and INSTALL modules.

Auditing Hardware Events for TSR Requests

To search for its startup signal, SNAP audits the keyboard with an interrupt handler for either Interrupt 09 (keyboard) or Interrupt 15h (Miscellaneous System Services). 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 11.2.1 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
```

The following describes the code's logic:

- If the program is running under MS-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 11.1 and 11.3 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).

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 whether 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 MS-DOS. These flags indicate the current system status. As the flowchart in Figure 11.3 shows, `CheckRequest` calls `CheckDos` (described following) to determine the status of the operating system. `CheckRequest` then calls `CheckHardware` to check hardware status.

`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 MS-DOS Activity

As Figure 11.2 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`.

When called from the `CheckRequest` procedure, `CheckDos` reads InDos to determine whether 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 MS-DOS is active. This follows from the general rule that interrupt handlers must not call any MS-DOS function.

The next two sections more fully describe the procedure `CheckDos`.

Interrupting MS-DOS Functions

Figure 11.3 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 MS-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.

For more information on this topic, see the section "Interrupting MS-DOS Functions," earlier in this chapter.

Monitoring the Critical Error Flag

The procedure `GetDosFlags` (Figure 11.2) determines the address of the Critical Error flag. The procedure stores the flag's address in the far pointer `CritErrAddr`.

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 11.3 shows, `Clock` and `Idle` invoke the TSR by calling the procedure `Activate`. 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)	MS-DOS detects a CTRL+C sequence from the keyboard or input stream
CritError	24h (Critical Error Handler)	MS-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 stops MS-DOS from taking 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 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.

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 11.3 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 an MS-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.

It is possible for SNAP to 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.

For more information about Interrupt 2Fh, see the section "Communicating through the Multiplex Interrupt," earlier in this chapter.

SNAP accesses Interrupt 2Fh through the procedure `CallMultiplex`, as shown in Figures 11.2 and 11.4. 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 a TSR

During deinstallation, `CallMultiplex` locates SNAP's multiplex handler as described previously. The handler `Multiplex` receives the verification request and returns in ES the code segment of the resident program.

`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	MS-DOS Idle handler
Multiplex	Multiplex handler

`Deinstall` calls MS-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, stopping the program 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 MS-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 MS-DOS Function 49h (Release Memory) to return SNAP's memory to the operating system. The program ends 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. MS-DOS can use the free memory to store the next program's environment block. However, MS-DOS loads the program itself above the still-resident TSR.

CHAPTER 12

Mixed-Language Programming

Mixed-language programming allows you to combine the unique strengths of Microsoft Basic, C, C++, and FORTRAN with your assembly-language routines. Any one of these languages can call MASM routines, and you can call any of these languages from within your assembly-language programs. This makes virtually all the routines from high-level–language libraries available to a mixed-language program.

MASM 6.1 provides mixed-language features similar to those in high-level languages. For example, you can 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, as explained in Chapter 20 of *Environment and Tools*.

The mixed-language features of MASM 6.1 do not make older methods of defining mixed-language interfaces obsolete. In most cases, mixed-language programs written with earlier versions of MASM will assemble and link correctly under MASM 6.1. (For more information, see Appendix A.)

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. You should already understand the languages you want to combine and should know how to write, compile, and link multiple-module programs with these languages.

This chapter covers only assembly-language interface with C, C++, Basic, and FORTRAN; it does not cover mixed-language programming between high-level languages. The focus here is the Microsoft versions of C, C++, Basic, and FORTRAN, but the same principles apply to other languages and compilers. Many of the techniques used in this chapter are explained in the material in Chapter 7 on writing procedures in assembly language, and in Chapter 8 on multiple-module programming.

The first section of this chapter discusses naming and calling conventions. The next section, “Writing an Assembly Procedure for a Mixed-Language Program,” provides a template for writing an assembly-language procedure that can be

called from another module written in a high-level language. This represents the essence of mixed-language programming. Assembly language is often used for creating fast secondary routines in a large program written in a high-level language.

The third section describes specific conventions for linking assembly-language procedures with modules in C, C++, Basic, and FORTRAN. 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.

Naming and Calling Conventions

Each language has its own set of conventions, which fall into two categories:

- The "naming convention" specifies how or if the compiler or assembler alters the name of an identifier before placing it into an object file.
- The "calling convention" determines how a language implements a call to a procedure and how the procedure returns to the caller.

MASM supports several different conventions. The assembler uses C convention when you specify a language type (*langtype*) of **C**, and Pascal convention for language types **PASCAL**, **BASIC**, or **FORTRAN**. To the assembler, the keywords **BASIC**, **PASCAL**, and **FORTRAN** are synonymous. MASM also supports the **SYSCALL** and **STDCALL** conventions, which mix elements of the C and Pascal conventions.

MASM gives you several ways to set the naming and calling conventions in your assembly-language program. Using **.MODEL** with a *langtype* sets the default for the module. This can also be done with the **OPTION** directive. 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.

When you write mixed-language routines, the easiest way to ensure convention compatibility is to adopt the conventions of the called procedure's language. However, Microsoft languages can change the naming and calling conventions for different procedures. If your program must call a procedure that uses an argument-passing method different from that of the default language, prototype the procedure first with the desired language type. This tells the assembler to override the conventions of the default language and assume the proper conventions for the prototyped procedure. "The MASM/High-Level–Language Interface" section in this chapter explains how to change the default conventions. The following sections provide more detail on the information summarized in Table 12.1.

Table 12.1 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	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.

Naming Conventions

"Naming convention" refers to the way a compiler or assembler stores the names of identifiers. The first two rows of Table 12.1 show how each language type affects symbol names. **SYSCALL** leaves symbol names as they appear in the source code, but **C** and **STDCALL** add an underscore prefix. **PASCAL**, **BASIC**, and **FORTRAN** change symbols to all uppercase.

The following list describes how these naming conventions affect a variable called `Big Time` in your source code:

Langtype Specified	Characteristics
SYSCALL	Leaves the name unmodified. The linker sees the variable as `Big Time`.
C, **STDCALL**	The assembler (or compiler) adds a leading underscore to the name, but does not change case. The linker sees the variable as `_Big Time`.
PASCAL, **FORTRAN**, **BASIC**	Converts all names to uppercase. The linker sees the variable as `Big Time`.

The C Calling Convention

Specify the C language type for assembly-language procedures called from programs that assume the C calling convention. Note that such programs are not necessarily written in C, since other languages can mimic C conventions.

Argument Passing

With the C calling convention, the caller pushes arguments from right to left as they appear in the caller's argument list. The called procedure returns without removing the arguments from the stack. It is the caller's responsibility to clean the stack after the call, either by popping the arguments or by adding an appropriate value to the stack pointer SP.

Register Preservation

The called routine must return with the original values in BP, SI, DI, DS, and SS. It must also preserve the direction flag.

Varying Number of Arguments

The additional overhead of cleaning the stack after each call has compensations. It frees the caller from having to pass a set number of arguments to the called procedure each time. Because the first argument in the list 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.

For example, consider the C library function **printf**, which accepts different numbers of arguments. A C program calls the function like this:

```
printf( "Numbers:  %f  %f  %.2f\n", n1, n2, n3 );
printf( "Also:     %f", n4 );
```

The first line passes four arguments (including the string in quotes) and the second line passes only two arguments. Notice that **printf** has no reliable way of determining how many arguments the caller has pushed. Therefore, the function returns without adjusting the stack. The C calling convention requires the caller to take responsibility for removing the arguments from the stack, since only the caller knows how many arguments it passed.

Use **INVOKE** to call a C-callable function from your assembly-language program, since **INVOKE** automatically generates the necessary stack-cleaning code after the call. You must also prototype the function with the **VARARG** keyword if appropriate, as explained in "Procedures," Chapter 7. Similarly, when you write a C-callable procedure that accepts a varying number of arguments, include **VARARG** in the procedure's **PROC** statement.

The Pascal Calling Convention

By default, the *langtype* for **FORTRAN**, **BASIC**, and **PASCAL** selects 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.

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

A routine that uses the Pascal calling convention must preserve SI, DI, BP, DS, and SS. 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.

The STDCALL and SYSCALL Calling Conventions

A **STDCALL** procedure adopts the C name and calling conventions when prototyped with the **VARARG** keyword. Refer to the section "Declaring Parameters with the PROC Directive" in Chapter 7. Without **VARARG**, the procedure uses the C naming and Pascal calling conventions. **STDCALL** provides compatibility with 32-bit versions of Microsoft compilers.

As Table 12.1 shows, **SYSCALL** is identical to the C calling convention, but does not add an underscore prefix to symbols.

Argument Passing

Argument passing order for both **STDCALL** and **SYSCALL** is the same as the C calling convention. The caller pushes the arguments from right to left and must remove the parameters from the stack after the call. However, **STDCALL** requires the called procedure to clean the stack if the procedure does not accept a variable number of arguments.

Register Preservation

Both conventions require the called procedure to preserve the registers BP, SI, DI, DS, and SS. Under **STDCALL**, the direction flag is clear on entry and must be returned clear.

Varying Number of Arguments

SYSCALL allows a variable number of arguments in the same way as the C calling convention. **STDCALL** also mimics the C convention when **VARARG** appears in the called procedure's declaration or definition. It allows a varying number of arguments and requires the caller to clean the stack. If not declared

or defined with **VARARG**, the called procedure does not accept a variable argument list and must clean the stack before it returns.

Writing an Assembly Procedure For a Mixed-Language Program

MASM 6.1 simplifies the coding required for linking MASM routines to high-level–language routines. You can use the **PROTO** directive to write procedure prototypes, and the **INVOKE** directive to call external routines. MASM simplifies procedure-related tasks in the following ways:

- 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 "Procedures" in Chapter 7. This section provides a template that 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 "Declaring Parameters with the **PROC** Directive" in Chapter 7.

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. For information on calling conventions, see "Naming and Calling Conventions" earlier in this chapter.

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 write such code is provided in "Declaring Parameters with the PROC Directive" in Chapter 7.

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 "Passing Arguments on the Stack" and "User-Defined Prologue and Epilogue Code" in Chapter 7.

The MASM/High-Level–Language Interface

Since high-level–language programs require initialization, you must write the main routine of a mixed-language program in the high-level language, or link with the startup code supplied by the high-level–language compiler. This gives the assembly code access to high-level routines or library functions. The next section explains how to link an assembly-language program with C-language startup code.

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 type checking and data conversion for the argument types so that the procedure receives compatible data. For explanations of how to write procedure prototypes and several examples of procedure declarations and the corresponding prototypes, see "Declaring Procedure Prototypes" in Chapter 7.

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. The use of H2INC is explained in Chapter 20 in *Environment and Tools*.

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.

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, 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. Though slower than passing a near reference, this method is necessary for passing data that lies outside the default data segment. (This is not an issue in Basic unless you have specifically requested far memory.)
- Value. Passes only a copy of the variable, not its address. With this method, the called routine gets a copy of the argument on the stack, but has no access to the original variable. The copy is discarded when the routine returns, and the variable retains its original value.

When you pass arguments between routines written in different languages, you must ensure that the caller and the called routine use the same conventions for passing and receiving arguments. 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 offsets. This lets the called routine use the **LES** and **LDS** instructions to read far addresses from the stack. Furthermore, each word of an argument is placed on the stack in order of significance. Thus, the high word of a long integer is pushed first, followed by the low word.

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, this same array 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]
```

The C/MASM Interface

This section summarizes the characteristics of the interface between MASM and Microsoft C and QuickC compilers. 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 16-bit C data types and equivalent data types in MASM 6.1. For 32-bit C compilers, **int** and **unsigned int** are equivalent to the MASM types **SDWORD** and **DWORD**, respectively.

C Type	Equivalent MASM Type
unsigned char	**BYTE**
char	**SBYTE**
unsigned short, **unsigned int**	**WORD**
int, **short**	**SWORD**
unsigned long	**DWORD**
long	**SDWORD**

float	**REAL4**
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, you should assemble MASM modules with the /Cx or /Cp option to prevent the assembler from converting names to uppercase.

Argument-Passing Defaults

C always passes arrays by reference and all other variables (including structures) by value. C programs in tiny, small, and medium model pass near addresses for arrays, unless another distance is specified. Compact-, large-, and huge-model programs pass far addresses by default. To pass by reference a variable type other than array, use the C-language address-of operator (**&**).

If you need to pass an array by value, declare the array as a structure member and pass a copy of the entire structure. However, this practice is rarely necessary and usually impractical except for very small arrays, since it can make substantial demands on stack space. If your program must maintain an array through a procedure call, create a temporary copy of the array in heap and provide the copy to the procedure by reference.

Changing the Calling Convention

Put **_pascal** or **_fortran** in the C function declaration to specify the Pascal calling convention.

Array Storage

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 12.1 C String Format

Since `msg` is an array of characters, it is passed by reference.

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 want to be public.

Structure Alignment

By default, C uses word alignment (unpacked storage) for all data objects longer than 1 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 aligns on byte boundaries by default.

When converting .H files with H2INC, you can use the /Zp command-line option to specify structure alignment. If you do not specify the /Zp option, H2INC uses word-alignment. Without H2INC, set the alignment to 2 when declaring the MASM structure, compile the C module with /Zp1, or assemble 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 12.2 shows the register conventions for returning simple data types to a C program.

Table 12.2 Register Conventions for Simple Return Values

Data Type	Registers
char	AL
short, **near**, **int** (16-bit)	AX
short, **near**, **int** (32-bit)	EAX
long, **far** (16-bit)	High-order portion (or segment address) in DX; low-order portion (or offset address) in AX
long, **far** (32-bit)	High-order portion (or segment address) in EDX; low-order portion (or offset address) in EAX

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.

Structures less than 4 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 these C prototypes, H2INC generates this code:

```
@proto_0        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 following short example, 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 12.2 shows how functions that observe the C calling convention use the stack frame.

Figure 12.2 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.

Using the C Startup Code

This section explains how to write an assembly-language program that can call C library functions. It links with the C startup module, which performs the necessary initialization required by the library functions.

You must follow these steps when writing such a program:

1. Specify the C convention in the **.MODEL** statement.
2. Include the following (optional) statement to note linkage with the C startup module:

   ```
   EXTERN  _acrtused:abs
   ```

3. Prototype or declare as external all C functions the program references.
4. Include a public procedure called `main` in your assembly-language module. The C startup code calls `_main` (which is why all C programs begin with a `main` function). This procedure serves as the effective entry point for your program.
5. Omit an entry point in the program's **END** directive. The C startup code serves as the true entry point when the program runs.
6. Assemble with ML's /Cx switch to preserve the case of nonlocal names.

The following example serves as a template for these steps. The program calls the C run-time function **printf** to display two variables.

```
.MODEL  small, c                        ; Step 1: declare C conventions
EXTERN  _acrtused:abs                   ; Step 2: bring in C startup
        .
        .
        .
printf  PROTO   NEAR,                   ; Step 3: prototype
                pstring:NEAR PTR BYTE,  ;         external C
                num1:WORD, num2:VARARG  ;         routines
        .DATA
format  BYTE    '%i    %i', 13, 0

        .CODE

main    PROC    PUBLIC                  ; Step 4: C startup calls here
        .
        .
        .
        INVOKE  printf, OFFSET format, ax, bx
        .
        .
        .
        END                             ; Step 5: no label on END
```

The C++/MASM Interface

C++ can apply a protocol called a “linkage specification” to mixed-language procedures. This lets you link C++ code in the same way as C code. All information in the preceding section applies when linking assembly-language and C++ routines through the C linkage specification.

The C linkage specification forces the C++ compiler to adopt C conventions—which are not the same as C++ conventions—for listed routines. Since MASM does not specifically support C++ conventions, set the C linkage specification in your C++ code for all mixed-language routines, as shown here:

extern “C” *declaration*

where *declaration* is the prototype of an exported C++ function or an imported assembly-language procedure. You can bracket a list of declarations:

```
extern "C"
{
    int    WriteLine( short attr, char *string );
    void   GoExit( int err );
}
```

or apply the specification to individual prototypes:

```
extern "C" int    WriteLine( short attr, char *string );
extern "C" void   GoExit( int err );
```

Note the syntax remains the same whether `WriteLine` and `GoExit` are exported C++ functions or imported assembly-language routines. The linkage specification applies only to called routines, not to external variables. Use the **extern** keyword (without the **"C"**) as you normally would when identifying objects external to the C++ module.

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 converts all exported names to uppercase.

Compatible Data Types

This list shows the FORTRAN data types that are equivalent to the MASM 6.1 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**	**REAL8**

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 require 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.

Array Storage

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]
```

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 12.3 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 (unpacked storage) for all data objects larger than 1 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. The MASM default is byte-alignment, so you should 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 4 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).

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 4-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 8-byte floating-point number.

A FORTRAN **LOGICAL*2** is stored as a 1-byte indicator value (1=true, 0=false) followed by an unused byte. A FORTRAN **LOGICAL*4** is stored as a 1-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.

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 the assembly procedure, consider how the parameters are placed on the stack, as illustrated in Figure 12.4.

Figure 12.4 FORTRAN Stack Frame

Figure 12.4 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 2-byte, not 4-byte, address. The return address is 4 bytes long because procedures called from FORTRAN must always be **FAR**.

The assembler code looks like this:

```
        .MODEL LARGE, FORTRAN

Power2  PROTO   FORTRAN, pFactor:FAR PTR SWORD, pPower:FAR PTR SWORD

        .CODE

Power2  PROC    FORTRAN, pFactor:FAR PTR SWORD, pPower:FAR PTR SWORD

        les     bx, pFactor     ; ES:BX points to factor
        mov     ax, es:[bx]     ; AX = value of factor
        les     bx, pPower      ; ES:BX points to power
        mov     cx, es:[bx]     ; CX = value of power
        shl     ax, cl          ; Multiply by 2^power
        ret                     ; Return result in AX
Power2  ENDP
        END
```

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 following list shows the Basic data types that are equivalent to the MASM 6.1 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 2-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 any other variable type by value, apply the **BYVAL** keyword to the argument in the **DECLARE** statement. You cannot pass arrays and user-defined types by value.

```
DECLARE SUB Test(BYVAL a%, b%, SEG c%)

CALL Test(x%, y%, z%)
CALLS Test(x%, y%, z%)
```

This **CALL** statement 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 conventions. This also allows a call to a MASM procedure with a varying number of arguments.

Array Storage

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.

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 stores arrays 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 handles memory allocation for arrays dynamically, and thus may shift the location of the array in memory.

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.
- The MASM routine that receives the array should not call back to one of the calling program's routines before it has finished processing the array. Changing data within the caller's heap—even data unrelated to the array—may change the array's location in the heap. This would invalidate any further work the called routine performs, since the routine would be operating on the array's old location.
- Basic can pass any member of an array by value. When passing individual array elements, these 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

Basic maintains a 4-byte string descriptor for each string, as shown in the following. The first field of the string descriptor contains a 2-byte integer indicating the length of the actual string text. The second field contains the offset address of this text within the caller's data segment.

Figure 12.5 Basic String Descriptor Format

An assembly-language procedure can store a Basic string descriptor as a simple structure, like this:

```
DESC    STRUCT
  len   WORD    ?       ; Length of string
  off   WORD    ?       ; Offset of string
DESC    ENDS

string  BYTE    "This text referenced by a string descriptor"
sdesc   DESC    (LENGTHOF string, string)
```

Version 7.0 or later of the Microsoft Basic Compiler provides new functions that access string descriptors. These functions simplify the process of sharing Basic string data with routines written in other languages.

Earlier versions of Basic offer the **LEN** (Length) and **SADD** (String Address) functions, which together obtain the information stored in a string descriptor. **LEN** returns the length of a string in bytes. **SADD** returns the offset address of a string in the data segment. The caller must provide both pieces of information so the called procedure can locate and read the entire string. The address returned by **SADD** is declared as type **INTEGER** but is actually equivalent to a C near pointer.

If you need to pass the far address of a string, use the **SSEGADD** (String Segment Address) function of Microsoft Basic version 7.0 or later. You can also determine the segment address of the first element with **VARSEG**.

External Data

Declaring global data in Basic follows the same two-step process as in other languages:

1. Declare shareable data in Basic with the **COMMON** statement.
2. Identify the shared variables in your assembly-language procedures with the **EXTERN** keyword. Place the **EXTERN** statement outside of a code or data segment when declaring far data.

Structure Alignment

Basic packs user-defined types. For MASM structures to be compatible, select byte-alignment.

Compiling and Linking

Always use medium model in assembly-language procedures linked with Basic modules. 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 2-byte integers should be returned from the MASM procedure in AX and 4-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 Basic when you change the calling convention with **CDECL**. 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 returns a variable's segment address, and **VARPTR** returns a variable's offset address. These intrinsic Basic functions enable your program to pass near or far addresses.

Example

This example calls the `Power2` procedure in the MASM 6.1 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 12.6 shows how the arguments are placed on the stack.

Figure 12.6 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     ; BX points to factor
        mov     ax, [bx]                ; Load factor into AX
        mov     bx, WORD PTR power      ; BX points to power
        mov     cx, [bx]                ; Load power into 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 4 bytes long because procedures called from Basic must be **FAR**.

CHAPTER 13

Writing 32-Bit Applications

This chapter is an introduction to 32-bit programming for the 80386. The guidelines in this chapter also apply to the 80486 processor, which is basically a faster 80386 with the equivalent of a 80387 floating-point processor. Since you are already familiar with 16-bit real-mode programming, this chapter covers the differences between 16-bit programming and 32-bit protected-mode programming.

The 80386 processor (and its successors such as the 80486) can run in real mode, virtual-86 mode, and in protected mode. In real and virtual-86 modes, the 80386 can run 8086/8088 programs. In protected mode, it can run 80286 programs. The 386 also extends the features of protected mode to include 32-bit operations and segments larger than 64K.

The MS-DOS operating system directly supports 8086/8088 programs, which it runs either in real mode or virtual-86 mode. Native 32-bit 80386 programs can be run by using a "DOS extender," by using the WINMEM32.DLL facility of Microsoft Windows 3.x, or by running a native 32-bit operating system, such as Microsoft Windows NT. You can use MASM to generate object code (OMF or COFF) for 32-bit programs. To do this, you will need a software development kit such as the Windows SDK for the target environment. Such kits include the linker and other components specific to your chosen operating environment.

32-Bit Memory Addressing

The 80386 has six segment registers. Four of these are familiar to 8086/8088 programmers: CS (Code Segment), SS (Stack Segment), DS (Data Segment), and ES (Extra Segment). The two additional registers, FS and GS, are used as data segment registers.

Memory addresses on 80x86 machines consist of two parts—a segment and an offset. In real-mode programs, the segment is a 16-bit number and the offset is a 16-bit number. Effective addresses are calculated by multiplying the segment by

16 and adding the offset to it. In protected mode, the segment value is not used directly as a number, but instead is an index to a table of "selectors." Each selector describes a block of memory, including attributes such as the size and location of the block, and the access rights the program has to it (read, write, execute). The effective address is calculated by adding the offset to the base address of the memory block described by the selector.

All segment registers are 16 bits wide. The offset in a 32-bit protected-mode program is itself 32 bits wide, which means that a single segment can address up to 4 gigabytes of memory. Because of this large range, there is little need to use segment registers to extend the range of addresses in 32-bit programs. If all six segment registers are initially set to the same value, then the rest of the program can ignore them and treat the processor as if it used a 32-bit linear address space. This is called 0:32, or flat, addressing. (The full segmented 32-bit addressing mode, in which the segment registers can contain different values, is called 16:32 addressing.) Flat addressing is used by the Windows NT operating system.

Figure 13.1 32-Bit Register Set

MASM Directives for 32-Bit Programming

If you use the simplified segment directives, a 32-bit program is surprisingly similar to a program for MS-DOS. Here are the differences:

- Supply the **.386** directive, which enables the 32-bit programming features of the 386 and its successors. The **.386** directive must precede the **.MODEL** directive.
- For flat-model programming, use the directive

  .MODEL flat, stdcall

  which tells the assembler to assume flat model (0:32) and to use the Windows NT standard calling convention for subroutine calls.
- Precede your data declarations with the **.DATA** directive.
- Precede your instruction codes with the **.CODE** directive.
- At the end of the source file, place an **END** directive.

Sample Program

The following sample is a 32-bit assembly language subroutine, such as might be called from a 32-bit C program written for the Windows NT operating system. The program illustrates the use of a variety of directives to make assembly language easier to read and maintain. Note that with 32-bit flat model programming, there is no longer any need to refer to segment registers, since these are artifacts of segmented addressing.

```
;* szSearch - An example of 32-bit assembly programming using MASM 6.1
;*
;* Purpose: Search a buffer (rgbSearch) of length cbSearch for the
;*          first occurrence of szTok (null terminated string).
;*
;* Method:  A variation of the Boyer-Moore method
;*          1. Determine length of szTok (n)
;*          2. Set array of flags (rgfInTok) to TRUE for each character
;*                  in szTok
;*          3. Set current position of search to rgbSearch (pbCur)
;*          4. Compare current position to szTok by searching backwards
;*                  from the nth position. When a comparison fails at
;*                  position (m), check to see if the current character
;*                  in rgbSearch is in szTok by using rgfInTok. If not,
;*                  set pbCur to pbCur+(m)+1 and restart compare. If
;*                  pbCur reached, increment pbCur and restart compare.
;*          5. Reset rgfInTok to all 0 for next instantiation of the
;*                  routine.

        .386
        .MODEL  flat, stdcall

FALSE   EQU     0
TRUE    EQU     NOT FALSE
```

```
        .DATA
; Flags buffer - data initialized to FALSE. We will
; set the appropriate flags to TRUE during initialization
; of szSearch and reset them to FALSE before exit.
rgfInTok        BYTE    256 DUP (FALSE);

        .CODE

PBYTE   TYPEDEF PTR BYTE

szSearch PROC PUBLIC USES esi edi,
         rgbSearch:PBYTE,
         cbSearch:DWORD,
         szTok:PBYTE

; Initialize flags buffer. This tells us if a character is in
; the search token - Note how we use EAX as an index
; register. This can be done with all extended registers.
        mov     esi, szTok
        xor     eax, eax
        .REPEAT
        lodsb
        mov     BYTE PTR rgfInTok[eax], TRUE
        .UNTIL  (!AL)

; Save count of szTok bytes in EDX
        mov     edx, esi
        sub     edx, szTok
        dec     edx

; ESI will always point to beginning of szTok
        mov     esi, szTok

; EDI will point to current search position
; and will also contain the return value
        mov     edi, rgbSearch

; Store pointer to end of rgbSearch in EBX
        mov     ebx, edi
        add     ebx, cbSearch
        sub     ebx, edx
```

```
; Initialize ECX with length of szTok
        mov     ecx, edx
        .WHILE  ( ecx != 0 )
        dec     ecx             ; Move index to current
        mov     al, [edi+ecx]   ;   characters to compare

; If the current byte in the buffer doesn't exist in the
; search token, increment buffer pointer to current position
; +1 and start over. This can skip up to 'EDX'
; bytes and reduce search time.
        .IF     !(rgfInTok[eax])
        add     edi, ecx
        inc     edi             ; Initialize ECX with
        mov     ecx, edx        ;   length of szTok
; Otherwise, if the characters match, continue on as if
; we have a matching token
        .ELSEIF (al == [esi+ecx])
        .CONTINUE
; Finally, if we have searched all szTok characters,
; and land here, we have a mismatch and we increment
; our pointer into rgbSearch by one and start over.
        .ELSEIF (!ecx)
        inc     edi
        mov     ecx, edx
        .ENDIF

; Verify that we haven't searched beyond the buffer.
        .IF     (edi > ebx)
        mov     edi, 0          ; Error value
        .BREAK
        .ENDIF
        .ENDW

; Restore flags in rgfInTok to 0 (for next time).
        mov     esi, szTok
        xor     eax, eax
        .REPEAT
        lodsb
        mov     BYTE PTR rgfInTok[eax], FALSE
        .UNTIL  !AL

; Put return value in eax
        mov     eax, edi
        ret
szSearch ENDP

end
```

APPENDIX A

Differences Between MASM 6.1 and 5.1

For the many users who come to version 6.1 of the Microsoft Macro Assembler directly from the popular MASM 5.1, this appendix describes the differences between the two versions. Version 6.1 contains significant changes, including:

- An integrated development 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 Windows NT and new instructions for the 80486 processor.

The **OPTION M510** directive (or the /Zm command-line switch) assures nearly complete compatibility between MASM 6.1 and MASM 5.1. However, to take full advantage of the enhancements in MASM 6.1, you will need to rewrite some code written for MASM 5.1.

The first section of this appendix describes the new or enhanced features in MASM 6.1. The second section, "Compatibility Between MASM 5.1 and 6.1," explains how to:

- Minimize the number of required changes with the **OPTION** directive.
- Rewrite your existing assembly code, if necessary, to take advantage of the assembler's enhancements.

New Features of Version 6.1

This section gives an overview of the new features of MASM 6.1 and provides references to more detailed information elsewhere in the documentation. For full explanations and coding examples, see the documentation listed in the cross-references.

The Assembler, Environment, and Utilities

Most of the executable files provided with MASM 6.1 are new or revised. For a complete list of these files, read the PACKING.TXT file on the distribution disk. The book *Getting Started* also provides information about setting up the environment, assembler, and Help system.

The Assembler

The macro assembler, named ML.EXE, can assemble and link in one step. Its new 32-bit operation gives ML.EXE the ability to handle much larger source files than MASM 5.1. The command-line options are new. For example, the /Fl and /Sc options generate instruction timings in the listing file. Command-line options are case-sensitive and must be separated by spaces.

For backward compatibility with MASM 5.1 makefiles, MASM 6.1 includes the MASM.EXE utility. MASM.EXE translates MASM 5.1 command-line options to the new MASM 6.1 command-line options and calls ML.EXE. See the *Reference* book 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 20 of *Environment and Tools.*

NMAKE

NMAKE replaces the MAKE utility. NMAKE provides new functions for evaluating target files and more flexibility with macros and command-line options. For more information, see *Environment and Tools.*

Integrated Environment

PWB is an integrated development environment for writing, developing, and debugging programs. For information on PWB and the CodeView debugging application, see *Environment and Tools*.

Online Help

MASM 6.1 incorporates the Microsoft Advisor Help system. Help provides a vast database of online help about all aspects of MASM, including the syntax

and timings for processor and coprocessor instructions, directives, command-line options, and support programs such as LINK and PWB.

For information on how to set up the help system, see *Getting Started*. 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 *Environment and Tools.*

Other Programs

MASM 6.1 contains the most recent versions of LINK, LIB, BIND, CodeView, and the mouse driver. The CREF program is not included in MASM 6.1. The Source Browser provides the information that CREF provided under MASM 5.1. For more information on the Source Browser, see Chapter 5 of *Environment and Tools* or Help.

Segment Management

This section lists the changes and additions to memory-model support and directives that relate to memory model.

Predefined Symbols

The following 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 about predefined symbols, see “Predefined Symbols” in Chapter 1.

Enhancements to the ASSUME Directive

MASM automatically generates **ASSUME** values for the code segment register (CS). It is no longer necessary to include lines such as

```
ASSUME CS:MyCodeSegment
```

in your programs. In addition, the **ASSUME** directive can include **ERROR**, **FLAT**, or *register*:*type*. MASM 6.1 issues a warning when you specify **ASSUME** values for CS other than the current segment or group.

For more information, see "Setting the ASSUME Directive for Segment Registers" in Chapter 2 and "Defining Register Types with ASSUME" in Chapter 3.

Relocatable Offsets

For compatibility with applications for Windows, the **LROFFSET** operator can calculate a relocatable offset, which is resolved by the loader at run time. See Help for details.

Flat Model

MASM 6.1 supports the flat-memory model of Windows NT, which allows segments as large as 4 gigabytes. All other memory models limit segment size to 64K for MS-DOS and Windows. For more information about memory models, see "Defining Basic Attributes with .MODEL" in Chapter 2.

Data Types

MASM 6.1 supports an improved data typing. This section summarizes the improved forms of data declarations in MASM 6.1.

Defining Typed Variables

You can now use the type names as directives to define variables. Initializers are unsigned by default. The following example lines are equivalent:

```
var1    DB      25
var1    BYTE    25
```

Signed Types

You can use the **SBYTE**, **SWORD**, and **SDWORD** directives to declare signed data. For more information about these directives, see "Allocating Memory for Integer Variables" in Chapter 4.

Floating-Point Types

MASM 6.1 provides the **REAL4**, **REAL8**, and **REAL10** directives for declaring floating-point variables. For information on these type directives, see "Declaring Floating-Point Variables and Constants" in Chapter 6 .

Qualified Types

Type definitions can now include distance and language type attributes. Procedures, procedure prototypes, and external declarations let you specify the type as a qualified type. A complete description of qualified types is provided in the section "Data Types" in Chapter 1.

Structures

Changes to structures since MASM 5.1 include:

- Structures can be nested.
- The names of structure fields need not be unique. As a result, you must qualify references to field names.
- Initialization of structure variables can continue over multiple lines provided the last character in the line before the comment field is a comma.
- Curly braces and angle brackets are equivalent.

For example, this code works in MASM 6.1:

```
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 "Compatibility between MASM 5.1 and 6.1," later in this appendix. For more information on structures and unions, see "Structures and Unions" in Chapter 5.

Unions

MASM 6.1 allows the definition of unions with the **UNION** directive. Unions differ from structures in that all fields within a union occupy the same data space. For more information, see "Structures and Unions" in Chapter 5.

Types Defined with TYPEDEF

The **TYPEDEF** directive defines a type for use later in the program. It is most useful for defining pointer types. For more information on defining types, see "Data Types" in Chapter 1, and "Defining Pointer Types with TYPEDEF" in Chapter 3.

Names of Identifiers

MASM 6.1 accepts identifier names up to 247 characters long. All characters are significant, whereas under MASM 5.1, names are significant to 31 characters only. For more information on identifiers, see "Identifiers" in Chapter 1.

Multiple-Line Initializers

In MASM 6.1, a comma at the end of a line (except in the comment field) implies that the line continues. For example, the following code is legal in MASM 6.1:

```
longstring      BYTE    "This string ",
                        "continues over two lines."
bitmasks        BYTE    80h, 40h, 20h, 10h,
                        08h, 04h, 02h, 01h
```

For more information, see "Statements" in Chapter 1.

Comments in Extended Lines

MASM 5.1 allows a backslash (\ ) as the line-continuation character if it is the last nonspace character in the line. MASM 6.1 permits a comment to follow the backslash.

Determining Size and Length of Data Labels

The **LENGTHOF** operator returns the number of data items allocated for a data label. MASM 6.1 also provides the **SIZEOF** operator. When applied to a type, **SIZEOF** 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.

MASM 6.1 recognizes the **LENGTH** and **SIZE** operators for backward compatibility. For a description of the behavior of **SIZE** under **OPTION M510**, see "Length and Size of Labels with OPTION M510," later in this appendix. For obsolete behavior with the **LENGTH** operator, see also "LENGTH Operator Applied to Record Types," page 356.

For information on **LENGTHOF** and **SIZEOF**, see the following sections in chapter 5: "Declaring and Referencing Arrays," "Declaring and Initializing

Strings," "Declaring Structure and Union Variables," and "Defining Record Variables."

HIGHWORD and LOWWORD Operators

These operators return the high and low words for a given 32-bit operand. 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

Under MASM 5.1, applying the **PTR** operator to a data initializer determines the size of the data displayed by CodeView. You can still use **PTR** in this manner in MASM 6.1, but it does not affect CodeView typing. Defining pointers with the **TYPEDEF** directive allows CodeView to generate correct information. See "Defining Pointer Types with TYPEDEF" in Chapter 3.

Procedures, Loops, and Jumps

With its significant improvements for procedure and jump handling, MASM 6.1 closely resembles high-level–language implementations of procedure calls. MASM 6.1 generates the code to correctly handle argument passing, check type compatibility between parameters and arguments, and process a variable number of arguments. MASM 6.1 can also automatically recast jump instructions to correct for insufficient jump distance.

Function Prototypes and Calls

The **PROTO** directive lets you prototype procedures in the same way as high-level languages. **PROTO** enables type-checking and type conversion of arguments when calling the procedure with **INVOKE**. For more information, see "Declaring Procedure Prototypes" in Chapter 7.

The **INVOKE** directive sets up code to call a procedure and correctly pass arguments according to the prototype. MASM 6.1 also provides the **VARARG** keyword to pass a variable number of arguments to a procedure with **INVOKE**. For more information about **INVOKE** and **VARARG**, see "Calling Procedures with INVOKE" and "Declaring Parameters with the PROC Directive" in Chapter 7.

The **ADDR** keyword is new since MASM 5.1. When used with **INVOKE**, it provides the address of a variable, in the same way as the address-of operator (**&**) in C. This lets you conveniently pass an argument by reference rather than value. See "Calling Procedures with INVOKE" in Chapter 7.

High-Level Flow-Control Constructions

MASM 6.1 contains several 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.

Directives new since MASM 5.1 include **.IF**, **.ELSE**, **.ELSEIF**, **.REPEAT**, **.UNTIL**, **.UNTILCXZ**, **.WHILE**, and **.ENDW**. MASM 6.1 also provides the associated **.BREAK** and **.CONTINUE** directives for loops and **IF** statements. For more information, see "Loops" in Chapter 7 and "Decision Directives" on page 171.

Automatic Optimization for Unconditional Jumps

MASM 6.1 automatically determines the smallest encoding for direct unconditional jumps. See "Unconditional Jumps" in Chapter 7.

Automatic Lengthening for Conditional Jumps

If a conditional jump cannot reach its target destination, MASM automatically recasts the code to use an unconditional jump to the target. See "Jump Extending," page 169.

User-Defined Stack Frame Setup and Cleanup

The prologue code generated immediately after a **PROC** statement sets up the stack for parameters and local variables. The epilogue code handles stack cleanup. MASM 6.1 allows user-defined prologues and epilogues, as described in "Generating Prologue and Epilogue Code" in Chapter 7.

Simplifying Multiple-Module Projects

MASM 6.1 simplifies the sharing of code and data among modules and makes the use of include files more efficient.

EXTERNDEF in Include Files

MASM 5.1 requires that you declare public and external all data and routines used in more than one module. With MASM 6.1, a single **EXTERNDEF** directive accomplishes the same task. **EXTERNDEF** lets you put global data declarations within an include file, making the data visible to all source files that include the file. For more information, see "Using EXTERNDEF" in Chapter 8.

Search Order for Include Files

MASM 6.1 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 on include files, see "Organizing Modules" in Chapter 8.

Enforcing Case Sensitivity

In MASM 5.1, sensitivity to case is influenced only by command-line options such as /MX, not the language type given with the **.MODEL** directive. In MASM 6.1, the language type takes precedence over the command-line options in specifying case sensitivity.

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 reference to an unused symbol. This prevents linkage with unneeded library code, as explained in "Using EXTERN with Library Routines," Chapter 8.

Expanded State Control

Several directives in MASM 6.1 enable or disable various aspects of the assembler control. These include 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 its compatibility with MASM 5.1. See "Using the OPTION Directive" in Chapter 1 and "Compatibility between MASM 5.1 and 6.1," later in this appendix.

The .NO87 Directive

The **.NO87** directive disables all coprocessor instructions. For more information, see Help.

The .486 and .486P Directives

MASM 6.1 can assemble instructions specific to the 80486, enabled with the **.486** directive. The **.486P** directive enables 80486 instructions at the highest privilege level (recommended for systems-level programs only). For more information, see Help.

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 6.1 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. For more information on listing files, see Appendix C or Help.

New Processor Instructions

MASM 6.1 supports these 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

For full descriptions of these instructions, see the *Reference* or Help.

Renamed Directives

Although MASM 6.1 still supports the old names in MASM 5.1, the following directives have been renamed for language consistency:

MASM 6.1	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 6.1 supports all instructions that work with the extended 32-bit registers of the 80386/486. For certain instructions, you can override the default operand

size with the **W** (word) and the **D** (doubleword) suffixes. For details, see the *Reference* or Help.

Macro Enhancements

There are significant enhancements to macro functions in MASM 6.1. Directives provide for a variable number of arguments, loop constructions, definitions of text equates, and macro functions.

Variable Arguments

MASM 5.1 ignores extra arguments passed to macros. In MASM 6.1, 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. The macro can then reference additional arguments relative to the last declared parameter. This procedure is explained in "Returning Values with Macro Functions" in Chapter 9.

Required and Default Macro Arguments

With MASM 6.1, you can use **REQ** or the := operator to specify required or default arguments. See "Specifying Required and Default Parameters" in Chapter 9.

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 "Defining Repeat Blocks with Loop Directives" in Chapter 9.

Text Macros

The **EQU** directive retains its old functionality, but MASM 6.1 also incorporates a **TEXTEQU** directive for defining text macros. **TEXTEQU** allows greater flexibility than **EQU**. For example, **TEXTEQU** can assign to a label the value calculated by a macro function. For more information, see "Text Macros" in Chapter 9.

The GOTO Directive for Macros

Within a macro definition, **GOTO** transfers assembly to a line labeled with a leading colon(**:**). For more information on **GOTO,** see 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, and return the position of a substring within a string. For information

on writing your own macro functions, see “Returning Values with Macro Functions” in Chapter 9.

Predefined Macro Functions

MASM 6.1 provides the following predefined text macro functions:

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 on predefined macros, see “String Directives and Predefined Functions” in Chapter 9.

MASM 6.1 Programming Practices

MASM 6.1 provides many features that make it easier for you to write assembly code. If you are familiar with MASM 5.1 programming, you may find it helpful to adopt the following list of new programming practices for programming with MASM 6.1. The list summarizes many of the changes covered in the following section, “Compatibility Between MASM 5.1 and 6.1.”

- 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. However, the assembler does not allow ambiguous references. You must include the structure type when referring to field names common to two or more structures.
- Separate macro arguments with commas, not spaces.
- Avoid adding extra ampersands in macros. For a list of the new rules about using ampersands in macros, see “Substitution Operator” in Chapter 9 and “OPTION OLDMACROS,” page 372.
- By default, code labels defined with a colon are local. Place two colons after code labels if you want to reference the label outside the procedure.

Compatibility Between MASM 5.1 and 6.1

MASM 6.1 provides a “compatibility mode,” making it easy for you to transfer existing MASM 5.1 code to the new version. You invoke the compatibility mode through the **OPTION M510** directive or the /Zm command-line switch. This

section explains the changes you may need to make to get your MASM 5.1 code to run under MASM 6.1 in compatibility mode.

Rewriting Code for Compatibility

In some cases, MASM 6.1 with **OPTION M510** does not support MASM 5.1 behavior. In several cases, this is because bugs in MASM 5.1 were corrected. To update your code to MASM 6.1, use the instructions in this section. This usually requires only minor changes.

Many of the topics listed here will not apply to your code. This section discusses topics in order of likelihood, beginning with the most common. In addition, you may have conflicts between identifier names and new reserved words. **OPTION NOKEYWORD** resolves errors generated from the use of reserved words as identifiers. See "OPTION NOKEYWORD," page 376, for more information.

Bug Fixes Since MASM 5.1

This section lists the differences between MASM 5.1 and MASM 6.1 due to bug corrections since MASM 5.1.

Invalid Use of LOCK, REPNE, and REPNZ

Except in compatibility mode, MASM 6.1 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	--	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, you can use both single and double quotation marks (' and ") to begin strings in macro arguments. The assembler does not generate an error or warning if the string does not end with quotation marks on a macro call. Instead, MASM 5.1 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 6.1 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.

To correct such errors in MASM 6.1, either end the string with a closing quotation mark as shown in the following 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 6.1 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 6.1 generates error A2203:

```
cannot declare scoped code label as PUBLIC
```

If you want to make a 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 6.1
```

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 6.1 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. This results in the loss of the sign bit.

MASM 6.1 allows a range of -2^{n-1} to 2^{n-1} for default values. Illegal initializers generate error A2071:

```
initializer too large for specified size
```

Design Change Issues

MASM 6.1 includes design changes that make the language more consistent. These changes are not affected by the **OPTION** directive, discussed later in this appendix. Therefore, the changes require revisions in your code. In most cases, the necessary revisions are minor and the circumstances requiring changes are rare.

Operands of Different Size

MASM 5.1 does not require operands to agree in size, as the following code illustrates:

```
.DATA?
var1    DB      ?
var2    DB      ?
.CODE
.
.
.
mov     var1, ax        ; Copy AX to word at var1
```

The operands for the **MOV** instruction do not match in size, yet the instruction assembles correctly. It places the contents of AL into `var1` and AH into `var2`, moving a word of data in one step. If the code defined `var1` as a word value, the instruction

```
mov     var1, al
```

would also assemble correctly, copying AL into the low byte of `var1` while leaving the high byte unaffected. Except at warning level 0, MASM 5.1 issues a warning to inform you of the size mismatch, but both scenarios are legal.

MASM 6.1 does not accept instructions with operands that do not agree in size. You must specifically "coerce" the size of the memory operand, like this:

```
        mov     BYTE PTR var1, al
```

Conflicting Structure Declarations

MASM 5.1 allows you to declare two or more structures with the same name. Each declaration replaces the previous declaration. However, the field names from previous declarations still remain in the assembler's list of declared values.

MASM 6.1 does not allow conflicting declarations of a structure. It generates errors A2160 through A2165 for each conflicting declaration. The errors note 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 6.1 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 some cases, MASM 5.1 accepts **HIGH** and **LOW** applied to relocatable memory expressions. 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, the instruction

```
        mov ax, LOW var2
```

is not legal. MASM 6.1 generates error A2105:

```
HIGH and LOW require immediate operands
```

The **OFFSET** operator is required on these operands in MASM 6.1, as shown in the following. Rewrite your code if necessary.

```
; MASM 6.1 code
        mov     al, LOW OFFSET var1
        mov     ah, HIGH OFFSET var2
```

OFFSET Applied to Group Names and Indirect Memory Operands

In MASM 6.1, 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 6.1, 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 6.1. For more information, see "Defining Record Variables" in Chapter 5.

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 expression

```
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 6.1. To update, rewrite the affected code.

RET Used with a Constant in Procedures with Epilogues

By default in MASM 6.1, the **RET** instruction followed by a constant suppresses automatic generation of epilogue code. MASM 5.1 ignores the operand and generates the epilogue. Remove the argument if necessary. See "Generating Prologue and Epilogue Code" in Chapter 7.

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.

In MASM 6.1, a code label defined at the beginning of a procedure refers to the first byte of the procedure after the prologue. If you need to label the start of the prologue code, place the label before the **PROC** statement. For more information, see "Generating Prologue and Epilogue Code" in Chapter 7.

Use of % as an Identifier Character

MASM 5.1 allows **%** as an identifier character. This behavior leads to ambiguities when **%** is used as the expansion operator in macros. Since **%** is not allowed as a character in MASM 6.1 identifiers, you must change the names of any identifiers containing the **%** character. For a list of legal identifier characters, see "Identifiers" in Chapter 1.

ASSUME CS Set to Wrong Value

With MASM 6.1 you do not need to use the **ASSUME** statement for the CS register. Instead, MASM 6.1 generates an automatic **ASSUME** statement for the code segment register to the current segment or group, as explained in "Setting the ASSUME Directive for Segment Registers" in Chapter 2.

Additionally, MASM 6.1 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 6.1, this results in warning A4004:

```
cannot ASSUME CS
```

To avoid this warning with MASM 6.1, delete the **ASSUME** statement for CS.

Code Requiring Two-Pass Assembly

Unlike version 5.1, MASM 6.1 does most of its work on its first pass, then performs as many subsequent passes as necessary. In contrast, MASM 5.1 always assembles in two source passes. As a result, you may need to revise or delete some pass-dependent constructs under MASM 6.1.

Two-Pass Directives

To assure compatibility, MASM 6.1 supports 5.1 directives referring to two passes. These include **.ERR1**, **.ERR2**, **IF1**, **IF2**, **ELSEIF1**, and **ELSEIF2**. For second-pass constructs, you must specify **OPTION SETIF2**, as discussed in "OPTION SETIF2," page 377. Without **OPTION SETIF2**, the **IF2** and **.ERR2** directives cause error A2061:

```
[[ELSE]]IF2/.ERR2 not allowed : single-pass assembler
```

MASM 6.1 handles first-pass constructs differently. It treats the **.ERR1** directive as **.ERR**, and the **IF1** directive as **IF**.

The following examples show you how you can rewrite typical pass-sensitive code for MASM 6.1:

- Declare `var` external only if not defined in current module:

```
; MASM 5.1:
    IF2
        IFNDEF  var
            EXTRN var:far
        ENDIF
    ENDIF

; MASM 6.1:
    EXTERNDEF   var:far
```

- Include a file of definitions only once to speed assembly:

```
; MASM 5.1:
    IF1
        INCLUDE file1.inc
    ENDIF

; MASM 6.1:
    INCLUDE FILE1.INC
```

- Generate a `%OUT` or `.ERR` message only once:

```
; MASM 5.1:
    IF2
        %OUT This is my message
    ENDIF

    IF2
        .ERRNZ A NE B
    ENDIF

; MASM 6.1:
    ECHO This is my message

    .ERRNZ A NE B    <ASSERTION FAILURE: A NE B>
```

- Generate an error if a symbol is not defined but may be forward referenced:

```
; MASM 5.1:
    IF2
        .ERRNDEF    var
    ENDIF

; MASM 6.1:
        .ERRNDEF    var
```

For information on conditional directives, see “Conditional Directives,” Chapter 1.

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 6.1 returns FALSE for the **IFDEF** expression and TRUE for the **IFNDEF** expression. When **OPTION M510** is enabled, the assembler generates warning A6005:

```
expression condition may be pass-dependent
```

To resolve the warning, place the symbol definition before the conditional test.

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, you should avoid comparisons with an address span, as in the following examples:

```
IF (OFFSET var1 - OFFSET var2) EQ 10
WHILE dx LT (OFFSET var1 - OFFSET var2)
REPEAT OFFSET var1 - OFFSET var2
```

However, the **DUP** operator allows such an expression as its count value. The assembler evaluates the **DUP** count on every pass, so even expressions involving forward references assemble correctly.

You can also use expressions containing span distances with the **.ERR** directives, since the assembler evaluates these directives after calculating all offsets:

```
.ERRE OFFSET var1 - OFFSET var2 - 10, <span incorrect>
```

.TYPE with Forward References

MASM 5.1 evaluates **.TYPE** on both assembly passes. This means it yields zero on the first pass and nonzero on the second pass, if applied to an expression that forward-references a symbol.

MASM 6.1 evaluates **.TYPE** only on the first assembly pass. As a result, if the operand references a symbol that has not yet been defined, **.TYPE** yields a value of zero. This means that **.TYPE**, if used in a conditional-assembly construction, may yield different results in MASM 6.1 than in MASM 5.1.

Obsolete Features No Longer Supported

The following two features are no longer supported by MASM 6.1. Because both are obscure features provided by early versions of the assembler, they probably do not affect your MASM 5.1 code.

The ESC Instruction

MASM 6.1 no longer supports the **ESC** instruction, which was used to send hand-coded commands to the coprocessor. Because MASM 6.1 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 6.1 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 "Storing Numbers in Floating-Point Format" in Chapter 6.

Using the OPTION Directive

The **OPTION** directive lets you control compatibility with MASM 5.1 code. This section explains the differences in MASM 5.1 and MASM 6.1 behavior that the **OPTION** directive can influence.

The **OPTION M510** directive (or /Zm command-line option) initiates all aspects of 5.1 compatibility mode. You can select from among specific characteristics of MASM 5.1 behavior with the **OPTION** arguments discussed in following sections. Each section also explains how to revise your code if you want to remove **OPTION** directives from your MASM 5.1 code.

Note If your code includes both **.MODEL** and **OPTION M510**, the **OPTION M510** statement must appear first. Wherever this appendix suggests using **OPTION M510** in your code, you can set the /Zm command-line option instead.

OPTION M510

This section discusses the **M510** argument to the **OPTION** directive, which selects the MASM 5.1 compatibility mode. In this mode, MASM 6.1 implements MASM 5.1 behavior relating to macros, offsets, scope of code labels, structures, identifier names, identifier case, and other behaviors.

The **OPTION M510** directive automatically sets the following:

```
OPTION OLDSTRUCTS       ; MASM 5.1 structures
OPTION OLDMACROS        ; MASM 5.1 macros
OPTION DOTNAME          ; Identifiers may begin with a dot (.)
OPTION SETIF2:TRUE      ; Two-pass code activates on every pass
```

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 "OPTION EXPR16," following
```

If you do not have a **.MODEL** directive in your module, **OPTION M510** adds:

```
OPTION OFFSET:SEGMENT   ; OFFSET operator defaults to
                        ;   segment-relative
                        ;   See "OPTION OFFSET," following
```

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 "OPTION NOSCOPED," following
OPTION PROC:PRIVATE     ; Labels defined with PROC are not
                        ;   public by default
                        ;   See "OPTION PROC," following
```

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 earlier.

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 you have modified your source code 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 following sections.

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, until your code uses only MASM 6.1 defaults.

Reserved Keywords Dependent on CPU Mode with OPTION M510

With **OPTION M510**, keywords and instructions 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**, any reserved word used as an identifier generates a syntax error. You can either rename the identifiers or use **OPTION NOKEYWORD**. For more information on **OPTION NOKEYWORD**, see “OPTION NOKEYWORD,” later in this appendix.

Invalid Use of Instruction Prefixes with OPTION M510

Code without **OPTION M510** generates errors for all invalid uses of the instruction prefixes. **OPTION M510** suppresses some of these errors to match MASM 5.1 behavior. MASM 5.1 does not check for illegal usage of the instruction prefixes **LOCK**, **REP**, **REPE**, **REPZ**, **REPNE**, and **REPNZ**.

Illegal usage of these prefixes results in error A2068:

```
instruction prefix not allowed
```

For more information on these instruction prefixes, see "Overview of String Instructions" in Chapter 5. See also "Bug Fixes from MASM 5.1," earlier in this appendix.

Size of Constant Operands with OPTION M510

In MASM 5.1, a large constant value that can fit only in the processor's default word (4 bytes for **.386** and **.486**, 2 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, the assembler interprets the value as a word.

Without **OPTION M510**, the assembler never assigns a size automatically. You must state it explicitly with the **PTR** operator, as shown in the following example:

```
; Without OPTION M510
        mov     [bx], WORD PTR 0100h
```

Code Labels when Defining Data 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. MASM 6.1 also allows such definitions 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 address unless the frame is explicitly specified, in which case it returns the segment address of the frame. A statement such as `SEG DGROUP:var` always returns `DGROUP`, whereas `SEG var` always returns the segment address of `var`. **OPTION M510** forces this same behavior in MASM 6.1.

If you do not use **OPTION M510**, the behavior of the **SEG** operator is determined by the **OPTION OFFSET** directive, as described in "OPTION OFFSET," later in this appendix.

In MASM 6.1, the value returned by the **SEG** operator applied to a nonexternal variable depends on compatibility mode:

- Without **OPTION M510**, **SEG** returns the address of the frame (the segment, group, or the value assumed to the segment register) if one has been explicitly set.
- With **OPTION M510**, **SEG** 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 6.1 changes the way expressions are evaluated. In MASM 5.1,

```
var-2[bx]
```

is parsed as

```
(var-2)[bx]
```

Without **OPTION M510**, you must rewrite the statement, since the assembler parses it as

```
var-(2[bx])
```

which generates an error.

Length and Size of Labels with OPTION M510

With **OPTION M510**, you can apply the **LENGTH** and **SIZE** operators to any label. For a code label, **SIZE** returns a value of 0FFFFh for **NEAR** and 0FFFEh for **FAR**. **LENGTH** always returns a value of 1. For strings, **SIZE** and **LENGTH** both return 1.

Without **OPTION M510**, **SIZE** returns values of 0FF01h, 0FF02h, 0FF04h, 0FF05h, and 0FF06h for **SHORT**, **NEAR16**, **NEAR32**, **FAR16**, and **FAR32** labels, respectively. **LENGTH** returns 1 except when used with **DUP**, in which case it returns the outermost count. For arrays initialized with **DUP**, **SIZE** returns the length multiplied by the size of the type.

The **LENGTHOF** and **SIZEOF** operators in MASM 6.1 handle arrays much more consistently. These operators return the number of data items and the number of bytes in an initializer. For a description of **SIZEOF** and **LENGTHOF**, see the following sections in Chapter 5: “Declaring and Referencing Arrays,” “Declaring and Initializing Strings,” “Defining Structure and Union Variables,” and “Defining Record Variables.”

Comparing Types Using EQ and NE with OPTION M510

With **OPTION M510**, the assembler converts types to a constant value before comparisons with **EQ** and **NE**. Code types are converted to values of 0FFFFh

(near) or 0FFFEh (far). If **OPTION M510** is not enabled, the assembler converts types to constants only when comparing them with constants. Thus, MASM 6.1 recognizes only equivalent qualified types as equal expressions.

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 how the assembler compares types with and without compatibility mode:

```
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 WORD              ; 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 WORD              ; False: SWORD NE WORD
```

Use of Constant and PTR as a Type with OPTION M510

You can use a constant as the left operand to **PTR** in compatibility mode. Otherwise, you must use a type expression. With **OPTION M510**, a constant must have a value of 1 (**BYTE**), 2 (**WORD**), 4 (**DWORD**), 6 (**FWORD**), 8 (**QWORD**) or 10 (**TBYTE**). The assembler treats the constant as the

parenthesized type. 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

MyData  WORD    0

        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

In compatibility mode, use the **PTR** operator to type-cast a constant to a structure type. This is most often done in data initializers to affect the CodeView information of the data label. Without **OPTION M510**, the assembler generates an error.

```
MYSTRC  STRUC
  f1    DB      0
MYSTRC  ENDS

MyPtr   DW      MYSTRC PTR 0    ; Illegal without OPTION M510
```

In MASM 6.1, the initializer type does not influence CodeView's type information.

Hidden Coercion of OFFSET Expression Size with OPTION M510

When programming for the 80386 or 80486, the size of an **OFFSET** expression can be 2 bytes for a symbol in a **USE16** segment, or 4 bytes for a symbol in a **USE32** or **FLAT** segment. With **OPTION M510**, you can use a 32-bit **OFFSET** expression in a 16-bit context. Without **OPTION M510**, you must use the **LOWWORD** operator to convert the offset size.

```
.386

; With OPTION M510

seg32   SEGMENT USE32
MyLabel WORD    0
seg32   ENDS

seg16   SEGMENT USE16 'code'                ; With OPTION M510:
        mov     ax,  OFFSET MyLabel         ; Legal
        mov     ax,  LOWWORD OFFSET MyLabel ; Legal
        mov     eax, OFFSET MyLabel         ; Legal
seg16   ENDS


; Without OPTION M510

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 is greater than 10 decimal, MASM 6.1 allows the radix specifiers **B** (binary) and **D** (decimal) only in compatibility mode. You must 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. If you want to keep **B** and **D** as radix specifiers when the current radix is greater than 10, you must specify **OPTION M510**. For more information about radixes, see "Integer Constants and Constant Expressions" in Chapter 1.

Naming Conventions with OPTION M510

By default, MASM 5.1 does not write the names of public variables in uppercase to the object file, even when a language type of **PASCAL, FORTRAN,** or **BASIC** is specified.

Unless you use **OPTION M510,** these language types in MASM 6.1 write identifier names in uppercase, even with the /Cp or /Cx command-line options. When you link with /NOI, case must match 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 6.1).

String Defaults in Structure Variables with OPTION M510

In compatibility mode, a constant initializer can override a structure field initialized with a string value. Without **OPTION M510**, only another string or a list can override a string initializer. 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        BYTE        "This is a string"
MTSTRUCT        ENDS

; With OPTION M510

MyInst          MTSTRUCT    <0>

; Without OPTION M510, either of these statements is correct

MyInst          MTSTRUCT    <<0>>
MyInst          MTSTRUCT    {<0>}
```

Effects of the ? Initializer in Data Definitions with OPTION M510

As described in “Declaring and Initializing Strings” in Chapter 5, the assembler treats the **?** initializer as either zero or as an unspecified value. In compatibility mode, however, the assembler always treats the **?** initializer as zero unless it is used with the **DUP** operator. In this case, the assembler allocates space, but does not initialize it with any value.

Current Address Operator with OPTION M510

In compatibility mode, the current address operator (**$**) applied to a structure returns the offset of the first byte of the structure. When **OPTION M510** is not enabled, **$** returns the offset of the current field in the structure.

Segment Association for FAR Externals with OPTION M510

In MASM 5.1, you must place an **EXTRN** directive for a variable in the same segment that holds the variable. For far data, this often entails opening and closing a segment just to place the **EXTRN** statement.

MASM 6.1 offers much greater flexibility in where **EXTERN** and **EXTERNDEF** statements can appear, as described in "Positioning External Declarations" in Chapter 8. However, in compatibility mode, MASM 6.1 emulates the behavior of MASM 5.1.

Defining Aliases Using EQU with OPTION M510

In MASM 5.1, you can equate one symbol with another. These equates are called "aliases."

Unless you specify **OPTION M510**, MASM 6.1 does not allow aliases defined with **EQU**. An immediate expression or text must appear as the right operand of an **EQU** directive. Change aliases to use the **TEXTEQU** directive, described in "Text Macros" in Chapter 9. This change may cause an expression to evaluate differently.

The following examples illustrate the differences between MASM 5.1 code, MASM 6.1 code with **OPTION M510**, and MASM 6.1 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 6.1 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 6.1 without OPTION M510
var1    EQU     3
var2    EQU     var1    ; Treated as var2 EQU 3
```

Difference in Text Macro Expansions with OPTION M510

MASM 6.1 recursively expands text macros used as values, whereas MASM 5.1 simply replaces the text macro with its value. The following example illustrates the difference:

```
; 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 6.1 requires an argument unless **OPTION M510** is enabled.

OPTION OLDSTRUCTS

This section describes changes in MASM 6.1 that apply to structures. With **OPTION OLDSTRUCTS** or **OPTION M510**:

- You can use plus operator (+) in structure field references.
- Labels and structure field names cannot have the same name with **OPTION OLDSTRUCTS**.

Plus Operator Not Allowed with Structures

By default, each reference to structure member names must use the dot operator (**.**) to separate the structure variable name from the field name. You cannot use the dot operator as the plus operator (+) or vice versa.

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 for all lines requiring change. Using the dot operator in any context other than for a structure field results in error A2166:

```
structure field expected
```

Unqualified structure references result in error A2006:

```
undefined symbol : identifier
```

The following example illustrates how to change MASM 5.1 code from the old structure references to the new type in MASM 6.1:

```
; OPTION OLDSTRUCTS (simulates MASM 5.1)
structname      STRUC
a               BYTE  ?
b               WORD  ?
structname      ENDS

structinstance  structname <>

        mov     ax, [bx].b              ; This code assembles
        mov     al, structinstance.a    ;   correctly only with
        mov     ax, [bx].4              ;   OPTION OLDSTRUCTS
                                        ;   or OPTION M510


; OPTION NOOLDSTRUCTS (the MASM 6.1 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 6.1
; Either this:
        ASSUME  bx:PTR structname
        mov     ax, [bx]
; or this:
        mov     ax, (structname PTR[bx]).b
```

Duplicate Structure Field Names

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 “Structures and Unions” in Chapter 5.

OPTION OLDMACROS

This section describes how MASM 5.1 and 6.1 differ in their handling of macros. Without **OPTION OLDMACROS** or **OPTION M510**, MASM 6.1 changes the behavior of macros in several ways. If you want the MASM 5.1 macro behavior, add **OPTION OLDMACROS** or **OPTION M510** to your MASM 5.1 code.

Separating Macro Arguments with Commas

MASM 5.1 allows white spaces or commas to separate arguments to macros. MASM 6.1 with **OPTION NOOLDMACROS** (the default) requires commas between arguments. For example, in the macro call

```
MyMacro  var1 var2 var3, var4
```

OPTION OLDMACROS causes the assembler to treat all four items as separate arguments. With **OPTION NOOLDMACROS**, the assembler treats

```
var1 var2 var3
```

as one argument, since the items are not separated with commas. To convert your macro code, replace spaces between macro arguments with a single comma.

New Behavior with Ampersands in Macros

The default **OPTION NOOLDMACROS** causes the assembler to interpret ampersands (**&**) within a macro differently than does MASM 5.1. MASM 5.1 requires one ampersand for each level of macro nesting. **OPTION OLDMACROS** emulates this behavior.

Without **OPTION OLDMACROS**, MASM 6.1 removes ampersands only once no matter how deeply nested the macro. To update your MASM 5.1 macros, follow this simple rule: replace every sequence of ampersands with a single ampersand. The only exception is when macro parameters immediately precede and follow the ampersand, and both require substitution. In this case, use two ampersands. For a description of the new rules, see “Substitution Operator” in Chapter 9.

This example shows how to update a MASM 5.1 macro:

```
; OPTION OLDMACROS (emulates 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 6.1 default)

createNames     macro   arg
    for         tail, <Next, Last>  ; FOR is the MASM 6.1
        for     num,  <1, 2>        ;   synonym for irp
        ; Define more names of the form: abcNext1?
arg&&tail&&num&?   label   BYTE
        ENDM
    ENDM
ENDM
```

OPTION DOTNAME

MASM 5.1 allows names of identifiers to begin with a period. The MASM 6.1 default is **OPTION NODOTNAME**. Adding **OPTION DOTNAME** to your code enables 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.

OPTION EXPR16

MASM 5.1 treats expressions as 16-bit words if you do not specify **.386** or **.386P** directives. MASM 6.1 by default treats expressions as 32-bit words, regardless of the CPU type. You can force MASM 6.1 to use the smaller expression size with the **OPTION EXPR16** statement.

Unless your MASM 5.1 code specifies **.386** or **.386P**, **OPTION M510** also sets 16-bit expression size. You can selectively disable this by following **OPTION M510** with the **OPTION EXPR32** directive, which sets the size back to 32 bits. You cannot have both **OPTION EXPR32** and **OPTION EXPR16** in your program.

It may not be easy to determine the effect of changing from 16-bit internal expression size to 32-bit size. In most cases, the 32-bit word size does not affect the MASM 5.1 code. However, problems may arise because of differences in

intermediate values during evaluation of expressions. You can compare the files for differences by generating listing files with the /Fl and /Sa command-line options with and without **OPTION EXPR16.**

OPTION OFFSET

The information in this section is relevant only if your MASM 5.1 code does not use the **.MODEL** directive. With no **.MODEL**, MASM 5.1 computes offsets from the start of the segment, whereas MASM 6.1 computes offsets from the start of the group. (With **.MODEL**, MASM 5.1 also computes offsets from the start of the group.)

To force MASM 6.1 to emulate 5.1 behavior, specify either **OFFSET:SEGMENT** or **OPTION M510**. Both directives cause the assembler to compute offsets relative to the segment if you do not include **.MODEL**.

To selectively enable MASM 6.1 behavior, place the directive **OPTION OFFSET:GROUP** after **OPTION M510**. In this case, you should ensure each **OFFSET** statement has a segment override where appropriate. The following example shows how **OPTION OFFSET:SEGMENT** affects code written for MASM 5.1:

```
OPTION  OFFSET:SEGMENT
MyGroup GROUP   MySeg

MySeg   SEGMENT 'data'
MyLabel LABEL   BYTE
        DW      OFFSET MyLabel          ; Relative to MySeg
        DW      OFFSET MyGroup:MyLabel  ; Relative to MyGroup
        DW      OFFSET MySeg:MyLabel    ; Relative to MySeg
MySeg   ENDS
```

In the preceding example, the first **OFFSET** statement computes the offset of `MyLabel` relative to `MySeg`. Without **OFFSET:SEGMENT**, MASM 6.1 returns the offset relative to `MyGroup`. To maintain the correct behavior with **OFFSET:GROUP**, specify a segment override, as shown in the following. The other two **OFFSET** statements already include overrides, and so do not require modification.

```
OPTION  OFFSET:GROUP
MyGroup GROUP   MySeg

MySeg   SEGMENT 'data'
MyLabel LABEL   BYTE
        DW      OFFSET MySeg:MyLabel    ; Relative to MySeg
        DW      OFFSET MyGroup:MyLabel  ; Relative to MyGroup
        DW      OFFSET MySeg:MyLabel    ; Relative to MySeg
MySeg   ENDS
```

When not in compatibility mode, the **OPTION OFFSET** directive determines whether the **SEG** operator returns a value relative to the group or segment. With **OPTION M510**, **SEG** is always segment-relative by default, regardless of the current value of **OPTION OFFSET**.

OPTION NOSCOPED

The information in this section applies only if the **.MODEL** directive in your MASM 5.1 code does not specify a language type. Without a language type, MASM 5.1 assumes code labels in procedures have no "scope"—that is, the labels are not local to the procedure. When not in compatibility mode, MASM 6.1 always gives scope to code labels, even without a language type.

To force MASM 5.1 behavior, specify either **OPTION M510** or **OPTION NOSCOPED** in your code. To selectively enable MASM 6.1 behavior, place the directive **OPTION SCOPED** after **OPTION M510**.

To determine which labels require change, assemble the module without the **OPTION NOSCOPED** directive. For each reference to a label that is not local, the assembler generates error A2006:

```
undefined symbol : identifier
```

OPTION PROC

The information in this section applies only if the **.MODEL** directive in your MASM 5.1 code does not specify a language type. Without a language type, MASM 5.1 makes procedures private to the module. By default, MASM 6.1 makes procedures public. You can explicitly change the default visibility to private with either **OPTION M510**, **OPTION PROC:PRIVATE**, or **OPTION PROC:EXPORT**.

To selectively enable MASM 6.1 behavior, place the directive **OPTION PROC:PUBLIC** after **OPTION M510**. You can override the default by adding the **PUBLIC** or **PRIVATE** keyword to selected procedures. The following example shows how to change MASM 5.1 code to keep a procedure private:

```
; MASM 5.1 (OPTION PROC:PRIVATE)
MyProc PROC NEAR

; MASM 6.1 (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.

OPTION NOKEYWORD

MASM 6.1 has several new keywords that MASM 5.1 does not recognize as reserved. To resolve any conflicts, you can:

- Rename any offending symbols in your code.
- Selectively disable keywords with the **OPTION NOKEYWORD** directive.

The second option lets you retain the offending symbol names in your code by forcing MASM 6.1 to not recognize them as keywords. For example,

```
OPTION NOKEYWORD:<INVOKE STRUCT>
```

removes the keywords **INVOKE** and **STRUCT** from the assembler's list of reserved words. However, you cannot then use the keywords in their intended function, since the assembler no longer recognizes them.

The following list shows MASM 6.1 reserved words new since MASM 5.1:

.BREAK	**.UNTILCXZ**	**FRSTORD**
.CONTINUE	**.WHILE**	**FRSTORW**
.DOSSEG	**ADDR**	**FSAVED**
.ELSE	**ALIAS**	**FSAVEW**
.ELSEIF	**BSWAP**	**FSTENVD**
.ENDIF	**CARRY?**	**FSTENVW**
.ENDW	**CMPXCHG**	**GOTO**
.EXIT	**ECHO**	**HIGHWORD**
.IF	**EXTERN**	**INVD**
.LISTALL	**EXTERNDEF**	**INVLPG**
.LISTIF	**FAR16**	**INVOKE**
.LISTMACRO	**FAR32**	**IRETDF**
.LISTMACROALL	**FLAT**	**IRETF**
.NO87	**FLDENVD**	**LENGTHOF**
.NOCREF	**FLDENVW**	**LOOPD**
.NOLIST	**FNSAVED**	**LOOPED**
.NOLISTIF	**FNSAVEW**	**LOOPEW**
.NOLISTMACRO	**FNSTENVD**	**LOOPNED**
.REPEAT	**FNSTENVW**	**LOOPNEW**
.STARTUP	**FOR**	**LOOPNZD**
.UNTIL	**FORC**	**LOOPNZW**

LOOPW	**PUSHCONTEXT**	**SWORD**
LOOPZW	**PUSHD**	**SYSCALL**
LOWWORD	**PUSHW**	**TEXTEQU**
LROFFSET	**REAL10**	**TR3**
NEAR16	**REAL4**	**TR4**
NEAR32	**REAL8**	**TR5**
OPATTR	**REPEAT**	**TYPEDEF**
OPTION	**SBYTE**	**UNION**
OVERFLOW?	**SDWORD**	**VARARG**
PARITY?	**SIGN?**	**WBINVD**
POPAW	**SIZEOF**	**WHILE**
POPCONTEXT	**STDCALL**	**XADD**
PROTO	**STRUCT**	**ZERO?**
PUSHAW	**SUBTITLE**	

OPTION SETIF2

By default, MASM 6.1 does not recognize pass-dependent constructs. Both the **OPTION M510** and **OPTION SETIF2** statements force MASM 6.1 to handle MASM 5.1 constructs that activate on the second assembly pass, such as **.ERR2**, **IF2**, and **ELSEIF2**.

Invoke the option like this:

OPTION SETIF2: {TRUE | FALSE}

When set to **TRUE**, **OPTION SETIF2** forces all second-pass constructs to activate on every assembly pass. When set to **FALSE**, second-pass constructs do not activate on any pass. **OPTION M510** implies **OPTION SETIF2:TRUE**.

Changes to Instruction Encodings

MASM 6.1 contains changes to the encodings for several instructions. In some cases, the changes help optimize code size.

Coprocessor Instructions

For the 8087 coprocessor, MASM 5.1 adds an extra **NOP** before the no-wait versions of coprocessor instructions. MASM 6.1 does not. In the rare case that the missing **NOP** affects timing, insert **NOP**.

For the 80287 coprocessor or better, MASM 5.1 inserts **FWAIT** before certain instructions. MASM 6.1 does not prefix any 80287, 80387, or 80486 coprocessor instruction with **FWAIT**, except for wait forms of instructions that have a no-wait form.

RET Instruction

MASM 5.1 generates a 3-byte encoding for **RET**, **RETN**, or **RETF** instructions with an operand value of zero, unless the operand is an external absolute. In this case, MASM 5.1 ignores the parameter and generates a 1-byte encoding.

MASM 6.1 does the opposite. It ignores a zero operand for the return instructions and generates a 1-byte encoding, unless the operand is an external absolute. In this case, MASM 6.1 generates a 3-byte encoding.

Thus, you can suppress epilogue code in a procedure but still specify the default size for **RET** by coding the return as

```
        ret     0
```

Arithmetic Instructions

Versions 5.1 and 6.1 differ in the way they encode the arithmetic instructions **ADC, ADD**, **AND**, **CMP**, **OR**, **SUB**, **SBB**, and **XOR**, under the following conditions:

- The first operand is either AX or EAX.
- The second operand is a constant value between 0 and 127.

For the AX register, there is no size or speed difference between the two encodings. For the EAX register, the encoding in MASM 6.1 is 2 bytes smaller. The **OPTION NOSIGNEXTEND** directive forces the MASM 5.1 behavior for **AND**, **OR**,
and **XOR**.

APPENDIX B

BNF Grammar

This appendix provides a complete description of symbols, operators, and directives for MASM 6.1. It uses the Backus-Naur Form (BNF) for grammar notation. You can use BNF grammar to determine the exact syntax for any language component and find all available options for any MASM command.

BNF definitions consist of “nonterminals” and “terminals.” Nonterminals are placeholders within a BNF definition, defined elsewhere in the BNF grammar. Terminals are endpoints in a BNF definition, consisting of MASM 6.1 keywords. In this Appendix, all nonterminals appear in italics type and all terminals appear in bold type.

BNF 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.
.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 the BNF Grammar

To illustrate the use of the BNF, Figure B.1 diagrams the definition of the **TYPEDEF** directive, starting with the nonterminal *typedefDir*.

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 italicized nonterminal in the *typedefDir* definition is also an entry in the BNF. Three vertical dots indicate a branching definition for a nonterminal that, for the sake of simplicity, this figure does not illustrate.

The BNF grammar allows recursive definitions. For example, the grammar uses *qualifiedType* as a possible definition for *qualifiedType*, which 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*

Nonterminal	Definition
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* ⟧
bool	**TRUE** | **FALSE**
byteRegister	AL | AH | BL | BH | CL | CH | DL | DH
cExpr	*aExpr* | *cExpr* ‖ *aExpr*
character	Any character with 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 ;;*

Nonterminal	Definition
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* ⟧
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 ;;*

Nonterminal	Definition
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* except *whiteSpaceCharacter*
digits	*decdigit* | *digits decdigit* | *digits hexdigit*
directive	*generalDir* | *segmentDef*
directiveList	*directive* | *directiveList directive*
distance	*nearfar* | **NEAR16** | **NEAR32** | **FAR16** | **FAR32**
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*

Nonterminal	Definition
e09	**OFFSET** *e10* | **SEG** *e10* | **LROFFSET** *e10* | **TYPE** *e10* | **THIS** *e10* | *e09* **PTR** *e10* | *e09* **:** *e10* | *e10*
e10	*e10* **.** *e11* | *e10* **⟦** *expr* **⟧** | *e11*
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 ;;* **%OUT** *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**

Nonterminal	Definition
endDir	**END** ⟦ *immExpr* ⟧ *;;*
endpDir	*procId* **ENDP** *;;*
endsDir	*id* **ENDS** *;;*
equDir	*textMacroId* **EQU** *equType ;;*
equType	*immExpr* | *textLiteral*
errorDir	*errorOpt ;;*
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*

Nonterminal	Definition
fieldInitList	*fieldInit* | *fieldInitList* **,** ⟦ *;;* ⟧ *fieldInit*
fileChar	*delimiter*
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	**SEG** *id* | DGROUP **:** *id* | *segmentRegister* **:** *id* | *id*
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*
hexdigit	a | b | c | d | e | f | A | B | C | D | E | F

Nonterminal	Definition
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 ;;*
localDir	**LOCAL** *parmList ;;*
localDirList	*localDir* | *localDirList localDir*

Nonterminal	Definition
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*
macroRepeat	*repeatDir constExpr ;;* *macroBody* **ENDM** *;;*
macroStmt	*directive* | *exitmDir* | **:** *macroLabel* | **GOTO** *macroLabel*

Nonterminal	Definition
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* | *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* ⟧
optionDir	**OPTION** *optionList ;;*

Nonterminal	Definition
optionItem	**CASEMAP :** *mapType* | **DOTNAME** | **NODOTNAME** | **EMULATOR** | **NOEMULATOR** | **EPILOGUE :** *macroId* | **EXPR16** | **EXPR32** | **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* | **SETIF2 :** *bool*
optionList	*optionItem* | *optionList* **,** ⟦ *;;* ⟧ *optionItem*
optText	**,** *textItem*
orOp	**OR** | **XOR**
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* ⟧
primary	*expr binaryOp expr* | *flagName* | *expr*

Nonterminal	Definition
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 ;;*
radixOverride	h | o | q | t | y | H | O | Q | T | Y
recordConst	*recordTag* **{** *oldRecordFieldList* **}** | *recordTag* < *oldRecordFieldList* >
recordDir	*recordTag* **RECORD** *bitDefList ;;*
recordFieldList	⟦ *constExpr* ⟧ | *recordFieldList* **,** ⟦ *;;* ⟧ ⟦ *constExpr* ⟧

Nonterminal	Definition
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**
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

Nonterminal	Definition
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	**:** | **.** | **[** | **]** | **(** | **)** | < | > | **{** | **}** | **+** | **-** | **/** | ***** | **&** | **%** | **!** | **'** | \ | **=** | **;** | **,** | **"** | *whiteSpaceCharacter* | endOfLine
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* except *quote*
structBody	*structItem ;;* | *structBody structItem ;;*
structDir	*structTag structHdr* [*fieldAlign*] [**, NONUNIQUE**] *;;* *structBody* *structTag* **ENDS** *;;*
structHdr	**STRUC** | **STRUCT** | **UNION**

Nonterminal	Definition
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 ;;*
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*

Nonterminal	Definition
typeId	*id*
unionTag	*id*
untilDir	**.UNTIL** *cExpr ;;* **.UNTILCXZ** ⟦ *cxzExpr* ⟧ *;;*
usesRegs	**USES** *regList*
whileBlock	**.WHILE** *cExpr ;;* *blockStatements ;;* **.ENDW**
whiteSpaceCharacter	ASCII 8, 9, 11–13, 26, 32

A P P E N D I X

Generating and Reading Assembly Listings

A listing file shows precisely how the assembler translates your source file into machine code. The listing documents the assembler's assumptions, memory allocations, and optimizations.

MASM creates an assembly listing of your source file whenever you do one of the following:

- Select the appropriate option in PWB.
- Use one of the related source code directives.
- 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, and places the tables at the end of the assembly listing. Only the types of symbols encountered in the program are included. For example, if your program has no macros, the symbol table does not have a macros section.

Generating Listing Files

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 dialog box for Set Debug or Release Options lists the choices summarized in Table C.1. This table also shows the equivalent source code directives and command-line options.

Table C.1 Options for Generating or Modifying Listing Files

To generate this information:

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 instruction timings	List Instruction Timings	—	/Fl /Sc
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 *name*
Specify subtitle for page	—	**SUBTITLE** *name*	/Ss *name*
Designate page length and line width, increment section number, or generate page breaks	—	**PAGE** [[*length*,*width*]][[+]]	/Sp *length* /Sl *width*
Generate first-pass listing	—	—	/Ep

[1] Select MASM Options from the Options menu, then choose Set Dialog Options from the MASM Options dialog box.

[2] See “Conditional Directives” in Chapter 1

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 following precedence levels. Selecting PWB options is equivalent to specifying /Fl /S*x* 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 can override these command-line options.

Reading the Listing File

The first half of the listing shows macros from the include file DOS.MAC, structure declarations, and data. After the **.DATA** directive, the columns on the left show offsets and initialized byte values within the data segment.

Instructions begin after the **.CODE** directive. The three columns on the left show offsets, instruction timings, and binary code generated by the assembler. The columns on the right list the source statements exactly as they appear in the source file or as expanded by a macro. Various symbols and abbreviations in the middle column provide information about the code, as explained in the following section. The subsequent section, “Symbols and Abbreviations,” explains the meanings of listing symbols.

Generated Code

The assembler lists the code generated from the statements of a source file. With the /Sc command-line switch, which generates instruction timings, each line has this syntax:

offset [[*timing*]] [[*code*]]

The *offset* is the offset from the beginning of the current code segment. The *timing* shows the number of cycles the processor needs to execute the instruction. The value of *timing* reflects the CPU type; for example, specifying the **.386** directive produces instruction timings for the 80386 processor. 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.

When assembling under the default **.8086** directive, *timing* includes an effective address value if the instruction accesses memory. The 80186/486 processors do not use effective address values. For more information on effective address timing, see the "Processor" section in the *Reference* book.

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:

```
mov     ax, [dx][di]
listtst.asm(77): error A2031: must be index or base register
```

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 example listing was produced using "List Generated Instructions" and "List Instruction Timings" in PWB. These options correspond to the ML command-line switches /Fl /Sg /Sc.

Table C.2 Symbols and Abbreviations in Listings

Character	Meaning
C	Line from include file
=	**EQU** or equal-sign (=) directive
nn[xx]	**DUP** expression: *nn* copies of the value *xx*
----	Segment/group address (linker must resolve)
R	Relocatable address (linker must resolve)
*	Assembler-generated code
E	External address (linker must resolve)
n	Macro-expansion nesting level (+ if more than 9)
|	Operator size override
&	Address size override
nn:	Segment override in statement
nn/	**REP** or **LOCK** prefix instruction

Table C.3 explains the five symbols that may follow timing values in your listing. The *Reference* book will help you determine correct timings for those values marked with a symbol.

Table C.3 Symbols in Timing Column

Symbol	Meaning
m	Add cycles depending on next executed instruction.
n	Add cycles depending on number of iterations or size of data.
p	Different timing value in protected mode.
+	Add cycles depending on operands or combination of the preceding.
,	Separates two values for "jump taken" and "jump not taken."

```
Microsoft (R) Macro Assembler Version 6.10   09/20/00   12:00:00
listtst.asm                                                  Page 1 - 1


                                        .MODEL  small, c
                                        .386
                                        .DOSSEG
                                        .STACK  256
                                        INCLUDE dos.mac
                                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
 = 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  01                         month   BYTE    1
 0001  01                         day     BYTE    1
 0002  0000                       year    WORD    ?
                                  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 01 14 07C9                     today   DATE    <01, 20, 1993>
0009 00                             flag    BYTE    0
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 24              1          BYTE    13d, 10d, '$'
= 0009                      1  lending EQU     LENGTHOF ending
0052 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
0063 ---- 0052 R                    FPMSG   FPBYTE          Msg
                                    PBYTE   TYPEDEF         PTR BYTE
                                    NPWORD  TYPEDEF NEAR    PTR WORD
                                    PVOID   TYPEDEF         PTR
                                    PPBYTE  TYPEDEF         PTR PBYTE

0000                                        .CODE
                                            .STARTUP
0000                        *@Startup:
0000   2   B8 ---- R        *        mov    ax, DGROUP
0003   2p  8E D8            *        mov    ds, ax
0005   2   8C D3            *        mov    bx, ss
0007   2   2B D8            *        sub    bx, ax
0009   3   C1 E3 04         *        shl    bx, 004h
000C   2p  8E D0            *        mov    ss, ax
000E   2   03 E3            *        add    sp, bx

                                    EXTERNDEF       work:NEAR
0010   7m  E8 0000 E                        call    work
```

```
                                        INVOKE  PutStr, ADDR msg
0013   2   68 0052 R      *        push   OFFSET Msg
0016   7m  E8 0029        *        call   PutStr
0019   2   83 C4 02       *        add    sp, 00002h

001C   2   B8 ---- R                   mov     ax, @data
001F   2p  8E C0                       mov     es, ax
0021   2   B0 63                       mov     al, 'c'
0023   4   26: 8B 0E                   mov     cx, es:num
           0020
0028   2   BF 0052                     mov     di, 82
002B   7n  F2/ AE                      repne   scasb
002D   4   66| A1 0000 R               mov     eax, ddData
0031   6   67& FE 03                   inc     BYTE PTR [ebx]

                                EXTERNDEF       morework:NEAR
0034   7m  E8 0000 E                   call    morework

                                       Display ending
0037   2   B4 09            1          mov     ah, 09h
0039   2   BA 0046 R        1          mov     dx, OFFSET ending
003C  37   CD 21            1          int     21h

                                      .EXIT
003E   2   B4 4C          *        mov    ah, 04Ch
0040  37   CD 21          *        int    021h



0042                            PutStr  PROC    pMsg:PTR BYTE

0042   2   55             *        push   bp
0043   4   8B EC          *        mov    bp, sp
0045   2   B4 02                       mov     ah, 02H
0047   4   8B 7E 04                    mov     di, pMsg
004A   4   8A 15                       mov     dl, [di]
                                       mov     ax, [dx][di]
listtst.asm(77): error A2031: must be index or base register

                                       .WHILE  (dl)
004C   7m  EB 10          *        jmp    @C0001
0059                      *@C0002:
0059  37   CD 21                       int     21h
005B   2   47                          inc     di
005C   4   8A 15                       mov     dl, [di]
                                       .ENDW
005E                      *@C0001:
005E   2   0A D2          *        or  dl, dl
0060 7m,3  75 F7          *        jne    @C0002
                                       ret
```

```
0062   4   5D            *        pop    bp
0063  10m  C3            *        ret    00000h
0064                           PutStr  ENDP

                               END
```

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 previous sample listing, but are summarized as follows.

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**, and so on). "Class" gives the segment's class (**CODE**, **DATA**, **STACK**, or **CONST**).

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 6.1 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” (later in this appendix) 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 since MASM 5.1.

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**	**PARA**
COMPACT	**MEMORY**	**PRIVATE***
CPU*	**NEARSTACK***	**PROLOGUE***
DOTNAME*	**NODOTNAME***	**RADIX***
EMULATOR*	**NOEMULATOR***	**READONLY***
EPILOGUE*	**NOKEYWORD***	**REQ***
ERROR*	**NOLJMP***	**SCOPED***
EXPORT*	**NOM510***	**SETIF2***
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 not case sensitive.

$	**DWORD**	**PASCAL**
?	**FAR**	**QWORD**
@B	**FAR16***	**REAL4***
@F	**FORTRAN**	**REAL8***
ADDR*	**FWORD**	**REAL10***
BASIC	**NEAR**	**SBYTE***
BYTE	**NEAR16***	**SDWORD***
C	**OVERFLOW?***	**SIGN?***
CARRY?*	**PARITY?***	**STDCALL***

SWORD*	**TBYTE**	**WORD**
SYSCALL*	**VARARG***	**ZERO?***

Special Operands for the 80386/486

FLAT*	**NEAR32***	**FAR32***

Predefined Symbols

Unlike most MASM reserved words, predefined symbols are case sensitive.

@CatStr*	**@Environ***	**@Model***
@code	**@fardata**	**@SizeStr***
@CodeSize	**@fardata?**	**@stack***
@Cpu	**@FileCur***	**@SubStr***
@CurSeg	**@FileName**	**@Time***
@data	**@InStr***	**@Version**
@DataSize	**@Interface***	**@WordSize**
@Date*	**@Line***	

Registers

AH	**CS**	**DX**	**SI**
AL	**CX**	**EAX**	**SP**
AX	**DH**	**EBP**	**SS**
BH	**DI**	**EBX**	**ST**
BL	**DL**	**ECX**	**TR3***
BP	**DR0**	**EDI**	**TR4***
BX	**DR1**	**EDX**	**TR5***
CH	**DR2**	**ES**	**TR6**
CL	**DR3**	**ESI**	**TR7**
CR0	**DR6**	**ESP**	
CR2	**DR7**	**FS**	
CR3	**DS**	**GS**	

Operators and Directives

.186	.ERRDIFI	.TYPE
.286	.ERRE	.UNTIL*
.286C	.ERRIDN	.UNTILCXZ*
.286P	.ERRIDNI	.WHILE*
.287	.ERRNB	.XALL
.386	.ERRNDEF	.XCREF
.386C	.ERRNZ	.XLIST
.386P	.EXIT*	ALIAS*
.387	.FARDATA	ALIGN
.486*	.FARDATA?	ASSUME
.486P*	.IF*	CATSTR
.8086	.LALL	COMM
.8087	.LFCOND	COMMENT
.ALPHA	.LIST	DB
.BREAK*	.LISTALL*	DD
.CODE	.LISTIF*	DF
.CONST	.LISTMACRO*	DOSSEG
.CONTINUE*	.LISTMACROALL*	DQ
.CREF	.MODEL	DT
.DATA	.NO87*	DUP
.DATA?	.NOCREF*	DW
.DOSSEG*	.NOLIST*	ECHO*
.ELSE*	.NOLISTIF*	ELSE
.ELSEIF*	.NOLISTMACRO*	ELSEIF
.ENDIF*	.RADIX	ELSEIF1
.ENDW*	.REPEAT*	ELSEIF2
.ERR	.SALL	ELSEIFB
.ERR1	.SEQ	ELSEIFDEF
.ERR2	.SFCOND	ELSEIFDIF
.ERRB	.STACK	ELSEIFDIFI
.ERRDEF	.STARTUP*	ELSEIFE
.ERRDIF	.TFCOND	ELSEIFIDN

ELSEIFIDNI
ELSEIFNB
ELSEIFNDEF
END
ENDIF
ENDM
ENDP
ENDS
EQ
EQU
EVEN
EXITM
EXTERN*
EXTERNDEF*
EXTRN
FOR*
FORC*
GE
GOTO*
GROUP
GT
HIGH
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

Processor Instructions

Processor instructions are not case sensitive.

8086/8088 Processor Instructions

AAA	**JAE**	**LDS**
AAD	**JB**	**LEA**
AAM	**JBE**	**LES**
AAS	**JC**	**LODS**
ADC	**JCXZ**	**LODSB**
ADD	**JE**	**LODSW**
AND	**JG**	**LOOP**
CALL	**JGE**	**LOOPE**
CBW	**JL**	**LOOPEW***
CLC	**JLE**	**LOOPNE**
CLD	**JMP**	**LOOPNEW***
CLI	**JNA**	**LOOPNZ**
CMC	**JNAE**	**LOOPNZW***
CMP	**JNB**	**LOOPW***
CMPS	**JNBE**	**LOOPZ**
CMPSB	**JNC**	**LOOPZW***
CMPSW	**JNE**	**MOV**
CWD	**JNG**	**MOVS**
DAA	**JNGE**	**MOVSB**
DAS	**JNL**	**MOVSW**
DEC	**JNLE**	**MUL**
DIV	**JNO**	**NEG**
ESC	**JNP**	**NOP**
HLT	**JNS**	**NOT**
IDIV	**JNZ**	**OR**
IMUL	**JO**	**OUT**
IN	**JP**	**POP**
INC	**JPE**	**POPF**
INT	**JPO**	**PUSH**
INTO	**JS**	**PUSHF**
IRET	**JZ**	**RCL**
JA	**LAHF**	**RCR**

RET	**SCAS**	**STOSB**
RETF	**SCASB**	**STOSW**
RETN	**SCASW**	**SUB**
ROL	**SHL**	**TEST**
ROR	**SHR**	**WAIT**
SAHF	**STC**	**XCHG**
SAL	**STD**	**XLAT**
SAR	**STI**	**XLATB**
SBB	**STOS**	**XOR**

80186 Processor Instructions

BOUND	**INSW**	**OUTSW**
ENTER	**LEAVE**	**POPA**
INS	**OUTS**	**PUSHA**
INSB	**OUTSB**	**PUSHW***

80286 Processor Instructions

ARPL	**SIDT**	**VERR**
LAR	**SLDT**	**VERW**
LSL	**SMSW**	
SGDT	**STR**	

80286 and 80386 Privileged-Mode Instructions

CLTS	**LIDT**	**LMSW**
LGDT	**LLDT**	**LTR**

80386 Processor Instructions

BSF	**CDQ**	**IRETF***
BSR	**CMPSD**	**JECXZ**
BT	**CWDE**	**LFS**
BTC	**INSD**	**LGS**
BTR	**IRETD**	**LODSD**
BTS	**IRETDF***	**LOOPD***

LOOPED*	**SETAE**	**SETNGE**
LOOPNED*	**SETB**	**SETNL**
LOOPNZD*	**SETBE**	**SETNLE**
LOOPZD*	**SETC**	**SETNO**
LSS	**SETE**	**SETNP**
MOVSD	**SETG**	**SETNS**
MOVSX	**SETGE**	**SETNZ**
MOVZX	**SETL**	**SETO**
OUTSD	**SETLE**	**SETP**
POPAD	**SETNA**	**SETPE**
POPFD	**SETNAE**	**SETPO**
PUSHAD	**SETNB**	**SETS**
PUSHD*	**SETNBE**	**SETZ**
PUSHFD	**SETNC**	**SHLD**
SCASD	**SETNE**	**SHRD**
SETA	**SETNG**	**STOSD**

80486 Processor Instructions

BSWAP*	**INVD***	**WBINVD***
CMPXCHG*	**INVLPG***	**XADD***

Instruction Prefixes

LOCK	**REPE**	**REPNZ**
REP	**REPNE**	**REPZ**

Coprocessor Instructions

Coprocessor instructions are not case sensitive.

8087 Coprocessor Instructions

F2XM1	**FADDP**	**FCHS**
FABS	**FBLD**	**FCLEX**
FADD	**FBSTP**	**FCOM**

FCOMP
FCOMPP
FDECSTP
FDISI
FDIV
FDIVP
FDIVR
FDIVRP
FENI
FFREE
FIADD
FICOM
FICOMP
FIDIV
FIDIVR
FILD
FIMUL
FINCSTP
FINIT
FIST
FISTP
FISUB
FISUBR
FLD
FLD1
FLDCW
FLDENV
FLDENVW*
FLDL2E
FLDL2T
FLDLG2
FLDLN2
FLDPI
FLDZ
FMUL
FMULP
FNCLEX
FNDISI
FNENI
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

80287 Privileged-Mode Instruction

FSETPM

80387 Instructions

FCOS
FLDENVD*
FNSAVED*
FNSTENVD*
FPREM1
FRSTORD*
FSAVED*
FSIN
FSINCOS
FSTENVD*
FUCOM
FUCOMP

FUCOMPP

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**

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. 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 programming 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.

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 1-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 4 bits of 1 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 8 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 C standard for calling a procedure—that is, pushing arguments onto the stack from right to left (in reverse order from the way they appear in the argument list). 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,” “hooking (an interrupt).”

character string See “string.”

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, 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."

description file A text file used as input for the NMAKE utility.

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 8 bytes of memory (MASM type **REAL8**). Double-precision values are accurate to 15 or 16 digits.

doubleword A 4-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 or print the contents of memory in 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. Windows extends the dynamic-link mechanism to serve as the primary method by which all system and nonsystem services are obtained. See "linking."

dynamic-link library (DLL) A library file that contains the executable code for a group of dynamically linked routines.

dynamic-link routine A routine in a dynamic-link library that can be linked at load time or run time.

E

element A single member variable of an array of like variables.

environment block The section of memory containing the MS-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

far address A memory location specified with a segment value plus an offset from the start of that segment. Far addresses require 4 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 4 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.

handler See "interrupt handler."

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, and FORTRAN 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 be either 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.

import library A pseudo library that contains addresses rather than executable code. The linker reads the addresses from an import library to resolve references to external dynamic-link library routines.

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. See "pointer."

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 A signal to the processor to halt its current operation and immediately transfer control to an interrupt handler. Interrupts are triggered either by hardware, as when the keyboard detects a keypress, or by software, as when a program executes the **INT** instruction. See "interrupt handler."

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. 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 that contains modules of compiled code. MS-DOS programs use normal run-time libraries, from which the linker extracts modules and combines them with other object modules to create executable program files. Windows-based programs can use dynamic-link libraries (see), which the operating system loads and links to calling programs. See also "import library."

linked list A data structure in which each entry includes a pointer to the location of the adjoining entries.

linking In normal static linking, the process in which the linker resolves all external references by searching run-time and user libraries, and then computes absolute offset addresses for these references. Static linking results in a single executable file. In dynamic linking (see), the operating system, rather than the linker, provides the addresses after loading the modules into separate parts of memory.

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.

logical segment A memory area in which a program stores code, data, or stack information. See "physical segment."

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-definition file A text file containing information that the linker uses to create a Windows-based program.

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." 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 2 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.

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 is larger than the allowable limit 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 segment The true memory address of a segment, referenced through a segment register.

pointer A variable containing the address of another variable. See “indirect memory operand.”

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 6.1 ensures 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 that 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 be used only by a protected-mode operating system. Special privileged instructions are enabled by **.286P**, **.386P**, and **.486P**. Privileged mode 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 data and addresses supplied by MS-DOS that a program can read during execution.

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 be both 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 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 processor flags.

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 running. See "dynamic linking."

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 A value that indirectly references a segment address. A protected-mode operating system, such as Windows, assigns selector values to programs, which use them as segment addresses. If a program attempts to use an unassigned selector, it triggers a General-Protection fault (see).

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 MS-DOS commands, then return to the debugger.

sign extended The process of widening an integer (for example, going from a byte to a word, or a word to a doubleword) while retaining its correct value and sign.

signed integer An integer value that uses the most-significant bit to represent the value's sign. If the bit is one, the number is negative; if zero, the number is positive. See "two's complement," "unsigned integer," "MSB."

single precision A real (floating-point) value that occupies 4 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. MS-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 An area of memory in which data items are consecutively stored 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.

stack switching Changing the stack pointers to point to another stack area.

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.

status bar See "linking."

static linking 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 variables that 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.

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 An integer in which the most-significant bit serves as part of the number, rather than as an indication of sign. For example, an unsigned byte integer can have a value from 0 to 255. A signed byte integer, which reserves its eighth bit for the sign, can range from -127 to +128. See "signed integer," "MSB."

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 (2 bytes). It can store values from 0 to 65,535 (or -32,768 to +32,767).

Index

! (literal-character operator) 235
!= (not equal operator) 178
” (double quotation marks) 109, 353
$ (current address operator) 368
% (expansion operator) 235, 248, 357
& (substitution operator) 238, 372
&& (logical AND operator) 178
’ (single quotation mark) 109, 353
( ) (parentheses) 106
+ (plus operator) 63, 66, 352, 370
. (dot operator) 126
. (structure-member operator) 64, 67, 352, 370
.186 directive 38
.286 directive 38
.286P directive 38
.287 directive 38
.386 directive
    FLAT, with 26, 36
    processor mode, specifying 38, 336
    segment mode, setting 46, 68
.386P directive 38
.387 directive 38
.486 directive
    FLAT, with 36
    processor mode, specifying 38
    segment mode, setting 46, 68
.486P directive 38
.8087 directive 38
: (colon) 22, 352, 354
: (segment-override operator) 50, 59–60, 64
:: (double colon) 197, 215, 352–354
; (semicolon) 21
;; (double semicolon) 227
< (less than operator) 178
< > (angle brackets) *See* Angle brackets
= = (equal operator) 178
> (greater than operator) 178
? (question mark initializer)
    array elements 109
    described 368
    variables 87
@ (at sign) 10
@@: (anonymous label) 170
[ ] (brackets) 107
[ ] (index operator) 63
\ (backslash character), MASM code 22
\ (line-continuation character) 121
{} (curly braces) 121, 131
|| (logical OR operator) 178
32-bit programming 335
80186 processor 3
80188 processor 3
80286 processor 3
80287 math coprocessor 3, 135
80386 processor 3, 335
80387 math coprocessor 3, 135
80486 processor 3, 135
8086-based processors 2–3
8087 math coprocessor 3, 135
8088 processor 3

A

AAD instruction 160
AAM instruction 160
ABS operand 220
Accessing data with pointers *See* Pointer variables
ADC instruction 92–94
ADD instruction 92–94
ADDR operator 197
Addresses
    displacement of 65
    dynamic 79
    effective 65
    errors in 54
    far 57, 74, 80
    near 57, 80
    physical 7
    registers, loading into 80
    relocatable 57
    segmented 7–8, 53
Addressing
    direct registers, used in 62–63
    indirect registers, used in 65, 68
    scaling operands 70
    specifying 60
Aliases 87, 369
ALIGN directive 3
Align types 45
    See also individual entries
.ALPHA directive 47
AND instruction 27, 99, 100
Angle brackets (< >)
    default parameters 230
    epilogues 202
    FOR loops 242

FORC loops 244

Angle brackets (< >) *(continued)*
macro text delimiters 234
prologues 202
records 131
structures and unions 121
Anonymous label (@@) 170
API (Application Programming Interface) 257
Architecture, segmented 2, 5
Architecture, unsegmented 5
Arguments
errors 196
macro 252
mixed-language programs, passing in 314
qualifiedtypes, with 16
stack, on 182
Arrays
accessing elements in 105
declaring 105
defined 105
defining 15
DUP, declaring with 106, 124
instructions for processing 110
length of 108
multiple-line declarations for 105
number of bytes in 108
referencing 108, 316
size of elements 108
with DUP operator *See* DUP operator
with SIZEOF operator *See* SIZEOF operator
with TYPE operator *See* TYPE operator
ASCIIZ 267
Assembly
actions during 23
conditional *See* Conditional assembly
INCLUDE files 212
language
book list xviii
mixed-language programs 312
listing files *See* Listing files
two-pass 358
Assembly pointers *See* Conditional assembly
Assembly-time variables 233
ASSUME directive
.MODEL, generated with 37
code segments, changing 357
enhancements 344
general-purpose registers 77
segment registers, setting 49–55, 58–59, 357
AT address combine type 46
/AT command-line option, ML 36
At sign (@) 10

B

Backslash character (\) 22
Backus-Naur Form *See* BNF grammar
Base Pointer (BP) register 73
Basic calling conventions 308–310
Basic/MASM programs 328–332
Binary Coded Decimals
    calculating with 156–160
    defining 156
    instructions for 156–160
    packed 158
    unpacked 159–160
Bits
    mask 99–102
    rotating 100
    shifting 100
BNF grammar 16, 379–380
BOUND instruction 108, 204
BP (Base Pointer) register 73
Brackets ([ ]) 107
.BREAK directive 173, 176
BSF instruction 100
BSR instruction 100
BYTE
    align type 45
    directive 86

C

C calling convention 309
C++/MASM programs 322–323
C/MASM programs 315–321
CALL instruction 180
Calling conventions 309
    Basic 308–310
    directives, specifying 37
    FORTRAN 308–310
    (list) 308
    mixed-language programming 308–309
    Pascal 310
    STDCALL 311
    SYSCALL 308–311
CARRY? flag as operand 178
Case sensitivity
    enforcing 348
    macro functions, predefined 245
    MASM statements 22
    radix specifiers 11
    reserved words 9, 407
    specifying
        command-line options, in 25
        language type 348
OPTION directive 25

Case sensitivity *(continued)*
symbols, predefined 10
CASEMAP
ALL argument, OPTION directive 25
NONE argument, OPTION directive 25
NOTPUBLIC argument, OPTION directive 25
CATSTR directive 245–247
CATSTR, compared with TEXTEQU directive *See* TEXTEQU directive
@CatStr predefined string function 245–247
CBW instruction 90
CDQ instruction 90
CLC instruction 104
Cleaning the stack 185
CLI instruction 5, 209
Client program 257, 266
CMC instruction 104
CMP instruction 166
CMPS instruction 110–114, 353
CMPSB instruction 114
.CODE directive 33, 40–42
Code segment *See* Segments, code
Code, near or far 57
@CodeSize predefined symbol 40
CodeView for Windows 264
Combine types
(list) 46
See also individual entries
.COM files
relocatable segment expression, lacking 62
starting address 56
tiny model, using 36, 46–47
COMM directive 16, 211, 217–218
Command-line driver, ML xvi
Command-line options *See* ML command-line options
COMMENT directive 22
Comments
extended lines, in 346
macros, in 227
source code 21–22
COMMON combine type 46
Communal variables 217
Compact model *See* Memory models, compact
Compatibility, MASM 5.1 *See* MASM 5.1 compatibility
Conditional assembly
assembly behavior, changing 23
conditions, testing for 28
directives 28
pointers 83, 187
Conditional-error directives (table) 29
Conditional jumps 164–170
Conditions, testing for conditional assembly *See* Conditional assembly
Constants
defined 11
expressions 12
immediate 61
integer 11–12
size 363
size of 12
symbolic 12
.CONST directive 33, 39–40
.CONTINUE directive 173, 176
Coprocessors
architecture 140–144
control registers 156
data format in registers 140
defined 135
described 3, 139
instructions
arithmetic 148–150
data transfer 146
described 146
(list) 414
overview 141
program control 151–155
memory access 145
operand formats
classical stack 141
memory 142
overview 141
register 143
register-pop 144
specifying 37, 140
status word register 156
steps for using 145
/Cp command-line option, ML 10, 245
@Cpu predefined symbol 254
Curly braces ({})
records 131
structures and unions 121
Current address operator ($) 368
@CurSeg predefined symbol 39, 219
CWD instruction 90
CWDE instruction 90
/Cx command-line option, ML 158

D

DAA instruction 162
DAS instruction 162
.DATA directive 33, 39–40
.DATA? directive 33, 39–40
@data predefined symbol 39
Data segment *See* Segments, data
@DataSize predefined symbol 39, 83

Data types
arrays *See* Arrays
attributes for 15
Binary Coded Decimals 159
defined 14
defining 87
directives 14
floating-point 136
initializers, as 14
integers, allocating memory for 85–86
new features, MASM 6.1 344
qualifiedtypes 15, 214
real 14, 136
signed 14, 86
strings *See* Strings
structures 117
unions 117
user-defined 15
Data, near or far 57, 58
Data-sharing methods 211
Data-sharing methods, multiple-module programs *See* Multiple-module programs
Date, system 11
DB directive 86
DD directive 86
DEC instruction 92–94
DF directive 86
DGROUP group name
.MODEL, defined by 34, 39, 51
DS registers, initializing to 56
MS-DOS programs, for 41–42
near data, accessing 57–58
segment 35–37, 51–52, 57
Direct memory operands
loading offset of 82
overview 60–64
Directives
.286P 38
.287 38
.386 *See* .386 directive
.386P 38
.387 38
.486P 38
.8087 38
.ALPHA 47
ALIGN 3
.BREAK 173, 176
BYTE 86
CATSTR 245–247
.CODE 33, 40–42
COMM 16, 211, 217–218
COMMENT 22
Conditional assembly 28
Conditional error 29, 358

Directives *(continued)*
.CONST 33, 39–40
.CONTINUE 173, 176
.DATA 33, 39–40
.DATA? 33, 39–40
Data declarations, for 87
Data types, for 14
Data-sharing *See* EXTERN directive
DB 86
DD 86, 136
Decision 171
DF 86
.DOSSEG 47
DQ 86, 136
DT 86, 136
DW 86
DWORD 86
ECHO 236
.ELSE 171
ELSE 28
.ELSEIF 171
ELSEIF 28
ELSEIF1 29, 358
ELSEIF2 29, 358
END 33, 56
.ENDIF 171
ENDIF 28
ENDM 227–239
ENDP 180–181, 206
ENDS 44
.ENDW 173
EQU 12, 369
.ERR 30
.ERR1 30, 358
.ERR2 30, 358
.ERRB 30, 231
.ERRDEF 30
.ERRDIF 30
.ERRE 30
.ERRIDN 29
.ERRNB 29, 231
.ERRNDEF 29
.ERRNZ 29
EVEN 3
.EXIT 33, 41–43
EXITM 248
EXTERN *See* EXTERN directive
EXTERNDEF *See* EXTERNDEF directive
FARDATA 33, 39–40
.FARDATA 39–40
.FARDATA? 33, 39–40
Floating-point 136
FOR 242–243, 249
FORC 244

Directives *(continued)*
FWORD 86
GROUP 51–52
.IF 171
IF 28–29
IF1 29, 358
IF2 29, 358
IFB 29, 231
IFDEF 29, 359
IFDIF 29
IFE 29
IFIDN 29
IFNB 29, 231
IFNDEF 29, 359
INCLUDE 212
INCLUDELIB 222
INSTR 245–246
INVOKE *See* INVOKE directive
LABEL 16
LOCAL 188–191, 232
loop-generating 173
.MODEL *See* .MODEL directive
.MSFLOAT 361
Naming conventions 37
.NO87 38, 349
obsolete 361
OPTION *See* OPTION directive
ORG 56
POPCONTEXT 255, 349
PROC 180–184, 193, 206, 312
PUBLIC 185, 211, 220
PUSHCONTEXT 255, 349
QWORD 86
.RADIX 11
REAL4 136–137
REAL8 136–137
REAL10 136–137
RECORD 130–131
Renamed since MASM 5.1 350
.REPEAT 173–177
REPEAT 240
SBYTE 86
SDWORD 86
SEGMENT 44–47
Segment order, controlling 47
.SEQ 47
SIZESTR 245–246
STACK *See* STACK directive
.STARTUP *See* .STARTUP directive
STARTUP *See* .STARTUP directive
STRUCT 118–129
SUBSTR 245–246
SWORD 86
TBYTE 86, 159

Directives *(continued)*
TEXTEQU *See* TEXTEQU directive
UNION 118–119, 122, 125–129
.UNTIL 173
.UNTILCXZ 173
.WHILE 173–177
WHILE 241
WORD 86
Directives: 36–38, 46
Displacement 66
Distance attributes 15
DIV instruction 97–98
Division 97, 102
DLLs
client program 257, 266
data segment 265–269
defined 257, 266
example 267–268
extension name 266
heap 261–262, 265–267
IMPLIB utility 258
initialization 261–262, 268–269
loading 258–260
programming requirements 260–261, 267
prologue and epilogue 264–267
stacks in 46, 264–267
summary 266
termination 262–264, 270
Document conventions vii
DOS *See* MS-DOS
.DOSSEG directive 47
Dot (.) operator *See* Structure-member operator
DOTNAME argument, OPTION directive 25
Double colon (::) 197, 215
Double quotation marks (”) 109
Double semicolon (;;) 227
Doublewords 86
DQ directive 86
DT directive 86
DUP operator
arrays, with 106, 124
record variables, with 131
structures and unions, with 121
DW directive 86
DWORD
align type 45
directive 86
Dynamic-link libraries *See* DLLs

E

ECHO directive 236
.ELSE directive 171
ELSE directive 28

.ELSEIF directive 171
ELSEIF directive 28
ELSEIF1 directive 358
ELSEIF2 directive 29, 358
EMULATOR argument, OPTION directive 27, 157
Emulator libraries 155–156
END directive 33, 56
.ENDIF directive 171
ENDIF directive 28
ENDM directive 227–239
ENDP directive 180–181, 206
ENDS directive 44
.ENDW directive 173
ENTER instruction 183
Environment
    target 4
    variables
        INCLUDE 213
        LIB 222
        returning values of 10
/EP command-line option, ML 342
EPILOGUE argument, OPTION directive 26, 201–203
Epilogue code
    defined 198
    macros 201–202, 264–265
    PROC statement, specifying arguments in 185
    procedures, with 26
    RET instruction 357
    standard 199
    user-defined 201
EQ operator 365
EQU directive 12, 369
Equal directive (=) 12
Equates, predefined *See* Predefined symbols
.ERR directive 29
.ERR1 directive 30, 358
.ERR2 directive 30, 358
.ERRB directive 29, 231
.ERRDEF directive 29
.ERRDIF directive 29
.ERRE directive 29
.ERRIDN directive 29
.ERRNB directive 29, 231
.ERRNDEF directive 29
.ERRNZ directive 29
Error detection 196
ERROR operand 49–50
Errors, argument passing 196
ESC instruction 360
EVEN directive 3
Executable (.EXE) files, controlling size of 223
Exit codes, Windows operating system 263
.EXIT directive 33, 41–43
EXITM directive 248

Expansion operator (%) 235–236, 248, 357
Explicit loading 258
Exponent bias 139
EXPORT operand 185
EXPORTS statement 261, 270
EXPR16 argument, OPTION directive 13, 26, 361, 373
EXPR32 argument, OPTION directive 13, 26, 373
Expression operators 178
Expressions
    assembly-time evaluation 23
    constant 12
    loop conditions, evaluating 179
    OPTION M510 behavior 364, 373
    order of evaluation 14
    size 366, 373
    word size 13, 26
Extension, filename 266
EXTERN directive
    data-sharing 211
    executable file size, limiting 223
    module-specific 220
    overview 16
    positioning 218
    procedure prototypes, declaring 193
External declarations 216–218
External variables 217, 369
EXTERNDEF directive
    data-sharing 211
    overview 16
    positioning 218
    procedure prototypes, declaring 193
    symbols, declaring 214–215

F

Far addresses, invoking 57, 74, 80–81, 197
Far code 57
Far data 58–60
.FARDATA directive 33, 39–40
.FARDATA? directive 39–40
FAR operator 169, 185
Far pointer 74, 80–81
FARSTACK operand
    example 35
    grouping 34
    in Windows-based programs 266
    MS-DOS program, initializing 43
    special cases, setting for 37
Farwords 86
FCOM instruction 153
Fields, statements in 21–22
Files
    .COM
    relocatable segment expression, lacking 62

Files *(continued)*
.COM *(continued)*
starting address 56
tiny model, using 36, 46–47
executable 24
include 212–213, 348
line numbers 11
naming 11
Flags
CARRY? 178
operands, as 178
OVERFLOW? 178
PARITY? 178
SIGN? 178
stack, saving on 73
ZERO? 178
Flags register *See* Registers, flags
Flat model *See* Memory models, flat
FLAT operand 46, 49–50
FLD1 instruction 147
FLDZ instruction 147
Floating-point
calculations 3
constants
decimal form 137
encoded hexadecimal format 137
syntax for defining 136
emulation 157–158
IEEE format 139
instructions
arithmetic 148–149
controlling 26
data transfer 147
not emulated (list) 158
program control 152–153, 156
operations 146
values
double precision 139
single precision 139
variables
IEEE format 138
Microsoft binary format 138
.MSFLOAT format 138
ranges 136
FOR directive 242–243, 249
FORC directive 244
FORCEFRAME operand 200–201
FORTRAN calling convention 308–310
FORTRAN/MASM programs 323–326
/Fpi command-line option, ML 26, 157
Frame 62
FS register 17
FTST instruction 153

Full segment definitions
described 32
segment registers, initializing 54–56
using 44–51
Full segment defintions *See* .STARTUP directive
FWORD directive 86
FXCH instruction 144

G

Global variables 211
GROUP directive 51–52
Groups
defined 51
DGROUP 51
SEG operator, returned by 62
GS register 17

H

H2INC 318
Heap space 261–262, 265–267
HEAPSIZE statement 261, 271
Help, online *See* Microsoft Advisor
HIGH operator 356
HIGHWORD operator 346
Huge model *See* Memory models, huge

I

/I command-line option, ML 213
Identifiers
ABS, using 220
naming restrictions 9, 346, 353, 357, 368
OPTION DOTNAME 373
OPTION NOKEYWORD 376
IDIV instruction 97–98
IEEE format 139
.IF directive 171
IF directive 28–29
IF1 directive 29, 358
IF2 directive 29, 358
IFB directive 29, 231
IFDEF directive 29, 359
IFDIF directive 29
IFE directive 29
IFIDN directive 29
IFNB directive 29, 231
IFNDEF directive 29, 359
Immediate operands 60–62
IMPLIB utility 258
Implicit loading 258
Import libraries 258

IMPORTS statement 266
IMUL instruction 95–96
IN instruction 5
INC instruction 92–94
INCLUDE directive 212
INCLUDE environment variable 213
Include files
  assembling 213
  nested 213
  overview 212, 348
INCLUDELIB directive 222
Index operator ([ ]) 63
Indirect memory operands 60, 64–70
Indirect procedure calls *See* INVOKE directive
Initializers
  allocating 87
  directives for 15
  multiple-line 346
Instance 261, 266
INSTR directive 245–246
@InStr predefined string function 245–246
Instruction Pointer (IP) register 20, 57, 161
Instructions
  ADC 92–94
  ADD 92–94
  AND 26, 99–100
  arithmetic 378
  bit-test 354
  BOUND 108, 204
  BSF 100
  BSR 100
  CALL 180
  CBW 90
  CDQ 90
  CLC 104
  CLI 5, 209
  CMC 104
  CMP 166
  CMPS 110–114, 353
  CMPSB 114
  conditional-jump 165–167
  coprocessor 377
  CWD 90
  CWDE 90
  DAA 162
  DAS 162
  DEC 92–94
  default segments, requiring 49
  DIV 97–98
  encodings, changes to 377–378
  ENTER 183
  ESC 360
  FCOM 153
  FLD1 147
Instructions *(continued)*
  FLDZ 147
  floating-point *See* Floating-point instructions
  FTST 153
  FXCH 144
  IDIV 97–98
  IMUL 95–96
  IN 5
  INC 92–94
  INT 204–205
  INTO 207
  JCXZ 170–173
  JECXZ 170–173
  JMP 49, 162
  JO 165
  jump 165–167, 170, 173
  LAHF 73
  LDS 81
  LEA 82, 104
  LEAVE 183
  LES 81
  (list) 412
  LOCK 353, 363
  LODS 110–115, 353
  logical 99–102
  LOOP 172
  LOOPE 172
  LOOPNE 172
  LOOPNZ 172
  LOOPZ 172
  MOV 49, 82, 89
  MOVS 110–113, 353
  MOVSX 92
  MOVZX 92
  MUL 95–96
  NOP 377
  NOT 99–100
  obsolete 360
  operands for 60
  OR 26, 99–100, 168
  OUT 5
  POP 49, 71
  POPA 74
  POPAD 74
  POPF 73
  POPFD 73
  privileged 2, 38
  PUSH 49, 71
  PUSHA 74
  PUSHAD 74
  PUSHF 73
  PUSHFD 73
  RCL 101–104
  RCR 101–104

Instructions *(continued)*
REP 110–112, 363
REPE 110–112, 363
REPNE 110–112, 353, 363
REPNZ 110–112, 353, 363
REPZ 110–112, 363
RET 378
RETF 181, 378
RETN 181, 378
ROL 101–104
ROR 101–104
SAL 101–104
SAR 101–104
SBB 92–94
SCAS 110–115, 353
SHL 101–104
SHR 101–104
STC 104
STI 5, 209
STOS 110–113, 353
SUB 92–94
TEST 167–168
timing xvii, 399–400
XCHG 90
XLAT 116
XLATB 116
XOR 26, 99–100
Integers
adding 92–94
allocating memory for 85–86
Binary Coded Decimal (BCD) 159
bit operations on 99
constants, defining 11–12
dividing 97–98
exchanging 90
hexadecimal 12
initializing 87
memory format 86
moving 89
multiplying 95–96
operations with 88
popping off stack 71
pushing onto stack 71
radix specifiers for 11
sign-extending 90
signed 86
size of 86
stack 71
subtracting 92–94
translating 116
types, defining 14, 86
value range 86
@Interface predefined symbol 37
Interrupt vector 205

Interrupt-enable flag 205
Interrupts
  CLI instruction 209
  handlers 206–207
  INT instruction 204–205
  MS-DOS 204, 285
  operation 206
  overview 204
  redefining 207
  STI instruction 209
  vector table 205
INTO instruction 207
INVOKE directive
  actions 194
  ADDR, invoking 197
  arguments, widening 196
  error detection 196
  far addresses, invoking 197
  generated code, checking 198
  indirect procedure calls 198
  mixed-language programs 312–313
  procedures, calling 193–197, 216
  type conversions 194–195

J

JCXZ instruction 170–173
JECXZ instruction 170–173
JMP instruction 49, 162
JO instruction 165
Jumps
  anonymous 170
  automatic 169
  conditional
    bit status 167
    comparisons 166
    extending 26, 169
    flag status 165–166
    instructions (list) 165–167
    overview 164
    zero value 168
  directives for 171
  extension, automatic 26, 169
  instructions 165–167
  optimization, automatic 162
  overview 161
  unconditional
    indirect operands 163
    jump tables 163
    overview 162

L

LABEL directive 16

Labels
anonymous 170
code
length 346
OPTION M510 behavior 363
OPTION NOSCOPED 375
procedures, in 357
referencing 352
size 346
visibility 354
LAHF instruction 73
LANGUAGE
BASIC argument, OPTION directive 26
C argument, OPTION directive 26
FORTRAN argument, OPTION directive 26
PASCAL argument, OPTION directive 26
STDCALL argument, OPTION directive 26
SYSCALL argument, OPTION directive 26
LANGUAGE argument, OPTION directive 193
Language attributes
.MODEL directive, with 34, 37
OPATTR operator 253
OPTION directive, with 26
Large model *See* Memory models, large
LDS instruction 81
LEA instruction 82, 104
LEAVE instruction 183
Length of strings *See* LENGTHOF operator
LENGTH operator 356–357, 364
LENGTHOF operator
number of items, returning 110, 124, 132, 346
structures, defining 108
unions, with 125
LES instruction 81
Libraries
C run-time 271
emulator 155–156
overview 221
source files, specifying in 222
LIBRARY statement 270
Line-continuation character (\) 121
LINK, command-line options *See* individual entries
Linkage specification 322–323
Linking
actions during 24, 45
segment order in 48
Listing files
code generated 399
command-line options 397–399
error messages 400
examples 401
generating 397
PWB options 397–399
reading 399, 405

Listing files *(continued)*
symbols used in (list) 400
tables in 405–406
Literal-character operator (!) 235
LJMP argument, OPTION directive 27
LOADDS operand 200–201
Loading local address variables *See* Local variables
Loading, actions during 24
Local addresses, loading *See* Local variables
LOCAL directive 188–191, 232
Local variables
creating 188
loading addresses of 82
procedures, in 188
LOCK instruction 353, 363
LODS instruction 110–115, 353
Logical AND 178
Logical instruction 99–100
Logical line 22
Lookup tables 241
LOOP instruction 172
LOOPE instruction 172
LOOPNE instruction 172
LOOPNZ instruction 172
Loops
conditions
expression evaluation 179
precedence 179
PTR operator in 178
relational operators for (list) 178
signed operands 178
writing 178
controlling 176
directives
.REPEAT 173–177
.WHILE 173–177
instructions (list) 172
macros
FOR 242–243, 249
FORC 244
REPEAT 240
WHILE 241
LOOPZ instruction 172
LOW operator 356
LOWWORD operator 346, 366
LROFFSET operator 344

M

M510 argument, OPTION directive
compatibility with MASM 5.1 26, 353–370
expression word size, setting 13
structures, with 119

Macros
arguments
commas 352, 372
quotation marks 353
testing 29, 252
variable 242, 249
calling 227
checking argument types with 253
comments (;;) 227
expansion 23
functions
defined 248
epilogues 201
EXITM 248
prologues 201
returning values 248
local symbols in 232
loops
FOR 242–243, 249
FORC 244
REPEAT 240
WHILE 242–243
MASM 5.1 behavior 25, 356, 372
nested 251
new features 351
operators
behavior in macro functions 251
expansion (%) 235–236, 248, 357
(list) 234
literal-character (!) 235
substitution (&) 238, 352, 372
OPTION OLDMACROS 372
parameters
default values 230
procedure parameters, compared to 234
required 229
substitution 238
passing arguments to 228, 235
predefined string functions 11
procedures
defined 226
functions, compared to 228
recursive 255
redefining 251
text
defined 226
forward referencing 356
numeric equates, compared to 234
OPTION M510 behavior 370
syntax 226
VARARG keyword 242, 249, 351
writing 227
Mask
defined 99

Mask *(continued)*
logic instructions, with 102
record operators, with 133
MASK operator 133
MASM 5.1 compatibility
address fixups 26
macro behavior 25, 356, 372
OPTION directive, specifying 25
overview xvi
structures 25
updating code 353–360
MASM utility xvi, 342
Math coprocessor *See* Coprocessors
Medium model *See* Memory models, medium
Memory
access 64
allocation 24
virtual 5
MEMORY combine type 46
Memory models
attributes 35
compact 36
described 34
determining 10
far code segments 40
far data segments 40
flat 36, 58, 336
huge 36
large 36
medium 36
model-independent code 83
near code segments 40
small 36
specifying in PROC statement 185
tiny 36, 46–47
Memory-resident programs *See* TSRs
Microsoft Advisor xiii, 342
Minus operator (–) 64
Mixed-language programming
argument passing 314
assembly procedures 312
Basic/MASM programs 328–332
C prototypes, converting with H2INC 318
C++/MASM programs 322–323
C/MASM programs 315–321
calling conventions
Basic 308–310
FORTRAN 308–310
(list) 308
Pascal 310
STDCALL 311
SYSCALL 308–311
column-major order 315

Mixed-language programming *(continued)*
compatible data types
Basic (list) 328
C (list) 315
FORTRAN (list) 323
external data 314
FORTRAN/MASM programs 323–326
initialization code 313, 321
INVOKE, using 312–313
naming conventions 308–309
overview 307
register preservation 314
row-major order 315
ML command-line options
/AT 36
/Cp 10, 245
described xvi
/EP 342
/Fpi 26, 157
/I 213
listing options (list) 397
overview xvi
/X 213
/Zm 62, 119
/Zp 119
Mode, real, protected *See* Real mode; Protected mode
.MODEL directive
attributes 34–35
DGROUP 51
language types, specifying 26, 308
memory model, defining 35–36
mode default 46
overview 34
positioning 46
simplified segment directives 33
@Model predefined symbol 35, 83
Module-definition file
described 270
statements
EXPORTS 261, 270
HEAPSIZE 261, 271
IMPORTS 266
LIBRARY 270
STUB 266
Module-specific EXTERN directive *See* EXTERN directive
MOV instruction 49, 82, 89
MOVS instruction 110–113, 353
MOVSX instruction 92
MOVZX instruction 92
MS-DOS interrupts 204, 285
MS-DOS operating system 2–6
MUL instruction 95–96
Multiple-module programs
alternatives to include files 219

Multiple-module programs *(continued)*
COMM, using 217
data-sharing methods 211
declaring symbols public and external 214
EXTERN with library routines 223
external declarations, positioning 218
EXTERNDEF, using 214
include files 212–213
libraries 221–222
modules 212
PROTO, using 216
PUBLIC and EXTERN, using 220
sharing symbols with include files 212
Multiplex interrupt 291, 304
Multiplication
instructions 95
shift operations 102

N

Naming conventions
directives 37
(list) 308
mixed-language programming 308–309
Naming restrictions 9
Naming restrictions, identifers *See* Identifiers
NE operator 365
Near address 57, 80
NEAR operator 169, 185
NEARSTACK operand
ASSUME statement 54
default stack type 37, 42
described 35
New features, MASM 6.1 xiv–xv, 342–351
NMAKE 270
.NO87 directive 38, 349
NODOTNAME argument, OPTION directive 25
NOEMULATOR argument, OPTION directive 27
NOKEYWORD argument, OPTION directive 9, 27, 353, 376
NOLJMP argument, OPTION directive 27, 170
NOM510 argument, OPTION directive 25
NONUNIQUE operand 118, 126
NOOLDMACROS argument, OPTION directive 26
NOOLDSTRUCTS argument, OPTION directive 26
NOP instruction 377
NOREADONLY argument, OPTION directive 27
NOSCOPED argument, OPTION directive 26, 362, 375
NOSIGNEXTEND argument, OPTION directive 27, 378
NOT instruction 99–100
NOTHING operand 49–50
Number of items with LENGTHOF operator *See* LENGTHOF operator
Numeric equates, compared to text macros 234

O

OFFSET
- FLAT argument, OPTION directive 27
- GROUP argument, OPTION directive 27
- SEGMENT argument, OPTION directive 27, 62

OFFSET operator 61, 82, 356, 374
Offsets
- accessing data with 74
- addresses 7
- described 5–7
- determining 23–24, 360, 374
- fixups for 26

OLDMACROS argument, OPTION directive 25, 239, 361, 372
OLDSTRUCTS argument, OPTION directive
- MASM 5.1 compatibility 25, 361, 370–372
- structures, with 119, 126

Online help *See* Microsoft Advisor
OPATTR operator 252–253
Operands
- ABS 220
- direct memory 60–64
- EXPORT 185
- FAR 15
- FARSTACK *See* FARSTACK operand
- FLAT 46, 49–50
- FORCEFRAME 244
- immediate 60–62
- indirect memory 60, 64–70
- NEAR 15
- PRIVATE READONLY 44–45
- registers 61
- size 66, 355
- USE16 44–46
- USE32 44–46

Operating systems
- (list) 4
- .MODEL, specifying with 34
- multitasking 6
- types *See* MS-DOS, Windows operating systems

Operators
- ADDR 197
- current address ($) 368
- dot (.) 126, 352, 370
- EQ 365
- expansion (%) 235–236, 248, 357
- expressions, in 12–13
- FAR 169, 185
- HIGH 356
- HIGHWORD 346
- index ([ ]) 63
- instructions, compared to 13
- LENGTH 356–357, 364
- LENGTHOF 346

Operators *(continued)*
- LOW 356
- LOWWORD 346, 366
- LROFFSET 344
- macro 251
- MASK 133
- minus (–) 64
- NE 365
- NEAR 169, 185
- OFFSET 61, 82, *See* OFFSET operator
- OPATTR 252–254
- plus (+) 63, 66
- precedence 14
- PTR *See* PTR operator
- PTR, example *See* PTR operator
- relational 357, 365
- relational (list) 178
- SEG 50, 62, 363
- segment-override (:) 59, 64
- SHORT 169
- SIZE 364–365
- size *See* PTR operator
- SIZEOF 86, 346
- structure-member (.) 64–67, 126, 352, 370
- substitution (&) 238
- .TYPE 252, 360
- TYPE 86
- WIDTH 133

OPTION directive
- CASEMAP 25
- described 23
- DOTNAME 25, 361, 373
- emulation mode 157
- EMULATOR 26, 157
- EPILOGUE 26, 201–203
- EXPR16 OPTION directive 13, 26, 361, 373
- EXPR32 OPTION directive 13, 26, 373
- LANGUAGE 26, 193
- language types, specifying 308
- list of arguments for 25
- LJMP 26
- M510 *See* M510 argument, OPTION directive
- NODOTNAME 25
- NOEMULATOR 26
- NOKEYWORD *See* NOKEYWORD argument, OPTION directive
- NOLJMP 27, 170
- NOM510 25
- NOOLDMACROS 26
- NOOLDSTRUCTS 26
- NOREADONLY 27
- NOSCOPED 26, 362, 375
- NOSIGNEXTEND 27, 378

OPTION directive *(continued)*
OFFSET 26, 62, 362, 374–375
OLDMACROS 25, 237
OLDSTRUCTS *See* OLDSTRUCTS argument, OPTION directive
PROC 185, 375
procedure use 26
PROLOGUE 26, 201–203
READONLY 26
SCOPED 25
SETIF2 25, 29–30
using 25, 361
OR instruction 27, 99–100, 168
ORG directive 56
OUT instruction 5
OVERFLOW? flag as operand 178

P

PAGE align type 45
PARA align type 45
Parentheses [( )] 106
PARITY? flag as operand 178
Pascal convention 310
Physical line 22
Plus operator (+) 66, 352, 370
Pointer variables 74–78
Pointers
accessing data with 74
arguments, as 80
copying 79
far 74, 80–81
initializing 78
location 74
operations 78
TYPEDEF, defined with 15, 75–78
types, to 15
Pointers and conditional Assembly *See* Conditional assembly
Pointers defined by TYPEDEF *See* TYPEDEF directive
POP instruction 49, 71
POPA instruction 74
POPAD instruction 74
POPCONTEXT directive 255, 349
POPF instruction 73
POPFD instruction 73
Positioning
EXTERN directive *See* EXTERN directive
EXTERNDEF directive *See* EXTERNDEF directive
Precedence operators 14
Predefined equates *See* Predefined symbols
Predefined functions for macros 11
Predefined string functions
@CatStr 245–247
@InStr 245–246

Predefined string functions *(continued)*
@SizeStr 245–246
@SubStr 245–246
Predefined symbols 39, 83
@Codesize 40
@Cpu 254
@CurSeg 39, 219
@Data 39
@DataSize 39, 83
@Interface 37
(list) 10, 409
@Model 35, 83
@stack 37
@Wordsize 39
case sensitivity 9–10
new to MASM 6.1 (list) 343
PRIVATE operand 185
Privilege levels 5
Problems, reporting xx
PROC
EXPORT argument, OPTION directive 25
PRIVATE argument, OPTION directive 25, 362
PUBLIC argument, OPTION directive 25, 185
PROC directive 180–184, 193, 206, 312
PROC statements with visibility *See also* Visibility
PROC with RET instruction *See* RET instruction
Procedure prototypes
declaring *See* EXTERNDEF directive
defined with *See* PROTO directive
defined with PROTO directive *See* PROTO directive
writing *See* PROTO directive
Procedures
arguments
far pointers 197
near addresses 197
passing 182
pointers 80
type conversions 195, 196
CALL instruction 180
calling *See* INVOKE directive
calls
indirect 198
optimizing 181
defining 180
epilogues 26
EXTERNDEF directive 214–215
See also EXTERNDEF directive,
include files 214
INVOKE directive 193–197, 216
libraries 221
local variables 188–192
See also Local variables
Macro *See* Macros, procedures
new features 347

Procedures *(continued)*
OPTION PROC 375
overview 180
parameters
declaring 184–186
variable numbers of 186–188, 194
PROC attributes, specifying 185
prologues 26
PROTO directive 193, 214, 216
See also PROTO directive
prototypes, writing 193
RET instruction 180
RETF instruction 181
RETN instruction 181
syntax description 184
VARARG keyword 186–188, 194
visibility 25, 375
Processors
See also Real mode; Protected mode
8086-based 2–3
.MODEL directive 37
modes, determining 10
target 2
timing xvii, 399–400
Product assistance xx
Program Segment Prefix (PSP) 56
Programming, MASM 6.1 practices 352
Programs
exiting 41
mixed-language 307
starting 41
PROLOGUE argument, OPTION directive 25, 201–203
Prologue code
arguments, specifying 185
code labels in 357
defined 198
macros for 201–203, 264–265
standard 199
user-defined 26, 201
Protected mode
described 2–7, 335
flat model 335
read-only segments 45
PROTO directive
include files 211, 214–216
procedure prototypes, defined with 193
procedure prototypes, writing 312
Prototypes
procedure
directives for 193
overview 193
qualifiedtypes, defined with 15
PTR operator
example 92

PTR operator *(continued)*
OPTION M510 behavior 365
pointer to type, as 15
signed number, specifying 178
size 66, 88
TYPEDEF, used with 75
PUBLIC combine type 45
PUBLIC directive 185, 211, 220
PUSH instruction 49, 71
PUSHA instruction 74
PUSHAD instruction 74
PUSHCONTEXT directive 255, 349
PUSHF instruction 73
PUSHFD instruction 73

Q

Quadwords 86
Qualifiedtypes
BNF grammar 16
defined 15
pointers, defining 75–76
prototypes, as 15
rules for use 15–16
Question mark initializer (?)
array elements 109
described 368
variables 87
Quotation marks (' or ") 109
QWORD directive 86

R

.RADIX directive 11
Radix specifiers
(list) 11
OPTION M510 behavior 367
RCL instruction 101–104
RCR instruction 101–104
Read-only code 27
READONLY argument, OPTION directive 26
READONLY operand 44–45
Real mode 2, 4, 7
Real numbers *See* Floating-point
REAL4 directive 136–137
REAL8 directive 136–137
REAL10 directive 136–137
RECORD directive 130–131
Records
defined 129
field ranges 354
LENGTH operator 357
operators 133–134
RECORD syntax 130–131

Records with SIZEOF operator *See* SIZEOF operator
Records with TYPE operator *See* TYPE operator
Recursive macros 255
Register operands 61
Registers
    16-bit 16–17, 67
    32-bit 335
    base 65–70
    coprocessor 140
    copying pairs of 82
    division (table) 98
    Eflags 20
    extended 17
    flags 20
    FS 17
    general purpose 19
    GS 17
    index 65–69
    indirect addressing 65
    indirect operands 67–68
    initializing 44
    Instruction Pointer (IP) 20, 57, 161
    (list) 409
    loading addresses into 80
    mixed 16-bit, 32-bit 70
    pointers as 77
    scaling 67–69
    segment *See* Segment registers
    Stack Pointer (SP) 19
    Stack Segment (SS) 73
    stacks, saving on 74
    types, defined with ASSUME 77
Relational operators (list) 178
Relocatable
    addresses 57
    expressions 62, 65
REP instruction 110–112, 363
REPE instruction 110–112, 363
Repeat blocks 239
.REPEAT directive 173
REPEAT directive 240
REPNE instruction 110–112, 353, 363
REPNZ instruction 110–112, 353, 363
Reporting problems xx
REPZ instruction 110–112
Reserved words
    described 8, 26
    (list) 407
    OPTION M510 behavior 362
    OPTION NOKEYWORD 376
RET instruction
    epilogue code, generating 200, 378
    instruction encodings, changes to 357
    PROC, with 180

RETF instruction 181, 378
RETN instruction 181, 378
ROL instruction 101–104
ROM-BIOS interrupts *See* Interrupts
ROR instruction 101–104
Rotate instructions 101
Routines, interrupt 206

S

SAL instruction 101–104
SAR instruction 101–104
SBB instruction 92–94
SBYTE directive 86
Scaling factor 107
Scaling index registers 67–69
SCAS instruction 110–112, 115, 353
Scope within visibility *See also* Visibility
SCOPED argument, OPTION directive 26
SDWORD directive 86
SEG operator 49, 62, 363
SEGMENT
    FLAT argument, OPTION directive 27
    USE16 argument, OPTION directive 27
    USE32 argument, OPTION directive 27
Segment arithmetic 7
SEGMENT directive 44–47
Segment mode, setting *See* .386 directive; .486 directive
Segment registers
    32-bit 335
    assigning 59, 62
    ASSUME directive 49–55, 58–59, 357
    changing 57
    default 60, 64
    described 18
    FS 18
    GS 18
    initializing 43, 54–57
    MS-DOS, under 24, 43
    near code 57
    restoring 59
    segment-override operator (:) 50, 59–60, 64
Segment registers
    initializing *See* STARTUP directive
    setting *See* STACK directive
Segment selectors 5
Segment-override operator (:) 50, 59–60, 64
Segmented architecture 2, 5
Segments
    32-bit 36, 335
    accessing data 74
    aligning 44–45
    class types 44, 47–48

Segments *(continued)*
code
creating 40
far 40
memory model support for 36
near 40
combining 40, 44–46
current 10
data
creating 39
default 49, 54–55, 59
far 40
memory model support for 36
near 39
defined 31
described 5–7
determining order of 47–48
determining position of 23–24
determining size of 44
fixups for 26
full segment definitions, defining 32
groups, defining 51
initializing 55
location of 6
naming 40
ordering with the linker 48
protection 6
READONLY 45
simplified segment directives 37–42
size, determining 10
types 44
USE16 44
USE32 44
values 55
word size, setting 46
Selector 335
Semicolon (;), comments 21
.SEQ directive 47
SETIF2 argument, OPTION directive 25, 29–30
Shift instructions 100
SHL instruction 101–104
SHORT operator 169
SHR instruction 101–104
Sign-extending integers 90
SIGN? flag as operand 178
Signed data 14, 91
Signed numbers, specifying *See* PTR operator
Significand 139
Simplified segment directives
code segments 41
code, starting and ending 42
data segments 40
described 32
language convention 36

Simplified segment directives *(continued)*
memory model 35
.MODEL, defining with 34
operating system 35
processor 38
segment registers, initializing 54–56
stack 39
stack distance 37
using 33
Single quotation mark (') 109
Size attribute, segments
FLAT 46
USE16 46
USE32 46
Size mismatch 355
Size of strings *See* SIZEOF operator
SIZE operator 364, 365
@SizeStr predefined string function 245–246
SIZEOF operator
arrays, with 108
described 346
records, with 132
strings, with 110
structures, with 124
types 86
unions, with 125
SIZESTR directive 245–246
Small model *See* Memory models, small
Source code, statements in 21
SP (Stack Pointer) register 19, 71–73
SS (Stack Segment) register 73
STACK combine type 45
.STACK directive
described 33
segment registers, setting 56
Stack distance 37
Stack frame 73, 200, 264–265
Stack Pointer (SP) register 19
@stack predefined symbol 37
Stack Segment (SS) register 73
Stacks
cleaning 185
creating 38
described 71
distance 37
far 10
FARSTACK 35, 37
in DLLs 264–267
local variables on 188–191
near 10
NEARSTACK 33, 35–37
operations with 72–74
operators 71
passing arguments 182

Stacks *(continued)*
pointer 71–73
POP instructions 71
PUSH instructions 71
saving flags 73
saving registers 74
segment register 18
separate 46
trace 264
.STARTUP directive
described 33
initializing segments 54–56
program, starting 41–42
segment address 37
Statements
case sensitivity 22
syntax 21
Status flags, saving 73
STC instruction 104
STDCALL calling convention 311, 336
STI instruction 5, 209
STOS instruction 110–113, 353
Strings
declaring 109
defined 105
defining 15
initializing 109
instructions
processing, for 110
requirements (table) 112, 353
length of 110
multiple-line declarations for 109
overview 111
predefined functions for macros 11
See also Predefined string functions
size of 110
type of 110
STRUCT directive 118–129
Structure-member operator (.) 64–67, 126, 352, 370
Structures
alignment of fields 118–119
array initializers 122
arrays 124
compatibility with MASM 5.1 25, 118
current address operator ($) 368
default field values 122
defined 117
fields
accessing 64, 67, 371
initializing 118
naming 119, 352, 372
initializers, as 123
MASM 5.1 behavior 25, 355, 370
memory allocation 117

Structures *(continued)*
nested 128–129
new features 345
operators 124
OPTION M510 behavior 366
OPTION OLDSTRUCTS 370
redeclaration 124, 355
referencing fields in 126
steps for using 118
string initializers 122, 368
syntax
types 118
variables 121
Structures with LENGTHOF operator *See* LENGTHOF operator
Structures with SIZEOF operator *See* SIZEOF operator
Structures with TYPE operator *See* TYPE operator
STUB statement 266
SUB instruction 92–94
Substitution operator (&) 238, 372
SUBSTR directive 245–246
@SubStr predefined string function 245–246
SWORD directive 86
Symbol table, listing files 405
Symbols
declaring public and external 214, 220
external 369
naming 346, 368
predefined 9–11
Symbols, declaring by EXTERNDEF directive *See* EXTERNDEF directive
Syntax, MASM 6.1 statements 21
SYSCALL calling convention 308–311
System date 11
System time 11

T

Tables, lookup 241
Target environment 4
TBYTE directive 86, 159
Terminate-and-Stay-Resident programs *See* TSRs
TEST instruction 167–168
Testing for zero 168
Text delimiters *See* Angle brackets
Text macros *See* Macros, text
TEXTEQU directive
aliases 369
CATSTR, compared with 247
syntax 226
Time, system 11
Timing (cycle/second) xvii, 399–400
Tiny model *See* Memory models, tiny
Trap flag 205

TSRs
active
described 275
interrupt handlers in 275
MS-DOS functions, calling 285
MS-DOS functions, interrupting 286, 302
deinstalling 292, 305
described 273
errors, trapping 288–289
examples
ALARM.ASM 279–280, 284
SNAP.ASM 293–305
existing data, preserving 290, 303
hardware events, auditing 275–276, 299
interrupt handlers 275
monitoring
Critical Error flag 287
system status 277, 300
MS-DOS internal stacks (lists) 286
multiplex interrupt 290, 304
passive 274
Type conversions *See* INVOKE directive
Type of strings *See* TYPE operator
TYPE operator
and OPATTR 252–253
arrays, with 108
compatibility 360, 365
records, with 132
string, with 110
structures, with 124
types 86
unions, with 125
TYPEDEF directive
aliases, created by 87, 137
BNF, from 380
data types, defining 87
indirect operands, defining 163
pointers, defined by 15, 75–78
procedure declarations 193
procedure prototypes 193
qualifiedtypes 16
TYPEDEF, used with PTR operator *See* PTR operator
Types, data *See* Data types

U

Unconditional jumps 162
UNION directive 118–119, 122, 125–129
Unions
arrays as initializers 122
arrays of 124
defined 117
fields 119, 127–129
memory allocation 117
Unions *(continued)*
nested 128–129
operators 125
referencing fields in 126
steps for using 118
strings as initializers 122
types 118
variables 121, 127
Unpacked BCD numbers 160
Unsegmented architecture 5
Unsigned data 91
.UNTIL directive 173
.UNTILCXZ directive 173
USE16 operand 44–46
USE32 operand 44–46
USES in PROC statement 184
Utilities
IMPLIB 258
MASM 342
ML xvi

V

VARARG keyword
macros, used in 242, 249, 351
procedures, used with 186–188, 194
Variables
assembly-time 233
communal 217
environment 10, 213, 222
external 217, 369
floating-point 136–138
global 211
initializing 87
integers, allocating memory for 85–86
local address, loading 82
naming restrictions 9
Virtual memory 5
Virtual-86 mode 2, 335
Visibility
PROC statement 25, 185
scope, within 9

W

WDEB386 debugger 264
WEP (Windows Exit Procedure) 263–264, 270
.WHILE directive 173
WHILE directive 241
WIDTH operator 133
Windows operating system
API 257, 262

applications 258, 261
DLLs 261
Windows operating system *(continued)*
exit codes 263
MS-DOS, compared 4
programming for 4
protected mode 2, 6
SDK 268
task header 265, 269
Windows NT 3–5
WORD align type 45
WORD directive 86
Word size
default 13, 363, 373
expressions, in 13, 26
@WordSize predefined symbol 39
Words, reserved *See* Reserved words

X

XCHG instruction 90
/X command-line option, ML 213
XLAT instruction 116
XLATB instruction 116
XOR instruction 27, 99–100

Z

ZERO? flag as operand 178
/Zm command-line option, ML 62, 119
/Zp command-line option, ML 119

Documentation Feedback—Microsoft® Macro Assembler Version 6.1

Please help us improve our documentation. When you have used MASM 6.1 for a while, please complete and return this form. Use the back of the form for additional suggestions and comments. Suggestions and comments become the property of Microsoft Corporation.

Rate each component of the document set:

Rate each from 1 (never use) to 5 (often use).

1 2 3 4 5 Getting Started

1 2 3 4 5 Programmer's Guide

1 2 3 4 5 Environment and Tools

1 2 3 4 5 Reference

1 2 3 4 5 Online Help

List what stands out most about each component:

List the one or two things that you like, dislike, or both. Or, use the space to comment on the rating you gave the component in the previous section.

*Getting Started*_____

_____

_____

_____

Programmer's Guide _____

_____

_____

_____

Environment and Tools _____

_____

_____

_____

Reference _____

_____

_____

_____

Online Help_____

_____

_____

_____

What one thing would you like to see added to or removed from each component?

Getting Started _____

_____

_____

_____

Programmer's Guide _____

_____

_____

_____

Environment and Tools _____

_____

_____

_____

*Reference*_____

_____

_____

_____

Online Help _____

_____

_____

_____

How many years of programming experience do you have?

___With assembly language

___With other programming languages (including application macro languages)

Name _____

Address _____

City/State/Zip _____

Home Phone (_____) _____

Work Phone (_____) _____

May we contact you for additional information about your comments? _ Yes _ No

Additional comments:

.....Fold.....

Place stamp
here.
Post Office
will not deliver
without proper
postage.

Microsoft Corporation
Languages—MASM 6.1
One Microsoft Way
Redmond WA 98052-9953

Environment and Tools

Microsoft® MASM

Assembly-Language Development System
Version 6.1

For MS-DOS® and Windows™ Operating Systems

Microsoft Corporation

Information in this document is subject to change without notice. Companies, names, and data used in examples herein are fictitious unless otherwise noted. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Microsoft Corporation.

©1992 Microsoft Corporation. All rights reserved.

Microsoft, MS, MS-DOS, XENIX, CodeView, and QuickC are registered trademarks and Windows and Windows NT are trademarks of Microsoft Corporation in the USA and other countries.

U.S. Patent No. 4955066

IBM is a registered trademark of International Business Machines Corporation.
Intel is a registered trademark of Intel Corporation.
UNIX is a registered trademark of American Telephone and Telegraph Company.
BRIEF is a registered trademark of SDC Software Partners II L.P.
Printed in the United States of America.

Document No. DB35751-1292

Contents Overview

Introduction xxi

Part 1 The Programmer's WorkBench
Chapter 1 Introducing the Programmer's WorkBench 3
Chapter 2 Quick Start 7
Chapter 3 Managing Multimodule Programs 35
Chapter 4 User Interface Details 57
Chapter 5 Advanced PWB Techniques 77
Chapter 6 Customizing PWB 109
Chapter 7 Programmer's WorkBench Reference 131

Part 2 The CodeView Debugger
Chapter 8 Getting Started with CodeView 293
Chapter 9 The CodeView Environment 319
Chapter 10 Special Topics 351
Chapter 11 Using Expressions in CodeView 375
Chapter 12 CodeView Reference 393

Part 3 Compiling and Linking
Chapter 13 Linking Object Files with LINK 457
Chapter 14 Creating Module-Definition Files 491
Chapter 15 Using EXEHDR 513

Part 4 Utilities
Chapter 16 Managing Projects with NMAKE 527
Chapter 17 Managing Libraries with LIB 581
Chapter 18 Creating Help Files with HELPMAKE 593
Chapter 19 Browser Utilities 615
Chapter 20 Using Other Utilities 631

Part 5 Using Help
Chapter 21 Using Help 663

Appendixes
Appendix A Error Messages 685
Appendix B Regular Expressions 845

Glossary 857

Index 873

Contents

Introduction . . . . . **xxi**
Scope and Organization of This Book xxii
Microsoft Support Services. xxiii
Support Services Within the United States xxiii
Support Services Worldwide. xxvi
Document Conventions xxxiii

Part 1 The Programmer's WorkBench

Chapter 1 Introducing the Programmer's WorkBench **3**
What's in Part 1 4
Using the Tutorial. 4
Conventions in the Tutorial. 5

Chapter 2 Quick Start **7**
The PWB Environment. 7
The Microsoft Advisor. 8
Entering Text 9
Saving a File 10
Indenting Text with PWB 11
Opening an Existing File. 14
Copying, Pasting, and Deleting Text 16
Single-Module Builds 18
Setting Build Options 18
Setting Other Options. 21
Building the Program 22
Fixing Build Errors 23
Running the Program 25
Debugging the Program 26
Using CodeView to Isolate an Error 26
Working Through a Program to Debug it. 29
Examining Memory in the Memory Window 33
Where to Go from Here 34

Chapter 3 Managing Multimodule Programs **35**
Multimodule Program Example 35
Opening the Project 36

Contents of a Project 38
Dependencies in a Project 39
Building a Multimodule Program 40
Running the Program 40
Project Maintenance 41
Using Existing Projects 42
Adding and Deleting a Project File 44
Changing Assembler and Linker Options 46
Changing Options for Individual Modules 49
The Program Build Process 51
Extending a PWB Project 52
Using a Non-PWB Makefile 55
Where to Go from Here 56

Chapter 4 User Interface Details 57
Starting PWB 57
From the Command Line 57
Using the Windows Operating System Program Manager 58
Using the Windows Operating System File Manager 59
The PWB Screen 59
PWB Menus 64
File 64
Edit 64
Search 65
Project 66
Run 66
Options 67
Browse 68
Window 69
Help 70
Executing Commands 70
Choosing Menu Commands 70
Shortcut Keys 71
Buttons 72
Dialog Boxes 72

Chapter 5 Advanced PWB Techniques 77
Searching with PWB 77
Searching by Visual Inspection 78
Using the Find Command 79
Using Regular Expressions 82

Using the Source Browser 88
Advanced Browser Database Information 93
Executing Functions and Macros 96
Executing Functions and Macros by Name 98
Writing PWB Macros 98
When Is a Macro Useful? 99
Recording Macros 99
Flow Control Statements 102
User Input Statements 104

Chapter 6 Customizing PWB 109
Changing Key Assignments 109
Changing Settings 112
Customizing Colors 114
Adding Commands to the Run Menu 115
How PWB Handles Tabs 118
PWB Configuration 120
Autoloading Extensions 121
The TOOLS.INI File 122
TOOLS.INI Statement Syntax 124
Environment Variables 127
Current Status File CURRENT.STS 128
Project Status Files 129

Chapter 7 Programmer's WorkBench Reference 131
PWB Command Line 131
PWB Menus and Keys 132
PWB Default Key Assignments 135
Note on Available Keys 139
PWB Functions 140
Cursor-Movement Commands 144
Predefined PWB Macros 207
PWB Switches 244
Extension Switches 246
Filename-Parts Syntax 247
Boolean Switch Syntax 248
Browser Switches 286
Help Switches 287

Part 2 The CodeView Debugger

Chapter 8 Getting Started with CodeView 293
Preparing Programs for Debugging 293
General Programming Considerations 294
Compiling and Linking 295
Debugging Strategies 297
Identifying the Bug 297
Locating the Bug 298
Setting up CodeView 299
CodeView Files 300
Configuring CodeView with TOOLS.INI 301
CodeView TOOLS.INI Entries 302
Memory Management and CodeView 308
The CodeView Command Line 308
Leaving CodeView 309
Command-Line Options 310
The CURRENT.STS State File 316

Chapter 9 The CodeView Environment 319
The CodeView Display 319
The Menu Bar 320
The Window Area 320
The Status Bar 321
CodeView Windows 321
How to Use CodeView Windows 321
The Source Windows 324
The Watch Window 324
The Command Window 326
The Local Window 328
The Register Window 329
The 8087 Window 330
The Memory Windows 330
The Help Window 332
CodeView Menus 332
The File Menu 332
The Edit Menu 334
The Search Menu 335
The Run Menu 336
The Data Menu 338
The Options Menu 342

The Calls Menu 346
The Windows Menu 347
The Help Menu 349

Chapter 10 Special Topics 351
Debugging in the Windows Operating System 351
Comparing CVW with CV 351
Preparing to Run CVW 352
Starting a Debugging Session 353
CVW Commands 357
CVW Debugging Techniques 360
Debugging P-Code 363
Requirements 364
Preparing Programs 365
P-Code Debugging Techniques 365
P-Code Debugging Limitations 367
Remote Debugging 367
Requirements 368
Remote Monitor Command-Line Syntax 370
Starting a Remote Debugging Session 371

Chapter 11 Using Expressions in CodeView 375
Common Elements 375
Line Numbers 376
Registers 377
Addresses 378
Address Ranges 379
Choosing an Expression Evaluator 380
Using the C and C++ Expression Evaluators 381
Additional Operators 381
Unsupported Operators 381
Restrictions and Special Considerations 382
The Context Operator 382
Numeric Constants 384
String Literals 385
Symbol Formats 385
Using C++ Expressions 386
Access Control 386
Ambiguous References 386
Inheritance 386
Constructors, Destructors, and Conversions 387

Overloading . . . 388
Operator Functions . . . 388
Debugging Assembly Language . . . 389
Memory Operators . . . 389
Register Indirection . . . 390
Register Indirection with Displacement . . . 391
Address of a Variable . . . 391
PTR Operator . . . 391
Strings . . . 392
Array and Structure Elements . . . 392

Chapter 12 CodeView Reference . . . 393
CodeView Command Overview . . . 398
CodeView Command Reference . . . 400

Part 3 Compiling and Linking

Chapter 13 Linking Object Files with LINK . . . 457
New Features . . . 457
Overview . . . 458
LINK Output Files . . . 459
LINK Syntax and Input . . . 460
The objfiles Field . . . 461
The exefile Field . . . 462
The mapfile Field . . . 463
The libraries Field . . . 463
The deffile Field . . . 466
Examples . . . 467
Running LINK . . . 468
Specifying Input with LINK Prompts . . . 469
Specifying Input in a Response File . . . 469
LINK Options . . . 471
Specifying Options . . . 471
The /ALIGN Option . . . 472
The /BATCH Option . . . 472
The /CO Option . . . 473
The /CPARM Option . . . 473
The /DOSSEG Option . . . 474
The /DSALLOC Option . . . 475
The /DYNAMIC Option . . . 475

The /EXEPACK Option 475
The /FARCALL Option. 476
The /HELP Option 477
The /HIGH Option 477
The /INFO Option 477
The /LINE Option. 478
The /MAP Option. 478
The /NOD Option. 479
The /NOE Option 479
The /NOFARCALL Option 479
The /NOGROUP Option. 480
The /NOI Option. 480
The /NOLOGO Option 480
The /NONULLS Option 480
The /NOPACKC Option 481
The /NOPACKF Option 481
The /OLDOVERLAY Option. 481
The /ONERROR Option 481
The /OV Option. 482
The /PACKC Option 482
The /PACKD Option 483
The /PACKF Option 484
The /PAUSE Option. 484
The /PCODE Option 485
The /PM Option 485
The /Q Option 485
The /r Option 486
The /SEG Option 486
The /STACK Option 487
The /TINY Option 487
The /W Option. 488
The /? Option. 488
Setting Options with the LINK Environment Variable 488
Setting the LINK Environment Variable. 488
Behavior of the LINK Environment Variable 489
Clearing the LINK Environment Variable 489

LINK Temporary Files 489
LINK Exit Codes 490

Chapter 14 Creating Module-Definition Files 491
New Features 491
MS-DOS Programs 491
Statements 492
Overlays 492
Overview 492
Module Statements 493
Syntax Rules 494
The NAME Statement 495
The LIBRARY Statement 496
The DESCRIPTION Statement 496
The STUB Statement 497
The APPLOADER Statement 498
The EXETYPE Statement 498
The PROTMODE Statement 499
The REALMODE Statement 500
The STACKSIZE Statement 500
The HEAPSIZE Statement 500
The CODE Statement 501
The DATA Statement 501
The SEGMENTS Statement 502
CODE, DATA, and SEGMENTS Attributes 503
The OLD Statement 505
The EXPORTS Statement 505
The IMPORTS Statement 506
The FUNCTIONS Statement 508
The INCLUDE Statement 510
Reserved Words 510

Chapter 15 Using EXEHDR 513
Running EXEHDR 513
The EXEHDR Command Line 513
EXEHDR Options 514
Executable-File Format 515
EXEHDR Output: MS-DOS Executable File 516
EXEHDR Output: Segmented-Executable File 518
DLL Header Differences 519
Segment Table 520

Exports Table 520
EXEHDR Output: Verbose Output 521
MS-DOS Header Information 521
New .EXE Header Information 521
Tables 523
Relocations 523

Part 4 Utilities

Chapter 16 Managing Projects with NMAKE 527
New Features 527
Overview 528
Running NMAKE 529
Command-Line Options 529
NMAKE Command File 533
The TOOLS.INI File 534
Contents of a Makefile 535
Using Special Characters as Literals 535
Wildcards 536
Comments 536
Long Filenames 537
Description Blocks 537
Dependency Line 538
Targets 538
Dependents 542
Commands 543
Command Syntax 543
Command Modifiers 544
Exit Codes from Commands 545
Filename-Parts Syntax 546
Inline Files 547
Macros 550
User-Defined Macros 551
Using Macros 554
Special Macros 554
Substitution Within Macros 560
Substitution Within Predefined Macros 561
Environment-Variable Macros 561
Inherited Macros 563
Precedence Among Macro Definitions 563

Inference Rules 563
Inference Rule Syntax 564
Inference Rule Search Paths 565
User-Defined Inference Rules 566
Predefined Inference Rules 567
Inferred Dependents 569
Precedence Among Inference Rules 570
Directives 570
Dot Directives 570
Preprocessing Directives 572
Sequence of NMAKE Operations 576
A Sample NMAKE Makefile 578
NMAKE Exit Codes 580

Chapter 17 Managing Libraries with LIB 581
Overview 581
Running LIB 582
The LIB Command Line 582
LIB Command Prompts 582
The LIB Response File 583
Specifying LIB Fields 583
The Library File 584
LIB Options 584
LIB Commands 586
The Cross-reference Listing 590
The Output Library 590
Examples 591
LIB Exit Codes 592

Chapter 18 Creating Help Files with HELPMAKE 593
Overview 594
Running HELPMAKE 595
Encoding 595
Decoding 597
Getting Help 598
Other Options 599
Source File Formats 599
Elements of a Help Source File 600
Defining a Topic 600
Creating Links to Other Topics 601
Formatting Topic Text 604

Dot and Colon Commands 605
Other Help Text Formats 609
Rich Text Format 609
Minimally Formatted ASCII 612
Context Prefixes 613

Chapter 19 Browser Utilities 615
Overview of Database Building 616
Preparing to Build a Database 616
How BSCMAKE Builds a Database 616
Methods for Increasing Efficiency 617
BSCMAKE 618
System Requirements for BSCMAKE 618
The BSCMAKE Command Line 619
BSCMAKE Options 620
Using a Response File 622
BSCMAKE Exit Codes 623
SBRPACK 623
Overview of SBRPACK 624
The SBRPACK Command Line 624
SBRPACK Exit Codes 626
CREF 626
Using CREF 626
Difference from Previous Releases 629

Chapter 20 Using Other Utilities 631
CVPACK 631
Overview of CVPACK 632
The CVPACK Command Line 632
CVPACK Exit Codes 633
H2INC 633
Basic H2INC Operation 634
H2INC Syntax and Options 635
Converting Data and Data Structures 638
Converting Function Prototypes 648
Summary of H2INC-Recognized Keywords and Pragmas 651
IMPLIB 652
About Import Libraries 652
The IMPLIB Command Line 653
Options 653
RM, UNDEL, and EXP 654

Overview of the Backup Utilities 654
The RM Utility 654
The UNDEL Utility 655
The EXP Utility 656
WX/WXServer 657
Running WX/WXServer 657

Part 5 Using Help

Chapter 21 Using Help 663
Structure of the Microsoft Advisor 663
Navigating Through the Microsoft Advisor 664
Using the Help Menu 665
Using the Mouse and the F1 Key 666
Using Hyperlinks 666
Using Help Windows and Dialog Boxes 667
Accessing Different Types of Information 669
Using Different Help Screens 672
Using Help in PWB 673
Opening a Help File 673
Global Search 674
Using QuickHelp 675
Using the /Help Option 676
Using the QH Command 676
Managing Help Files 679
Managing Many Help Files 680

Appendixes

Appendix A Error Messages 685
Error Message Lists 685
BSCMAKE Error Messages 688
CodeView C/C++ Expression Evaluator Errors 692
CodeView Error Messages 700
CVPACK Error Messages 716
EXEHDR Error Messages 720
Math Coprocessor Error Messages 722
H2INC Error Messages 724
HELPMAKE Error Messages 761
IMPLIB Error Messages 767

LIB Error Messages 769
LINK Error Messages 775
ML Error Messages 798
NMAKE Error Messages 828
PWB Error Messages. 840
SBRPACK Error Messages 842

Appendix B Regular Expressions 845
Regular-Expression Summaries 845
UNIX Regular-Expression Syntax. 848
Tagged Regular Expressions. 850
Tagged Expressions in Build:Message. 852
Justifying Tagged Expressions 852
Predefined Regular Expressions. 853
Non-UNIX Regular-Expression Syntax 854
Non-UNIX Matching Method 855

Glossary 857

Index 873

Figures and Tables

Figures

Figure 2.1 PWB Display 8
Figure 0.1 The SHOW Project 36
Figure 0.2 The PWB Build Process 51
Figure 0.3 User Interface Elements 60
Figure 0.4 Window Elements 61
Figure 0.5 Status Bar Elements 62
Figure 0.6 PWB Menu Elements 63
Figure 0.7 Dialog Box Elements 73
Figure 0.8 Key Box and Check Box 74
Figure 0.9 Regular Expression Example 83
Figure 0.10 Complex Regular Expression Example 84
Figure 0.11 How PWB Displays Tabs 119
Figure 0.12 Arranged Windows 213
Figure 0.13 Vertical Tiling 278
Figure 0.14 Horizontal Tiling 278
Figure 0.15 CodeView Display 320
Figure 0.16 Format for a Segmented-Executable File 516
Figure 0.17 NMAKE Description Block 537
Figure 0.18 Microsoft Advisor Global Contents Screen 664

Tables

Table 0.1 File Menu and Keys 132
Table 0.2 Edit Menu and Keys 133
Table 0.3 Search Menu and Keys 133
Table 0.4 Project Menu and Keys 134
Table 0.5 Run Menu and Keys 134
Table 0.6 Browse Menu and Keys 134
Table 0.7 Window Menu and Keys 135
Table 0.8 Help Menu and Keys 135
Table 0.9 PWB Default Key Assignments 136
Table 0.10 PWB Functions 140
Table 0.11 Cursor-Movement Commands 145
Table 0.12 PWB Macros 208
Table 0.13 PWB Color Names 252
Table 0.14 PWB Color Values 254

Table 0.1 CodeView TOOLS.INI Entries 302
Table 0.2 CodeView Command-Line Options 310
Table 0.1 Moving Around with the Keyboard 323
Table 0.1 Registers 377
Table 0.1 Register Names 395
Table 0.2 CodeView Command Summary 398
Table 0.1 Module Statements 493
Table 0.1 Predefined Inference Rules 567
Table 0.2 Binary Operators for Preprocessing 574
Table 0.2 Formatting Attributes 605
Table 0.3 Dot and Colon Commands 606
Table 0.4 RTF Formatting Codes 610
Table 0.5 Microsoft Product Context Prefixes 613
Table 0.6 Standard h. Contexts 614
Table A.1 Error Codes Listed by Utility 686
Table A.2 Error Codes Listed by Error Code Range 687
Table .1 UNIX Regular-Expression Summary 845
Table .2 UNIX Predefined Expressions 846
Table .3 CodeView Regular Expressions 847
Table .4 Non-UNIX Regular-Expression Summary 847
Table .5 Non-UNIX Predefined Expressions 848
Table .6 UNIX Regular-Expression Syntax 848
Table .7 Predefined Regular Expressions and Definitions 853
Table .8 Non-UNIX Regular Expression Syntax 854

Introduction

Microsoft® Macro Assembler (MASM) includes a full set of development tools — editor, compiler, linker, debugger, and browser — for writing, compiling, and debugging your programs. You can work within the Microsoft Programmer's WorkBench (PWB) integrated environment, or you can use the tools separately to develop your programs.

Environment and Tools describes the following development tools:

- The Programmer's WorkBench (PWB). PWB is a comprehensive tool for application development. Within its environment is everything you need to create, build, browse, and debug your programs. Its macro language gives you control over not only editing but also build operations and other PWB functions.
- The Microsoft® CodeView™ debugger. This is a diagnostic tool for finding errors in your programs. Two versions of CodeView are described: one for MS-DOS® and one for Microsoft Windows™. Each CodeView version has specialized commands for its operating environment, as well as other commands for examining code and data, setting breakpoints, and controlling your program's execution.
- LINK, the Microsoft Segmented-Executable Linker. The linker combines object files and libraries into an executable file, either an application or a dynamic-link library (DLL).
- EXEHDR, the Microsoft EXE File Header Utility. EXEHDR displays and modifies the contents of an executable-file header.
- NMAKE, the Microsoft Program Maintenance Utility. NMAKE simplifies project maintenance. Once you specify which project files depend on others, you can use NMAKE to automatically execute the commands that will update your project when any file has changed.
- LIB, the Microsoft Library Manager. LIB creates and maintains standard libraries. With LIB, you can create a library file and add, delete, and replace modules.

- HELPMAKE, the Microsoft Help File Maintenance Utility. HELPMAKE creates and maintains Help files. You can use HELPMAKE to create a Help file or to customize the Microsoft Help files.
- BSCMAKE, the Microsoft Browser Database Maintenance Utility, and SBRPACK, the Microsoft Browse Information Compactor. BSCMAKE creates browser files for use with the PWB Source Browser. SBRPACK compresses the files that are used by BSCMAKE.

Environment and Tools also describes these special-purpose utilities:

- H2INC, the Microsoft C Header Translation Utility. H2INC translates C header files into MASM-compatible include files.
- CVPACK, the Microsoft Debugging Information Compactor. CVPACK compresses the size of debugging information in an executable file.
- IMPLIB, the Microsoft Import Library Manager. IMPLIB creates an import library that resolves external references from a Windows-based application to a DLL.
- RM, the Microsoft File Removal Utility; UNDEL, the Microsoft File Undelete Utility; and EXP, the Microsoft File Expunge Utility. These utilities manage, delete, and recover backup files.

Scope and Organization of This Book

This book has five parts and five appendixes to give you complete information about PWB, CodeView, and the utilities included with MASM.

Part 1 is a brief PWB tutorial and comprehensive reference. The first three chapters introduce PWB and provide a tutorial that describes the features of the integrated environment and how to use them. Chapters 4, 5, and 6 contain detailed information on the interface, advanced PWB techniques, and customization. Chapter 7 contains a complete reference to PWB's default keys and all functions, predefined macros, and switches.

Part 2 provides full information on the Microsoft CodeView debugger. Chapter 8 tells how to prepare programs for debugging, how to start CodeView, and how to customize CodeView's interface and memory usage. Chapter 9 describes the environment, including the CodeView menu commands and the format and use of each CodeView window. Chapter 10 explains how to use expressions, including the C and C++ expression evaluators. Chapter 11 describes techniques for debugging Windows-based programs. Chapter 12 contains a complete reference to CodeView commands.

The chapters in Parts 3 and 4 describe the utilities. These chapters are principally for command-line users. Even if you're using PWB, however, you may find the detailed information in Parts 3 and 4 helpful for a better

understanding of how each tool contributes to the program development process.

Part 3 provides information about compiling and linking your program. LINK command-line syntax and options are covered in Chapter 13. The contents and use of module-definition files are explained in Chapter 14. Chapter 15 describes how to use EXEHDR to examine the file header of a program.

Part 4 presents the other utilities. NMAKE, the utility for automating project management, is described in Chapter 16. Chapter 17 covers LIB, used in managing standard libraries. Procedures for using HELPMAKE to create and maintain Help files are in Chapter 18. The tools for creating a browser database are discussed in Chapter 19. Finally, Chapter 20 describes how to use the following special-purpose utilities: H2INC, CVPACK, IMPLIB, RM, UNDEL, and EXP.

Part 5 presents the Microsoft Advisor Help system and the QuickHelp program. It describes the structure of the Help files, how to navigate through the Help system, and how to manage Help files.

The appendixes provide supplementary information. Appendix A describes error messages. Appendix B describes regular expressions for use in PWB and CodeView.

Microsoft Support Services

Microsoft offers a variety of no-charge and fee-based support options to help you get the most from your Microsoft product. For an explanation of these options, please see one of the following sections:

- If you are in the United States, see “Support Services Within the United States.”
- If you are outside the United States, see “Support Services Worldwide.”

Support Services Within the United States

If you have a question about Microsoft Macro Assembler (MASM), one of the following resources may help you find an answer:

- The index in the product documentation and other printed product documentation.
- Context-sensitive online Help available from the Help menu.

- The README files that come with your product disks. These files provide general information that became available after the books in the product package were published.
- Electronic options such as CompuServe forums or bulletin board systems, if available.

If you cannot find the information you need, you can obtain product support through several methods. In addition, you can locate training and consultation services in your area.

For information about Microsoft incremental fee-based support service options, call Microsoft Inside Sales at (800) 227-4679, Monday through Friday, between 6:30 a.m. and 5:30 p.m. Pacific time.

Note Microsoft's support services are subject to Microsoft's prices, terms, and conditions in place in each country at the time the services are used.

Microsoft Forums on CompuServe

Microsoft Product Support Services are available on several CompuServe forums. For an introductory CompuServe membership kit specifically for Microsoft users, dial (800) 848-8199 and ask for operator 230. If you are already a CompuServe member, type `go microsoft` at any `!` prompt.

Microsoft Product Support Services

You can reach Microsoft Product Support Services Monday through Friday between 6:00 a.m. and 6:00 p.m. Pacific time.

- For assistance with Microsoft MASM, dial (206) 646-5109.

When you call, you should be at your computer with Microsoft MASM running and the product documentation at hand. Have your file open and be prepared to give the following information:

- The version of Microsoft MASM you are using.
- The type of hardware you are using, including network hardware, if applicable.
- The operating system you are using.
- The exact wording of any messages that appeared on your screen.
- A description of what happened and what you were trying to do when the problem occurred.
- A description of how you tried to solve the problem.

Microsoft Product Support for the Deaf and Hard-of-Hearing

Microsoft Product Support Services are available for the deaf and hard-of-hearing Monday through Friday between 6:00 a.m. and 6:00 p.m. Pacific time.

Using a special TDD/TT modem, dial (206) 635-4948.

Product Training and Consultation Services

Within the United States, Microsoft offers the following services for training and consultation:

Authorized Training Centers

Microsoft Authorized Training Centers offer several services for Microsoft product users. These include:

- Customized training for users and trainers.
- Training material development.
- Consulting services.

For information about the training center nearest you, call Microsoft Consumer Sales at (800) 426-9400 Monday through Friday between 6:30 a.m. and 5:30 p.m. Pacific time.

Consultant Referral Service

Microsoft's Consultant Relations Program can refer you to an independent consultant in your area. These consultants are skilled in:

- Macro development and translation.
- Database development.
- Custom interface design.

For information about the consultants in your area, call the Microsoft Consultant Relations Program at (800) 227-4679, extension 56042, Monday through Friday between 6:30 a.m. and 5:30 p.m. Pacific time.

Support Services Worldwide

If you are outside the United States and have a question about Microsoft MASM, Microsoft offers a variety of no-charge and fee-based support options. To solve your problem, you can:

- Consult the index in the product documentation and other printed product documentation.
- Check context-sensitive online Help available from the Help menu.
- Check the README files that come with your product disks. These files provide general information that became available after the books in the product package were published.
- Consult electronic options such as CompuServe forums or bulletin board systems, if available.

If you cannot find a solution, you can receive information on how to obtain product support by contacting the Microsoft subsidiary office that serves your country.

Note Microsoft's support services are subject to Microsoft's prices, terms, and conditions in place in each country at the time the services are used.

Calling a Microsoft Subsidiary Office

When you call, you should be at your computer with Microsoft MASM running and the product documentation at hand. Have your file open and be prepared to give the following information:

- The version of Microsoft MASM you are using.
- The type of hardware you are using, including network hardware, if applicable.
- The operating system you are using.
- The exact wording of any messages that appeared on your screen.
- A description of what happened and what you were trying to do when the problem occurred.
- A description of how you tried to solve the problem.

Microsoft subsidiary offices and the countries they serve are listed below.

Area	Telephone Numbers
Argentina	Microsoft de Argentina S.A. Phone: (54) (1) 814-0356 Fax: (54) (1) 814-0372
Australia	Microsoft Pty. Ltd. Phone: (61) (02) 870-2200 Fax: (02) 805-1108 Bulletin Board Service: (612) 870-2348 Technical Support: (61) (02) 870-2131 Sales Information Centre: (02) 870-2100
Austria	Microsoft Ges.m.b.H. Phone: 0222 - 68 76 07 Fax: 0222 - 68 16 2710 Information: 060 - 89 - 247 11 101 Prices, updates, etc.: 060 - 89 - 3176 1199 CompuServe: msce (Microsoft Central Europe) Technical support: Windows, Windows for Workgroups, Microsoft Mail: 0660 - 65 - 10 Microsoft Excel for Windows, Microsoft Excel for OS/2, PowerPoint for Windows: 0660 - 65 - 11 Word for MS-DOS, Windows Write: 0660 65 - 12 Word for Windows, Word for OS/2: 0660 - 65 - 13 Works for MS-DOS, Works for Windows, Publisher, WorksCalc, WorksText: 0660 - 65 - 14 C PDS, FORTRAN PDS, Pascal, Macro Assembler PDS, QuickC, QuickC for Windows, QuickPascal, QuickAssembler, Profiler: 0660 - 65 - 15 COBOL PDS, Basic PDS, QuickBASIC, Visual Basic: 0660 - 65 - 16 MS-DOS: 0660 - 65 - 17 Macintosh Software: 0660 - 65 - 18 Project for Windows, Project for MS-DOS, Multiplan, Mouse, Flight Simulator, Paintbrush, Chart: 0660 - 67 - 38 FoxPro: 0660 - 67 - 61
Baltic States	See Germany
Belgium	Microsoft NV Phone: 02-7322590 Fax: 02-7351609 Technical Support Bulletin Board Service: 02-7350045 (1200/2400/9600 baud, 8 bits, no parity, 1 stop bit, ANSI terminal emulation) (Dutch speaking) Technical Support: 02-5133274 (English speaking) Technical Support: 02-5023432 (French speaking) Technical Support: 02-5132268 Technical Support Fax: (31) 2503-24304
Bermuda	See Venezuela
Bolivia	See Argentina

Area	Telephone Numbers
Brazil	Microsoft Informatica Ltda. Phone: (55) (11) 530-4455 Fax: (55) (11) 240-2205 Technical Support Phone: (55) (11) 533-2922 Technical Support Fax: (55) (11) 241-1157 Technical Support Bulletin Board Service: (55) (11) 543-9257
Canada	Microsoft Canada Inc. Phone: 1 (416) 568-0434 Fax: 1 (416) 568-4689 Technical Support Phone: 1 (416) 568-3503 Technical Support Facsimile: 1 (416) 568-4689 Technical Support Bulletin Board Service: 1 (416) 507-3022
Caribbean Countries	See Venezuela
Central America	See Venezuela
Chile	See Argentina
Colombia	See Venezuela
Denmark	Microsoft Denmark AS Phone: (45) (44) 89 01 00 Fax: (45) (44) 68 55 10
Ecuador	See Venezuela
England	Microsoft Limited Phone: (44) (734) 270000 Fax: (44) (734) 270002 Upgrades: (44) (81) 893-8000 Technical Support: Main Line (All Products): (44) (734) 271000 Windows Direct Support Line: (44) (734) 271001 Database Direct Support Line: (44) (734) 271126 MS-DOS 5 Warranty Support: (44) (734) 271900 MS-DOS 5 Fee Support Line: (44) (891) 315500 OnLine Service Assistance: (44) (734) 270374 Bulletin Board Service: (44) (734) 270065 (2400 Baud) Fax Information Service: (44) (734) 270080
Finland	Microsoft OY Phone: (358) (0) 525 501 Fax: (358) (0) 522 955
France	Microsoft France Phone: (33) (1) 69-86-46-46 Telex: MSPARIS 604322F Fax: (33) (1) 64-46-06-60 Technical Support Phone: (33) (1) 69-86-10-20 Technical Support Fax: (33) (1) 69-28-00-28

Area	Telephone Numbers
French Polynesia	See France
Germany	Microsoft GmbH Phone: 089 - 3176-0 Telex: (17) 89 83 28 MS GMBH D Fax: 089 - 3176-1000 Information: 0130 - 5099 Prices, updates, etc.: 089 - 3176 1199 Bulletin board, device drivers, tech notes : BTX: microsoft# or *610808000# CompuServe: msce (Microsoft Central Europe) Technical support: Windows, Windows for Workgroups, Microsoft Mail: 089 - 3176 - 1110 Microsoft Excel for Windows, Microsoft Excel for OS/2, PowerPoint for Windows: 089 - 3176 - 1120 Word for MS-DOS, Windows Write: 089 - 3176 - 1130 Word for Windows, Word for OS/2: 089 - 3176 - 1131 Works for MS-DOS, Works for Windows, Publisher, WorksCalc, WorksText: 089 - 3176 - 1140 C PDS, FORTRAN PDS, Pascal, Macro Assembler PDS, QuickC, QuickC for Windows, QuickPascal, QuickAssembler, Profiler: 089 - 3176 - 1150 COBOL PDS, Basic PDS, QuickBASIC, Visual Basic: 089 - 3176 - 1151 MS-DOS: 089 - 3176 - 1152 Macintosh Software: 089 - 3176 - 1160 Project for Windows, Project for MS-DOS, Multiplan, Mouse, Flight Simulator, Paintbrush, Chart: 089 - 3176 - 1170 FoxPro: 089 - 3176 - 1180
Hong Kong	Microsoft Hong Kong Limited Technical Support: (852) 804-4222
Ireland	See England
Israel	Microsoft Israel Ltd. Phone: 972-3-752-7915 Fax: 972-3-752-7919
Italy	Microsoft SpA Phone: (39) (2) 269121 Telex: 340321 I Fax: (39) (2) 21072020 Technical Support: Microsoft Excel for Windows, Project for Windows, Works for Windows: (39) (2) 26901361 Word, Works for MS-DOS: (39) (2) 26901362 Windows, PowerPoint, Publisher, Windows for Workgroups, Works : (39) (2) 26901363 Basic, COBOL, Visual Basic, MS-DOS-based, Fox Products: (39) (2) 26901364 C, FORTRAN, Pascal, Macro Assembler (MASM), and SDKs: (39) (2) 26901354 LAN Manager, SQL Server, Microsoft Mail, Microsoft Mail Gateways: (39) (2) 26901356

Area	Telephone Numbers
Japan	Microsoft Company Ltd. Phone: (81) (3) 3363-1200 Fax: (81) (3) 3363-1281 Technical Support: MS-DOS-based Applications: (81) (3) 3363-0160 Windows-based Applications: (81) (3) 3363-5040 Language Products (Microsoft C, Macro Assembler [MASM], QuickC): (81) (3) 3363-7610 Language Products (Basic, FORTRAN, Visual Basic, Quick Basic): (81) (3) 3363-0170 All Products Technical Support Fax: (81) (3) 3363-9901
Korea	Microsoft CH Phone: (82) (2) 552-9505 Fax: (82) (2) 555-1724 Technical Support: (82) (2) 563-9230
Liechtenstein	See Switzerland (German speaking)
Luxemburg	Microsoft NV Phone: (32) 2-7322590 Fax: (32) 2-7351609 Technical Support Bulletin Board Service: (31) 2503-34221 (1200/2400/9600 baud, 8 bits, No parity, 1 stop bit, ANSI terminal emulation) (Dutch speaking) Technical Support: (31) 2503-77877 (English speaking) Technical Support: (31) 2503-77853 (French speaking) Technical Support: (32) 2-5132268 Technical Support Fax: (31) 2503-24304
México	Microsoft México, S.A. de C.V. Phone: (52) (5) 325-0910 Fax: (52) (5) 280-0198 Technical Support: (52) (5) 325-0912 Sales: (52) (5) 325-0911
Netherlands	Microsoft BV Phone: 02503-13181 Fax: 02503-37761 Technical Support Bulletin Board Service: 02503-34221 (1200/2400/9600 baud, 8 bits, No parity, 1 stop bit, ANSI terminal emulation) (Dutch speaking) Technical Support: 02503-77877 (English speaking) Technical Support: 02503-77853 Technical Support Fax: 02503-24304
New Zealand	Technology Link Centre Phone: 64 (9) 358-3724 Fax: 64 (9) 358-3726 Technical Support Applications: 64 (9) 357-5575
Northern Ireland	See England

Area	Telephone Numbers
Norway	Microsoft Norway AS Phone: (47) (2) 95 06 65 Fax: (47) (2) 95 06 64 Technical Support: (47) (2) 18 35 00
Papua New Guinea	See Australia
Paraguay	See Argentina
Peru	See Venezuela
Portugal	MSFT, Lda. Phone: (351) 1 4412205 Fax: (351) 1 4412101
Puerto Rico	See Venezuela
Republic of China	Microsoft Taiwan Corp. Phone: (886) (2) 504-3122 Fax: (886) (2) 504-3121
Republic of Ireland	See England
Scotland	See England
Spain	Microsoft Iberica SRL Phone: (34) (1) 804-0000 Fax: (34) (1) 803-8310 Technical Support: (34) (1) 803-9960
Sweden	Microsoft AB Phone: (46) (8) 752 56 00 Fax: (46) (8) 750 51 58 Technical Support: Applications: (46) (8) 752 68 50 Development and Network products: (46) (8) 752 60 50 MS-DOS: (46) (071) 21 05 15 (SEK 4.55/min) Sales Support: (46) (8) 752 56 30 Bulletin Board Service: (46) (8) 750 47 42 Fax Information Service: (46) (8) 752 29 00

Area	Telephone Numbers
Switzerland	(German speaking) Microsoft AG Phone: 01 - 839 61 11 Fax: 01 - 831 08 69 Infomation: 0049 - 89 - 247 11 101 Prices, updates, etc.: 0049 - 89 - 3176 1199 CompuServe: msce (Microsoft Central Europe) Technical support: Windows, Windows for Workgroups, Microsoft Mail: 01 - 342 - 4085 Microsoft Excel for Windows, Microsoft Excel for OS/2, PowerPoint for Windows: 01 - 342 - 4082 Word for MS-DOS, Windows Write: 01 - 342 - 4083 Word for Windows, Word for OS/2: 01 - 342 - 4087 Works for MS-DOS, Works for Windows, Publisher, WorksCalc, WorksText: 01 - 342 - 4084 C PDS, FORTRAN PDS, Pascal, Macro Assembler PDS, QuickC, QuickC for Windows, QuickPascal, QuickAssembler, Profiler: 01 - 342 - 4036 COBOL PDS, Basic PDS, QuickBASIC, Visual Basic: 01 - 342 - 4086 MS-DOS: 01 - 342 - 2152 Macintosh Software: 01 - 342 - 4081 Project for Windows, Project for MS-DOS, Multiplan, Mouse, Flight Simulator, Paintbrush, Chart : 01 - 342 - 0322 FoxPro: 01 / 342 - 4121 (French speaking) Microsoft SA, office Nyon Phone: 022 - 363 68 11 Fax: 022 - 363 69 11 Technical support: 022 - 738 96 88
Uruguay	See Argentina
Venezuela	Corporation MS 90 de Venezuela S.A. Phone: 0058.2.914739 Fax: 0058.2.923835
Wales	See England
Venezuela	Phone: 0058.2.914739 Fax: 0058.2.923835

Document Conventions

This book uses the following typographic conventions:

Examples	Description
README.TXT, COPY, LINK, /CO	Uppercase (capital) letters indicate filenames, MS-DOS commands, and the commands to run the tools. Uppercase is also used for command-line options, unless the option must be lowercase.
printf, **IMPORT**	Bold letters indicate keywords, library functions, reserved words, and CodeView commands. Keywords are required unless enclosed in double brackets as explained below.
expression	Words in italic are placeholders for information that you must supply (for example, a function argument).
[[*option*]]	Items inside double square brackets are optional.
{*choice1* | *choice2*}	Braces and a vertical bar indicate a choice between two or more items. You must choose one of the items unless all the items are also enclosed in double square brackets.
`CL ONE.C TWO.C`	This font is used for program examples, user input, program output, and error messages within the text.
Repeating elements...	Three horizontal dots following an item indicate that more items having the same form may follow.

`while( )`
`{`
`.`
`.`
`.`
`}`
	Three vertical dots following a line of code indicate that part of the example program has intentionally been omitted.

F1, ALT+A	Small capital letters indicate the names of keys and key sequences, such as ENTER and CTRL+C. A plus (+) indicates a combination of keys. For example, CTRL+E means to hold down the CTRL key while pressing the E key. The cursor-movement keys on the numeric keypad are called ARROW keys. Individual ARROW keys are referred to by the direction of the arrow on the top of the key (LEFT, RIGHT, UP, DOWN). Other keys are referred to by the name on the top of the key (PGUP, PGDN).

Examples	Description
Arg Meta Delete (ALT+A ALT+A SHIFT+DEL)	A bold series of names followed by a series of keys indicates a sequence of PWB functions that you can use in a macro definition, type in a dialog box, or execute directly by pressing the sequence of keys. In this book, these keys are the default keys for the corresponding functions. Some functions are not assigned to a key, and the word "Unassigned" appears in the place of a key. In PWB Help, the current key that is assigned to the function is shown.
"defined term"	Quotation marks usually indicate a new term defined in the text.
dynamic-link library (DLL)	Acronyms are usually spelled out the first time they are used.

P A R T 1

The Programmer's WorkBench

Chapter 1 Introducing the Programmer's WorkBench 3
Chapter 2 Quick Start. 7
Chapter 3 Managing Multimodule Programs. 35
Chapter 4 User Interface Details 57
Chapter 5 Advanced PWB Techniques 77
Chapter 6 Customizing PWB 109
Chapter 7 Programmer's WorkBench Reference. 131

CHAPTER 1

Introducing the Programmer's WorkBench

The Microsoft Programmer's WorkBench (PWB) is a powerful tool for application development. PWB combines the following features:

- A full-featured programmer's text editor.
- An extensible "build engine" which allows you to assemble and link your programs using the PWB environment. The build engine can be extended to support any programming tool.
- Error-message browsing. Once a build completes, you can step through the build messages, fixing errors in your source programs.
- A Source Browser. When working with large systems, it is often difficult to remember where program symbols are accessed and defined. The Source Browser maintains a database that allows you to go quickly to where a given variable, function, type, class, or macro is defined or referenced.
- An extensible Help system. The Microsoft Advisor Help system provides a complete reference on using PWB and MASM. You can also write new Help files and seamlessly integrate them into the Help system to document your own library routines or naming conventions.
- A macro language that can control editing functions, program builds, and other PWB operations.

For increased flexibility, you can write extensions to PWB. These extensions can perform tasks that are inconvenient in the PWB macro language. For example, you can write extensions to perform file translations, source-code formatting, text justification, and so on. As with the macro language, PWB extensions have full access to most PWB capabilities. For information about how to write PWB extensions, see the Microsoft Advisor Help system (choose "PWB Extensions" from the main Help table of contents).

PWB comes with extensions for C/C++, Basic, and Fortran, in addition to assembly language, to facilitate mixed-language programming. To install one of

these extensions, simply rename its corresponding .XXT file to a .MXT file in the \BIN subdirectory where you installed MASM, as described in *Getting Started*. Also, because an increasing number of programmers are using C++, the PWB Browser extension supports classes.

What's in Part 1

This part of the book introduces you to the fundamentals of PWB. Chapter 2, "Quick Start," shows you how to use the PWB editor and build a simple single-module program from PWB. Chapter 3, "Managing Multimodule Programs," expands upon the information you learned in Chapter 2. It teaches you how to build a more complicated program that consists of several modules. You should be able to work through these two chapters in less than three hours.

As you work through these chapters, you may want to refer to Chapter 4, "User Interface Details," which explains options for starting PWB, briefly describes all of the menu commands, and summarizes how menus and dialog boxes work. The user interface information is presented in one chapter for easy access.

Chapter 5, "Advanced PWB Techniques," shows how to use the PWB search facilities (including searching with regular expressions), how to use the Source Browser, how to execute functions and macros, and how to write PWB macros.

Chapter 6, "Customizing PWB," describes how to redefine key assignments, change PWB settings, add commands to the PWB menu, and use the TOOLS.INI initialization file to store startup and configuration information for PWB.

Chapter 7, "PWB Reference," contains an alphabetical reference to PWB menus, keys, functions, predefined macros, and switches. It contains the essential information you need to know to take the greatest advantage of PWB's richly customizable environment.

Using the Tutorial

You probably want to get right to work with MASM. The tutorial chapters 2 and 3 can help you become productive very quickly. To get the most out of this material, here are a few recommendations:

- Follow the steps presented in the tutorial. It is always tempting to explore the system and find out more about the product through independent research. However, just as programming requires an orderly sequence of steps, some aspects of PWB also require sequenced actions.

- If you complete a step and something seems wrong—for example, if your screen doesn't match what is in the book—back up and try to find out what's wrong. Troubleshooting tips will help you take corrective actions.
- When working through this tutorial, consider how you might use these techniques in your own work. PWB is like a full tool chest. You probably won't learn (or even want to learn) all of PWB's capabilities right away. But as time goes on, you'll have uses for many of the tools you don't use immediately.

Conventions in the Tutorial

Procedures described in the course of the tutorial are introduced with headings designated by a triangular symbol. A list of the steps making up the procedure then follows. For example:

To open a file:

1. From the File menu, choose Open.

   PWB displays the Open File dialog box.
2. In the File List list box, select the file that you want to open.
3. Choose OK.

In procedures, the heading gives you a capsule summary of what the steps will accomplish. Each numbered step is an action you take to complete the procedure. Some steps are followed by an explanation, an illustration, or both.

CHAPTER 2

Quick Start

This chapter gets you started with PWB. You'll learn the basics by building and debugging a C-callable routine that generates a 2-byte pseudo-random number.

Some of the source code that you will be using is included with the sample programs shipped with MASM 6.1. If you chose not to install the sample code when you set up MASM, run SETUP to install it (see *Getting Started* for more information).

To start PWB in the Windows operating system for this tutorial, double-click the PWB icon in the MASM group.

In MS-DOS, type

```
PWB
```

at the prompt.

To leave PWB at any time:

- From the File menu, choose Exit, or press ALT+F4.

The PWB Environment

If this is the first time you have used PWB, you see the menu bar, the status bar, and an empty desktop (assuming a standard installation). If you have used PWB before, it opens the file you last worked with.

PWB uses a windowed environment to present information, get information from you, and allow you to edit programs. The environment has the following components:

- An editor for writing and revising programs
- A "build engine"—the part of PWB that helps you assemble, link, and execute your programs from within the environment

- A source-code browser
- Commands for program execution and debugging
- The Microsoft Advisor Help system

The browser and the Help system are dynamically loaded extensions to the PWB platform. Microsoft languages and the utilities are also supported in PWB by extensions. Other extensions are available, such as the Microsoft Source Profiler. PWB presents all of these components through menus and dialog boxes.

The following figure shows some parts of the PWB interface.

Figure 2.1 PWB Display

Chapter 4, “User Interface Details,” contains a thorough description of these elements and the rest of the PWB environment. Refer to this chapter when you need specific information about an unfamiliar interface element.

The Microsoft Advisor

PWB makes programming easier by providing the Microsoft Advisor Help system, which contains comprehensive information about:

- PWB editing functions
- PWB advanced features

- PWB menus and dialog boxes
- CodeView debugger
- Intel 80x86 assembly language
- MASM 6.1 assembler options
- Microsoft utilities (such as NMAKE, LINK, and so on)

The Advisor provides context-sensitive Help and general Help. Context-sensitive Help provides information about the menu, dialog box, or language element at the cursor. To see context-sensitive Help, you can simply point to an item on the screen and press either the right mouse button or the F1 key. PWB displays a Help window showing the requested information. You can also get context-sensitive Help and more general Help by using the Help menu.

To answer questions of a less specific nature, you can access the Contents screen by choosing Contents from the Help menu or by pressing SHIFT+F1. From the Advisor contents, you can access Help on any other subject in the database.

To get started using the Microsoft Advisor:

- From the Help menu, choose the Help on Help command.

Help on Help teaches you how to use the Microsoft Advisor Help system. For more information on using Help, see Chapter 21.

To close the Help window:

- Click the upper-left corner of the Help window (the Close box), press ESC, choose Close from the File menu, or press CTRL+F4.

Note Click the Close box, choose Close from the File menu, or press CTRL+F4 to close any open window in PWB.

The following sections explain basic editing procedures. If you're already familiar with these, you can skip to "Opening an Existing File" on page 14.

Entering Text

In this section, you'll learn basic PWB procedures by entering a simple C-callable assembly-language routine.

To start a new file:

1. Move the mouse cursor ("point") to the File menu on the menu bar and click the left button, or press ALT+F from the keyboard.

   PWB opens the File menu.

2. Point to the New command and click the left button, or press N to choose New.

   PWB opens a window with the title `Untitled.001`.

Pressing the ALT key from the keyboard changes focus to the menu bar, and pressing the highlighted key in a menu name opens that menu. Similarly, within a menu, pressing a key highlighted in one of the commands causes that command to be carried out. Using the keyboard, you can also easily move to the beginning of a file by typing CTRL+HOME, or to the end of a file by typing CTRL+END.

Starting with your cursor in the upper-left corner of the edit window, type the following comment line:

```
; C-CALLABLE PSEUDO-RANDOM NUMBER GENERATOR ROUTINE
```

Your screen should appear as follows:

Saving a File

Now that you've started entering your program, save your work before proceeding.

To save a file:

- From the File menu, choose Save, or press SHIFT+F2.

  PWB displays the Save As dialog box.

This dialog box has several options that you use to pass information to PWB. PWB indicates the active option — in this case, the File Name text box — by highlighting the area in which you can enter text. For more information about dialog boxes, see Chapter 4, "User Interface Details."

Because you have not yet saved the file, it still has the name `Untitled.001`. Type `ONEOF.ASM` in the File Name text box. Then click OK or press ENTER to save the file (if you want, you can first select the directory where the file will be saved, using the Drives / Dirs list box).

Note Now that you have named your file, choosing Save from the File menu does not bring up a dialog box. Your file is immediately saved to disk.

Indenting Text with PWB

Most assembly-language programmers format their code in several text columns (for example, a label column, an instruction column, a parameter column, and a comment column). You can create these columns differently in PWB than in other text editors. In PWB, you can move the cursor ("point") to any position on the screen and start typing text. PWB will take care of inserting whatever new lines, spaces, or tabs are necessary to place the text in the position you are typing. By setting options, you can determine whether PWB will use spaces or tab characters to create the necessary white space (see "How PWB Handles Tabs" on page 118).

Type the following comment lines to document the routine:

```
;       unsigned int OneOf ( unsigned int range )
;       ---------------------------------------------------------------
;       Routine uses a linear congruential method to calculate
;       a pseudo-random number, treats the number as a fraction
;       between 0 and 1, multiplies it times the range,
;       truncates the result to an integer, and returns it.
;
; Algorithm:   a[i] = ( ( a[i-1] * b ) + 1 ) mod m
;              where b = 4961 and m = 2^16
;
OneOf   PROC NEAR C PUBLIC USES bx dx, range:WORD
OneOf   ENDP
```

When you enter assembly-language code, you will often be adding a line indented to the same column as the line above. PWB saves you time by automatically indenting new lines when you press the ENTER key.

- If there is no line or a blank line immediately below the new line, PWB matches the indentation of the line above it.
- If there is a line immediately below the new line, PWB matches the indentation of the line below it.

You'll now type some text after the line containing the PROC NEAR directive.

To insert space for a new line using a mouse:

1. Position the cursor anywhere past the end of the line containing PROC NEAR. Precise positioning of the cursor is not critical because (by default) PWB trims trailing spaces from the end of your lines.
2. Click the left mouse button.
3. Press ENTER to make a new line.

   If you are in overtype mode, change to insert mode by pressing the INS key. Otherwise, pressing ENTER simply moves the cursor to the beginning of the next line. PWB displays the letter O on the status bar and shows the cursor as an underscore to signal that you are in overtype mode.

To insert the new line using the keyboard:

1. Move the cursor to the line containing the PROC NEAR directive by pressing the UP ARROW key.
2. Press END to move the cursor to the end of the line.
3. Press ENTER to make a new line.

Now type the following lines, using the TAB key to indent and space the instructions:

```
        mov     ax, 4961        ; Load the constant into AX and multiply
        mul     rndPrev         ; it by the previous value in the series
        inc     ax              ; add one to the product
        mov     rndPrev, ax     ; and save it, mod 2^16

        mov     bx, range       ; Now load the range argument,
        mul     bx              ; multiply it times the new number,
        mov     ax, dx          ; and return the high 16 bits
```

Your program now looks like this:

Now that you have finished entering the code for the routine, save the file. From the File menu, choose Save, or press SHIFT+F2. Because you have already named and saved the file once before, PWB simply saves it, without bringing up the Save As dialog box.

Note You can turn on automatic file saving by setting the **Autosave** switch to yes with the Editor Settings command on the Options menu. When **Autosave** is turned on, PWB automatically saves your file before executing certain commands such as running your program or switching to another file. For example, if you run a program that is not yet stabilized, PWB ensures that your file is stored safely in case you have to reboot.

Opening an Existing File

The remainder of this chapter uses a different file, RND.ASM, which you can now open in PWB. This file contains code to let you test the routine you just entered. It has several errors you will correct as you follow the tutorial.

To open RND.ASM:

1. From the File menu, choose Open (press ALT+F, O).

   PWB displays the Open File dialog box.

   PWB uses `*.*` as the default filename in the File Name text box. This causes PWB to display all files in the current directory in the File List box. If you know the name of the file you want to open, you can replace the `*.*` by typing the filename into the File Name text box.

2. If you are not in the directory or drive where the sample programs are located, press TAB twice to move to the Drives/Dirs box, or click inside it. The example file, RND.ASM, is located in the \SAMPLES\PWBTUTOR subdirectory of your main MASM directory, if you accepted the default directory suggested by SETUP.

   The current directory is shown directly beneath the File Name text box. Subdirectories of the current directory are listed in the Drives/Dir box, followed by the available disk drives. Although the box is only large enough to display five entries at a time, you can scroll through the subdirectories or drives to find the one you want by using the DOWN ARROW or PAGE DOWN key, or by using the scroll bar to the right of the box.

   A directory entry consisting of two periods (`..`) indicates the "parent directory" of the one you are currently in. Selecting the `..` directory causes you to move one level up your directory tree to the directory immediately

above the current directory. For example, if you are in the directory, C:\MASM\SAMPLES, then the `..` directory would be C:\MASM. Using the `..` entry helps you walk one step at a time along directory "paths."

You'll notice that the cursor is a blinking underline. That means that although you have selected the list box, you haven't yet chosen an item.

3. Use the arrow keys to move to the \SAMPLES\PWBTUTOR subdirectory of your main MASM directory.

   As you press the arrow keys, you'll notice that the cursor changes to a bar that highlights the whole selection. This is called the "selection cursor." The text of the selected item also appears in the File Name box.

4. When you have highlighted the drive or directory you want, press ENTER to move there. Using the mouse, you can simply double-click on a directory or drive entry to move to it, without having to go through an intermediate selection step.
5. Use the TAB key or mouse to move to the File List box.
6. Use the arrow keys to move to `RND.ASM`, or click on it with the left mouse button.
7. When you have highlighted `RND.ASM`, press ENTER or choose the OK button to accept your selection and open the file. Just as with the directory or drive entries, you can simply double-click on the filename to open it, bypassing the selection step.

   PWB opens RND.ASM for editing.

Copying, Pasting, and Deleting Text

The RND.ASM code contains a placeholder routine named OneOf, which returns 0. You can now delete it and replace it with the random number routine that you created in the previous section of this tutorial.

You have already typed in and saved the OneOf routine in a different file. Rather than type it over again, you can copy and paste it using PWB's clipboard (a temporary storage place for text). To do this, open the Window menu and choose the `ONEOF.ASM` window (if you no longer have it open, you will need to go back to the directory in which you saved it and open it using the Open command on the File menu).

Next, to copy the routine most conveniently, you will change the way text is selected. Three selection modes are available on the Edit menu:

- Stream mode—by default, the editor starts in stream selection mode, which allows selection to begin at any point, and selects all characters in a stream between the beginning and end positions of the cursor.
- Line mode—selects complete lines of text, starting with the entire line on which the cursor begins, and ending with the entire line on which it ends.
- Box mode—allows you to select a rectangular section of text, one corner of which is the starting position of the cursor, and the opposite corner of which is the ending position of the cursor.

The currently active selection mode is marked with a dot on the Edit menu. Clicking on a mode selects it. You can also change modes while selecting text. Just select text by clicking the left mouse button and dragging the mouse. Then, without releasing the left mouse button, press the right mouse button to toggle among the selection modes.

In this case, line selection mode is the most convenient.

To change the selection mode:

- From the Edit menu, choose Line Mode.

Next, place the cursor on the top line of text for the routine:

```
;       unsigned int OneOf ( unsigned int range )
```

To select lines of text using the keyboard:

- Press SHIFT+DOWN ARROW until the cursor is on the line containing ENDP.

To select lines of text using the mouse:

- Hold down the left mouse button and drag the cursor to the line containing ENDP.

To copy and paste the text that has been selected:

1. From the Edit menu, choose Copy. This action places the section of text that has been selected into the clipboard. You can also invoke the copy command using the shortcut key combination CTRL+INS.
2. From the Window menu, choose the `RND.ASM` window.
3. Go to the place where you want to insert the routine (line 51). Press ALT+A, type `51`, then press CTRL+M to jump to line 51.

   This sequence of keystrokes is pronounced "Arg 51 Mark." The PWB function **Arg** begins an argument (51) that is passed to the **Mark** function. When you pass a number to **Mark**, PWB moves the cursor to that line.

   You can also do this from the menu by typing the line number in the Goto Mark dialog box from the Search menu.

   The cursor is at the beginning of line 51, exactly where you want to insert the new routine.

4. From the Edit menu, choose Paste, or use the SHIFT+INS shortcut keystroke to paste the contents of the clipboard into that location.

To delete the old placeholder routine:

1. Use the PAGE DOWN key and arrow keys or mouse to move to the first line of the placeholder routine, just below the ENDP line of the inserted routine.
2. Select the six lines of the old routine, using SHIFT+DOWN ARROW or by selecting with the mouse.
3. From the Edit menu, choose Delete, or press DEL.

   The selected section is deleted.

Important If you select an area of text and then type something or otherwise insert text, PWB *replaces* the selected text (deletes it and substitutes what you are typing or inserting), without saving it on the clipboard. You can recover the text by choosing Undo at once from the Edit menu. In the example above, if you had selected the six lines of old routine before pasting in the new routine, those lines would have been deleted and replaced by the paste operation.

You have inserted the new routine into RND.ASM. Save the file by choosing Save from the File menu.

Single-Module Builds

The next step is to assemble and link the RND program to see if it works. Assembling and linking the source files is called “building the project.” It results in an executable file. A project build can also:

- Create and update the browser database.
- Create a Windows-based dynamic-link library (DLL).
- Build a library of routines.

Setting Build Options

Before you build a program, you must tell PWB what kind of file to create by using the commands on the Options menu. Use the commands from the Options menu to specify:

- The run-time support for your program. This is important for mixed-language program development, where you have some source files in assembler and some in another language. With Basic, for example, the run-time support must be Basic’s run-time support.

  The run-time support you choose determines the run-time libraries that are used and the types of target environments that can be supported.

- Project template. The template describes in detail how PWB is to build a project for a specific type of file (.EXE, .COM, .DLL, .LIB) and the operating environment for the target file (MS-DOS, the Windows operating system, and so on).
- Either a debug or release build. Debug options normally specify the inclusion of CodeView debugging information, where release options do not. You may want to generate a different listing file for a debug build than for a release build, or you may not want any listing file for one type of build or the other.
- A build directory. PWB builds your object and executable files in your current directory unless you specify otherwise. (This option is reserved for projects that use explicit project files, which are described in Chapter 3.)

To set the project template for RND.ASM:

1. From the Options menu, choose Set Project Template from the Project Templates cascaded menu.

Note that the actual order of the menu items may differ from the illustration because PWB's extensions can be loaded in any order.

2. PWB displays the Set Project Template dialog box.

This dialog box typically has the entries `None` and `Assembler` in the Runtime Support list box. If you have installed other languages, their names appear as well.

Since the RND program does not require run-time support, leave None selected.

3. Move to the Project Templates list box by clicking in the box, pressing the TAB key the appropriate number of times, or by pressing ALT+T.
4. Select `DOS EXE`.
5. Choose the OK button to set the new project template.

To set the build options for RND.ASM:

1. From the Options menu, choose Build Options.

   PWB displays the Build Options dialog box.

2. Turn on Use Debug Options by choosing the Option button or by pressing ALT+D.

   This option tells PWB that you are building a debugging version of the program. PWB uses debug options when you build or rebuild until you use the Build Options dialog box to choose Use Release Options.

3. Choose the OK button.
4. From the Options menu, choose Languages Options, then choose MASM options from the secondary menu.

   PWB displays the Macro Assembler Global Options dialog box.

5. Choose Set Debug Options.

   PWB displays the Macro Assembler Debug Options dialog box.

   In the Debug Information box, CodeView should already be selected, indicating that the assembler will generate the information that CodeView needs to correlate assembled code with source code.

6. Select Generate Listing File and Include Instruction Timings. This causes the assembler to create a listing file showing you exactly how it assembled your program, and to include in the listing how many clock cycles each instruction will take to execute.
7. Choose the OK button twice.

PWB saves all the options that you specify. You don't have to respecify them each time you work on your project.

The following illustration shows the three sets of options that PWB maintains for each project. Global options are used for every build. Debug options are

used when Use Debug Options is turned on in the Build Options dialog box. Release options are used when Use Release Options is turned on.

You can set assembler and linker options for both types of builds (debug and release) by using the Language Options commands and the LINK Options command. The Build Options command then determines which type of build, using which set of options, is actually performed when you assemble a file or rebuild the project.

Global options, on the other hand, typically include settings for warning level, memory model, and language variant. These are options that do not change between debug and release versions of a project.

Setting Other Options

The Options menu also contains commands that allow you to describe the desired project build more completely. You don't need to change most of these options to build RND.ASM because the default values supplied by the template will work well.

The Options menu contains the following commands:

- MASM Options in the Language Options cascaded menu. These commands let you specify assembler options specific to debug and release builds, and general options common to both types of builds. Using the MASM Global Options dialog box, you can specify memory model, warning level, and so on.

  If you have more languages installed, their Compiler Options commands also appear in the Languages Options cascaded menu.

- LINK Options. This command parallels the Compiler Options commands. You can specify options specific to debug or release builds and general options common to both debug and release builds.

  Use LINK Options to specify items such as stack size and additional libraries. You can also select different libraries for debug and release builds. This is handy if you have special libraries for debugging and fast libraries for release builds.

- NMAKE Options. This command lets you specify NMAKE command-line options for all builds. This option is particularly useful if you have an existing makefile that was not created by PWB or if you have modified your PWB project makefile. For more information about these subjects, see “Using a Non-PWB Makefile” on page 55.
- CodeView Options. This command allows you to set options for the CodeView debugger.

Building the Program

Now that you’ve set your options, you can build the program. Note that the sample program contains intentional errors that you will correct.

To start the project build:

1. From the Project menu, choose Build.

   PWB tells you that your build options have changed and asks if you want to `Rebuild All`.

2. Choose Yes to rebuild your entire project.

After the build is completed, PWB displays the following dialog box:

You can choose one of several actions in this dialog box:

- View the complete results of the build by opening the Build Results window.
- Run the program if building in MS-DOS. You can run an MS-DOS program right away if the build succeeds. If the build fails, you should fix the errors before you attempt to run the program.

To run a successfully built Windows-based program, you must be running under the Windows operating system, and have started the WXServer program *before* you start PWB.

- Debug the program if building in MS-DOS. If the build succeeds but you already know the program is not producing the intended results, you can debug your MS-DOS program using CodeView.

  To debug a Windows-based program, you should be running under the Windows operating system, and already have the WXServer running when you start PWB or CodeView.

- Get Help by choosing the Help button or by pressing F1 (as in every PWB dialog box).
- Cancel the dialog box. This returns you to normal editing.

Choose View Results to close the dialog box (press ENTER). PWB displays the results of the build so that you can review the build messages or step through them to view the location of each error. The next section describes how to do this.

Fixing Build Errors

For each build, PWB keeps a complete list of build errors and messages in the Build Results window. The RND.ASM program that you just built contains several errors that you'll identify and fix in this section.

If you want to examine build errors in a specific order, you can do so in the Build Results window by placing the cursor on whatever error you wish to examine, and selecting Goto Error from the Project menu. PWB opens a window onto the appropriate source file and places the cursor on the line at which that error was recognized. When you are finished with each error, selecting the Build Results window from the Window menu will return you to the Build Results window.

In many cases, however, you will want to work through the errors one after another. This is the easiest method for fixing the build errors in RND.ASM.

To fix errors one after another:

1. From the Project menu, choose Next Error, or press SHIFT+F3.

   PWB positions the cursor on the location of the first error or warning in your program. In this case, a comma is missing after the 10 at the end of the first line of the `banr2` data declaration.

2. Correct the first error by inserting a comma immediately after the 10.
3. From the Project menu, choose Next Error, or press SHIFT+F3.

   PWB moves the cursor to the location of the second error. Here, `"Esc"` in the string on the line below the cursor is enclosed in double quotes, and the string itself is also enclosed in double quotes. As a result, the assembler interprets the first set of quotes around `Esc` as the end of the string, and then does not recognize `Esc` as a valid instruction or directive. This can be fixed by substituting a pair of single quotes for the pair of double quotes either around the string or around `Esc`.

4. Fix the error by changing the double quotes (`""`) around `Esc` to single quotes (`''`).

   Because of this error, the data symbol **again** was not defined during the first assembly pass, which also meant that the constant **lagain** could not be evaluated. As a result, two more errors were generated, which can now be ignored.

5. From the Project menu, choose Next Error, or press SHIFT+F3.

   PWB positions the cursor on the location of the third error, a simple typographical error where the **mov** instruction was spelled “mob.”

6. Correct the third error by replacing the “b” in mob with a “v.”

Now that all the build errors in RND.ASM have been corrected, save the file by choosing Save from the File menu or by pressing SHIFT+F2.

Running the Program

The next step is to build and run the program.

To run the program:

1. From the Run menu, choose Execute (be sure that you have saved RND.ASM first).

   PWB detects that you've changed the source and displays a dialog box with the following options:

2. Choose Build Target to build the program.

   When the build completes, PWB displays the following dialog box:

3. Choose Run Program to run the finished program.

When you run it, the RND program will start by asking you to supply a range value between 1 and 65,535. Type `1234` and press ENTER. The program will then ask you to confirm that 1,234 is indeed the correct range. When you type `y`, the program is supposed to display a list of random numbers within that range. Instead, however, the program restarts when you type `y`. Something is wrong.

To get out of the program and back to PWB, press CTRL+C (in the case of this particular program, you can also use the ESC key to exit when the program asks for confirmation of a range value). Before blanking your program's output, PWB will display the message, "Strike any key to continue..." so that you can examine the final state of the screen.

The following sections describe the process of debugging using the Microsoft CodeView debugger. If you're already familiar with CodeView, skip to Chapter 3, "Managing a Multimodule Program."

Debugging the Program

PWB integrates several Microsoft tools to produce a complete development environment. Among those tools are NMAKE, a program maintenance utility, and CodeView, a symbolic debugger. Whenever you build programs using PWB, PWB in turn invokes NMAKE to manage the build process. In the same way, PWB can serve as a gateway to CodeView when you need to debug a program you have built.

Earlier, you chose Use Debug Options in the Build Options dialog box. A debug build typically includes the assembler options that generate CodeView information. Therefore, the program is ready to debug with the CodeView debugger.

Using CodeView to Isolate an Error

In addition to the typographical errors that you just corrected, RND.ASM contains a logical error which will prevent it from running properly. You can use CodeView to isolate this error.

To start CodeView:

- From the Run menu, choose Debug.

  If anything in your program is out of date, PWB asks if you want to build or rebuild the current target. If you modified the source file in any way, PWB considers it out of date relative to the executable file that you built earlier. If this happens, build the program and choose Debug from the Run menu.

CodeView now starts, displaying three windows on its main debugging screen.

The first thing to do is set up the CodeView screen so that it best suits your way of working. When you leave CodeView, your setup will be saved in CURRENT.STS. The next time you use CodeView, that setup will be restored when the program starts.

The right screen layout depends a lot on your work style, and on the project you are working on. In this case, many of CodeView's more advanced features will not be necessary, so we will set up a simple screen.

By default, three windows are initially displayed: "locals," "source1," and "command." Close the locals window, since it will not be needed in debugging RND, open a register window and a memory window, and arrange the windows in the screen.

To close a window using the mouse:

- Click the upper left corner of the window.

To close a window using the keyboard:

- Use the F6 key to move into the window that you want to close. Choose Close from the Windows menu, or press CTRL+F4.

To open the Register and Memory windows:

1. From the Windows menu, choose Register, or press ALT+7.

   The Register window displays the contents of the processor's registers, either in "Native" (8086) mode, or in "32-bit" (80386-80486) mode.

2. At the bottom of the Options menu, click Native if it is not already selected.
3. Choose Memory 1 from the Windows menu, or press ALT+F5.

   Memory windows display the contents of a specified block of memory, so that you can watch changes as your program runs.

To move and size a window using the mouse:

1. To move a window, place the cursor on its top line, not in a corner. Then drag the window to a new location.
2. To size a window, move the cursor to the lower right corner of the window. Then drag the corner to change the window's size.

To move and size a window using the keyboard:

1. Using the F6 key, shift focus to the window you want to size.
2. Choose Move or Size from the Windows menu.
3. Use the arrow keys to move or size the window.
4. Press ENTER when you are finished.

When you have positioned and sized the windows to your satisfaction, set the source window to show both your source text and the actual instructions assembled by MASM, and set the memory window to stay fully up to date as the program executes.

To display mixed source and assembler output:

1. From the Options menu, choose Source1 Window.

   CodeView displays the Source1 Window Options dialog box.
2. In the Display box, choose Mixed Source and Assembly.
3. Choose OK.

To set the Memory1 window to be updated frequently:

1. From the Options menu, choose Memory1 Window.

   CodeView displays the Memory1 Window Options dialog box.
2. Select the “Re-evaluate expression always (live)” check box.
3. Choose the OK button.

Working Through a Program to Debug it

CodeView has placed you at the program’s starting point. The registers are as they would be at that point, and the memory window shows whatever the DS register is pointing to. The instructions that appear at the top of the source window have been created by the .STARTUP directive, as you can see if you scroll up a few lines.

CodeView provides various ways to control and examine the execution of a program. The “Step” command (F10 key) executes the next instruction in the program, and if that instruction is a call, executes the entire called code up through the return. “Trace” (F8 key), on the other hand, jumps to the called code and traces through it too, one instruction at a time. You can also run the program up to a given point, or set breakpoints at several points. With RND, we will only need to use a few of the possible debugging tools.

To Step through the program:

- Use the F10 key to step through the first couple of instructions of the .STARTUP code.

You will notice that as each instruction is executed, CodeView briefly displays the program output screen, and updates the Register window to show changes in the registers. As the DS register is loaded, the Memory window displays the data segment of the RND program.

Stepping is a slow way to move through the program. In many cases, as with RND, you will want to move quickly to the point where the program failed, to see what the matter was. In RND, everything seemed to be working correctly until you entered `y` to confirm the range.

To run a program up to a given place:

1. Scroll through the code to the comment line:

   ```
   ;       Read in a character from the keyboard
   ```

   Three lines below the comment is a **cmp** instruction.

2. Place the cursor on the line containing the **cmp** instruction, either by using the arrow keys or the mouse.

3. Press the F7 key, or by clicking on that line with the *right* mouse button.

CodeView procedes to execute the program up to (but not including) that line. The display switches to the output screen where the program shows its introductory message, then requests a range value.

To work through the RND program and find the bug:

1. Type in a range value smaller than 65,535 and press ENTER.

   The program redisplays the range value and asks for confirmation.

2. Press `y`.

   CodeView returns you to the source window in the debugging screen.

   The succeeding instructions are designed to recognize an ESC or a `y`, and are presumably failing in some way, causing the program to start from the beginning.

3. Using the F10 key, step through the various **cmp** instructions.

   You will find that the code works as expected, recognizes the `y`, and proceeds.

4. Go on to the next the next jump or branch.

   The next possible branch in program execution occurs at the call to OneOf. Although this seems unlikely to be causing the program to start over, it is the next thing to test.

5. Position the cursor on the call instruction, and press either F7 or the right mouse button to execute the program up to that call.

   So far, so good: the program continues to run as expected.

6. Now press F10 to execute the call itself.

The program now erroneously starts over. We now know that the problem must be located in the OneOf routine.

7. Press CTRL+C, then ENTER to get out of the program.
8. Choose Restart from the Run menu to return to the beginning of the program.
9. As you did before, scroll down to where OneOf is called and execute the program up to just before the call.
10. This time, use F8 to trace through the call.

    You will notice that CodeView now shifts into the called routine, allowing you to step through the OneOf code instruction by instruction.

11. Step or trace through the OneOf routine, using F10 or F8, and look for the problem.

    You will discover a simple error of omission: the routine has no **ret** instruction at the end. As a result, execution continues into the succeeding code, which happens to be the .STARTUP code.

    Having found the problem, you can leave CodeView and return to PWB.

12. From the File menu, select Exit.

    CodeView closes, saving your settings for next session.

13. From PWB, insert a new line in the RND.ASM file just before the ENDP line of the OneOf routine.

14. Type a **ret** instruction there.

15. Save the corrected file by choosing Save from the File menu, or by pressing SHIFT+F2.
16. Select Execute from the Run menu to rebuild the program and try it again.

This time, it should work without problems.

Examining Memory in the Memory Window

In addition to being able to watch the register contents change as your code runs, CodeView lets you see what happens to locations in memory. For example, you may have noticed that `OFFSET lnBuf` was assembled as hex `E4`. By setting the memory window at that address, you can watch what happens in the lnBuf buffer as the program formats a line of output. One way to reach that address, since it is fairly close, is to scroll in the memory window until you get to it. However, this is often impractical.

To set the address in the Memory Window:

1. From the Options menu, select Memory1 Window.
2. In place of `DS:0` in the Address Expression field of the Memory1 Window Options dialog box, type `DS:0x00e4`.

Now, you can step through cycles of a formatting loop and watch the buffer change.

To step through a formatting loop in RND.EXE:

1. In the source window, scroll to the instruction `dec bl` around line 150, which completes a formatting cycle for a random number.
2. Press F7 or click in that line with the right mouse button.

   If you know for sure that `dec bl` is on line 150, you can move to the Command window and type `g @150` followed by the ENTER key. This instructs CodeView to execute the program up through line 150 in the source file.

3. While watching the memory window, press F7 again, or click the `dec bl` instruction again with the right mouse button.

   As the loop executes again, you can see the memory area change to reflect the new value being formatted into lnBuf.

To switch from CodeView back to PWB:

- Choose Exit from the CodeView File menu.

Where to Go from Here

Now that you've created, built, and debugged a simple program, you've begun to discover the power of PWB. Chapter 3, "Managing Multimodule Programs," describes how to create and manage projects with more than one source file.

CHAPTER 3

Managing Multimodule Programs

This chapter expands on the work you did in Chapter 2 and explains how to build and maintain multimodule programs using PWB's integrated project-management facilities. PWB offers an efficient way to manage complex projects. You organize and build your project entirely within PWB, using convenient menus and dialog boxes instead of makefiles or batch files.

PWB stores the information needed to build and manage your program in two files, the project makefile and the project status file. These are called the "project." When you open the project, PWB automatically configures itself to build your program. To move from one project to another, you close one project and open another.

Multimodule Program Example

In this chapter, you'll learn to set up a multimodule project in PWB by building SHOW.EXE, a three-module program. The SHOW program displays text files on character-based screens with MS-DOS.

The following modules make up SHOW.EXE:

Module	Function
SHOW.ASM	Program driver; contains .STARTUP entry point, and calls all other procedures.
PAGER.ASM	Contains procedures for paging through a file and writing text to the screen buffer
SHOWUTIL.ASM	Contains miscellaneous procedures.

The program also contains a common header file SHOW.INC in addition to these three source modules. Figure 3.1 shows the components of SHOW and how they combine to build the executable file.

Figure 3.1 The SHOW Project

To build SHOW.EXE, you need to assemble the three source files and link them together, having specified the assembler and link options that will produce the kind of file you are trying to make. All this build information is stored in the SHOW project make and status files.

Opening the Project

Start by opening the SHOW project. (If you have not started PWB, do so now.)

To create a project:

1. From the Project menu, choose New Project.

   PWB displays the New Project dialog box.

2. Type `show` in the Project Name text box.
3. Choose Set Project Template.

   PWB displays the Set Project Template dialog box.

4. Select the following options:

- Runtime Support: `None`.
- Project Template: `DOS EXE`.

At this point, the Set Project Template dialog box should appear as follows:

This initial specification tells PWB what kind of executable file you intend to build, and is saved as part of the project.

5. Choose OK to return to the New Project dialog box.

   In this case, a project makefile, SHOW.MAK, already exists. Since PWB would ordinarily create and save a new makefile at this point, you are now asked whether you want to overwrite the existing file.

6. Choose Yes to overwrite the existing file.

   PWB saves the new SHOW.MAK and returns to the New Project dialog box.

7. Choose OK.

   PWB now displays the Edit Project dialog box so that you can add files to your new project.

The next section describes the types of files that can be added to the project. The tutorial then continues by listing the example files to add to the list.

Contents of a Project

A project file list can contain the following files:

- Source code files (.ASM).
- Object files (.OBJ) in special cases.
- Library files (.LIB) for libraries that change.
- Module-definition files (.DEF) for DLLs.
- Resource-assembler source files (.RC) for Microsoft Windows-based programs.

These file types are all that are needed to create most MS-DOS and Windows-based applications. Include files, such as SHOW.INC, need not be listed because PWB automatically adds them when it scans your source files for dependencies.

When you select assembler run-time support with a Windows-based project template in the Set Project Template dialog box, PWB automatically specifies standard library files such as LIBW.LIB. Therefore, you need not add standard library files to the project list.

To add the SHOW files to your project:

1. Choose the files you want to add to the project from the File List box. In this case, you'll add SHOW.ASM, PAGER.ASM, and SHOWUTIL.ASM. These files are located in the \MASM\SAMPLES\SHOW directory. If you installed Microsoft MASM 6.1 in a directory other than MASM, adjust the path accordingly.

You can scroll the File List box by clicking the scroll bars or by pressing the arrow keys.

2. For each file, select it and choose Add / Delete to add the file to the Project list box. Or, you can double-click a file to add or remove it from the list. To add all three files at once, you can type `*.ASM` in the File Name field, press ENTER, and then choose Add All.
3. Choose Save List when you have added all three files.

   PWB uses the rules in the project template along with the list of files that you just specified to scan the sources for include dependencies and to create the project makefile. This process is described in the next section.

Now your project completely describes what you want to build (the project template), the component source files, and the commands used to build the project.

Dependencies in a Project

When you save the project, PWB generates a makefile from the project template, files, and options you specified. This file also contains a list of instructions that are interpreted by NMAKE. In addition, PWB generates the project status file, which saves the project template, the editor state, and the build environment for the project. For more information on the project status file, see “Project Status Files” on page 129.

When you build the project, NMAKE examines the build rules in the project makefile. These are rules that specify targets (such as an object or an executable

file) and the commands required to build them. For example, a rule for making an .OBJ file from an .ASM file can be expressed as follows:

```
.asm.obj:
       ML /c $<
```

To reduce the amount of time builds take, NMAKE assembles or links only the targets that are out-of-date with respect to their corresponding source file. This process is simple if there is a one-to-one correspondence between sources and targets. However, many programs use the INCLUDE directive to include files containing common equates, macros, and other program text. The object files must be made dependent not only on the source file but also on the files that are used by the source file.

You don't need to add include (.INC) files to your project. When you save the project, PWB scans your source files looking for INCLUDE directives and builds dependencies on these files. NMAKE will thereafter recompile a source file if you change a file that it includes.

Building a Multimodule Program

Now that the project files are complete, you can build the program in the same way you built the single-module program.

To build a multimodule program:

1. You are starting a new project, so you should use debug options for the initial builds. Choose the Use Debug Options button in the Build Options dialog box.
2. From the Project menu, choose Build.

   PWB displays a dialog box to inform you that build information has changed because you altered the build options.
3. Choose Yes to rebuild your entire project.

As the program is built, PWB shows status messages about the progress of the build. When the build completes, a dialog box displays a summary of any errors encountered during the build process.

Note The Next Error command on the Project menu works the same for a multimodule build as for a single-module build. Because errors in a multimodule build can occur in different files, PWB automatically switches to the file that contains the error.

In some cases, you will want to force a complete rebuild of your project by choosing Rebuild All from the Project menu. The difference between Build and

Rebuild All is that Build compiles and links only out-of-date targets and Rebuild All compiles all targets, regardless of whether they are current.

Running the Program

Now that your program is built, you can test it from PWB.

To run SHOW:

1. From the Run menu, choose Program Arguments.
2. Type the name of a text file to pass to the SHOW program. The SHOW.ASM source file is a good file to use.
3. Choose OK to set the program arguments. PWB saves the arguments so that you can run or debug the program many times with the same command line.
4. From the Run menu, choose Execute.

SHOW will display the first screen of text in the file you passed to it. You can use the arrow keys and PAGE UP and PAGE DOWN to move around in the text file.

Press `Q` and then any key to return to PWB.

You have successfully created a multimodule project, built the program, and run it, all from within the Programmer's WorkBench. You can now leave PWB.

To leave PWB:

- From the File menu, choose Exit or press ALT+F4.

  PWB saves your project and returns to the operating-system prompt. If you started PWB from within the Windows operating system, you will return to the Windows operating system.

Creating a PWB project is an important first step. However, most of the time you will be maintaining projects. The next section provides an overview of project maintenance. The tutorial then continues with the SHOW project.

Project Maintenance

Once you have created a project, you may have to change it to reflect the changes in your project organization. You can:

- Add new file-inclusion directives to your source files.
- Add new source, object, or library files.
- Delete obsolete files.
- Move modules within the list.

- Change assembler and linker options.
- Change options for individual modules.

When you add a new INCLUDE directive to a source file, you add a new dependency between files. For the most accurate builds, you need to regenerate include dependencies for the project.

To regenerate include dependencies:

1. From the Project menu, choose Edit Project.
2. Select the Set Include Dependencies check box.
3. Choose Save List.

   PWB regenerates the include dependencies for the entire project and rewrites the project makefile.

To add new files to an existing project:

1. From the Project menu, choose Edit Project.
2. For each file that you want to add to the project:
   - Select the file from the File List box, or type the name of the file in the File Name text box.
   - Choose the Add / Delete button to add the file.
3. Choose Save List to rewrite the project makefile, set up the dependencies, and add the commands for the new files.

To delete files from a project:

1. From the Project menu, choose Edit Project.
2. For each file that you want to remove from the project:
   - Select the file from the File List box, or type the name of the file in the File Name text box.
   - Choose the Add / Delete button to remove the file from the list.
3. Choose Save List.

With most programming languages, you won't need to move modules within a project. However, some languages or custom projects require files to be in a specific order. If you're programming in Basic, for example, you must place the main module of your program at the top of the list. Unlike other languages, Basic does not define an explicit name where execution begins. Entry to a Basic program is defined by the first file in the list.

To move a file to the top of the project file list:

1. From the Project menu, choose Edit Project.
2. Select the file you want to move to the top of the list.
3. Choose the To Top of List button.

Using Existing Projects

You'll now make modifications to the SHOW project that you just created. During a PWB session, the project you open remains open unless you explicitly change it. If you have not already started PWB, you should do so now. In the Windows operating system, double-click the PWB icon in the MASM program group.

If you are not compiling from within the Windows operating system, you can start PWB and open the SHOW project from the operating-system command line by typing the command:

```
PWB /PP SHOW
```

If the SHOW project is the last project you had open in PWB, type the following command:

```
PWB /PL
```

You can set up PWB to reopen the last project automatically at startup by choosing Editor Settings from the Options menu, and then by setting the Boolean switch Lastproject to Yes.

If you have already started PWB, open the project now.

To open the project from within PWB:

1. From the Project menu, choose Open Project.
2. Choose SHOW.MAK from the File List box or type `show` in the Project Name text box.

3. Choose OK.

When you open the project, PWB restores the project's environment, including:

- The window layout with the window style, size, and position for each window.
- The file history—a list of open files for each window and the last cursor position in each file.
- The last find string.
- The last replace string.
- The options that you used for the last find or find-and-replace operation, such as regular expressions. See "Using Regular Expressions" on page 82 for more information about regular expressions.
- The project template (for example, `DOS EXE`) and any customizations you have made to the template such as changing the build type or an assembler or linker option.
- The command-line arguments for your program.

Note PWB can save all environment variables, including PATH, INCLUDE, LIB, and HELPFILES, depending on how the **envcursave** and **envprojsave** switches are set. For more information, see “Environment Variables” on page 127.

Also, if you turn the **restorelayout** switch off, PWB does not restore the window layout, the find strings and options, or the file history of a project. Instead, PWB keeps the current editor state when opening a project.

Adding and Deleting a Project File

As you develop a project, you will occasionally add new modules. The following example presents the steps needed to add a library file to the SHOW project. Note that this procedure is only an example, and in fact, SHOW does not use or require any library support.

To add a file to your project:

1. From the Project menu, choose Edit Project.

   The file and directory navigation lists in this dialog box work in exactly the same way as those in the Open File dialog box.

2. Choose the parent directory symbol (`..`) in the Drives / Dirs list box to move up the directory tree to the SAMPLES directory.
3. Choose the parent directory symbol (`..`) again to move up the directory tree to the MASM directory.
4. Choose the LIB directory in the Drives / Dirs list box to move down the tree into the LIB directory.

Notice that the directory displayed after the label `File List` reflects the directory change.

5. Make sure the File Name text box contains `*.*` or `*.LIB`.
6. Select LIBW.LIB in the File List box.
7. Choose the Add / Delete button to add the file to the project.

   LIBW.LIB is being used here as an example of how to add a file to your project. In practice, because it is a system library that will not change, there is no reason to add it. However, if you have a library of your own that is being used by your project, you would add it to the project in this way.

8. Since LIBW.LIB is not a source file and cannot have include dependencies, you can clear the Set Include Dependencies check box. If this check box is selected, PWB regenerates the dependencies for all the files in the project.
9. Choose Save List.

LIBW.LIB is now part of the project. Since SHOW is not a program designed to run under Microsoft Windows, you should now delete this library from the project again.

To delete a file from your project:

1. From the Project menu, choose Edit Project.
2. In the Edit Project dialog box, you can either select LIBW.LIB in the Project list box and then select Add / Delete, or simply double-click on LIBW.LIB in the Project list box to delete it.

Changing Assembler and Linker Options

Up to this point, you have used PWB's default build options for all the examples. These options are sufficient for most cases, but in special cases, you will want to adjust them.

When you are debugging a program, you should choose the debug build type. When producing a debug build, the assembler and linker include a good deal of extra information in the program for CodeView to use in debugging. When you are ready to use the program, choose the release build type, so that the extra debugging information is no longer incorporated into the program.

To specify whether a build should be for release or debug:

1. From the Options menu, choose Build Options.
2. Choose Use Debug Options or Use Release Options in the Build Options dialog box.
3. Choose OK.

When you specify a release build, PWB does not change your debug options. For more information on global options, debug options, and release options, see "Setting Build Options" on page 18.

To change assembler options:

1. From the Language Options cascaded menu on the Options menu, choose MASM Options.

   The Macro Assembler Global Options dialog box contains a number of options that are common to both the release and debug builds.

At the bottom of the dialog box are buttons that set options that are specific to the current type of build (debug or release), and that show the assembler flags corresponding to those options. Default settings were determined when you chose the project template.

Note You can choose the Set Debug Options button to view and set the options for debug builds. However, this does not change the type of build that is performed when you build the project. To set the type of build, choose Build Options from the Options menu.

---

2. Choose Set Debug Options.

   PWB displays a dialog box in which you can specify debug options.

   If you had chosen Set Release Options, PWB would have displayed the same dialog box, so that you could select options for release builds.

3. Choose OK to return to the Macro Assembler Global Options dialog box.
4. Choose OK to save the new assembly options and return to the main PWB screen.

To change the linker options:

1. From the Options menu, choose LINK Options.

   PWB displays the LINK Options dialog box.

2. Choose Additional Global Options to review and select additional global link options.

   PWB displays the Additional Global Link Options dialog box.

3. Choose OK when you are finished to return to the LINK Options dialog box.
4. Choose Additional Debug Options to review and select additional debug link options.

   PWB displays the Additional Debug Options dialog box.

5. Choose OK when you are finished to return to the LINK Options dialog box.
6. Choose OK to close the LINK Options dialog box and use any new options you might have set.

You are now ready to build your project with any new options you have selected.

To build a modified project:

- From the Project menu, choose Rebuild All.

Changing Options for Individual Modules

Most of the modules in a program can generally be built using the same options. However, you may occasionally want to modify the options for a single module.

The example that follows shows how to customize your project to change the assembler options for PAGER.ASM only. To do this, you manually edit the instructions in the project makefile for compiling PAGER.ASM.

To open SHOW.MAK for editing:

1. If the SHOW project is open, choose Close Project from the Project menu.

   This step is important because you cannot edit a PWB makefile for a project that is currently open.

2. Choose the Open command from the File menu and open the SHOW.MAK file in the editor.

To customize the assembly of PAGER.ASM:

1. Find the rule for compiling PAGER.ASM:

```
PAGER.obj : PAGER.ASM show.inc
!IF $(DEBUG)
    $(ASM) /c $(AFLAGS_G) $(AFLAGS_D) /FoPAGER.obj PAGER.ASM
!ELSE
    $(ASM) /c $(AFLAGS_G) $(AFLAGS_R) /FoPAGER.obj PAGER.ASM
!ENDIF
```

This rule contains a conditional statement with two commands. The first command is for debug builds, and the second command is for release builds. You can edit either one of these commands, or both. They contain the following macros defined earlier in the makefile:

Macro	Definition
ASM	The name of the MASM assembler
AFLAGS_G	Global options for assembly
AFLAGS_D	Debug options for assembly
AFLAGS_R	Release options for assembly

As an example, suppose that PAGER.ASM contained data structures which you want to pack on 32-bit boundaries for the release build only. The `/Zp4` flag tells the ML program to pack data structures on 4-byte boundaries.

2. Edit the release build command as follows.

```
$(ASM) /c $(AFLAGS_G) $(AFLAGS_R) /Zp4 /FoSHOWUTIL.obj SHOWUTIL.ASM
```

Because it is hard to be sure what options the flags macros will invoke, the new option should be placed *after* them, so that it will supersede any instructions they may contain.

Note that both the assembler options, such as `/Zp`, and NMAKE macros, such as `AFLAGS_G`, are case sensitive and must appear exactly as shown.

Warning After this modification, PWB still recognizes this makefile as a PWB makefile. However, if you make changes beyond adding options to individual command lines, PWB may no longer recognize the file as a PWB makefile. If this happens, you can delete the makefile and re-create it, or you can use it as a non-PWB makefile. For more information on using non-PWB makefiles, see "Using a Non-PWB Makefile" on page 55.

You could save your changes to the makefile by choosing Save from the File menu, then reopen the project and rebuild SHOW with the custom option you just installed. Because PAGER.ASM does not contain any data declarations, however, the new options have no real purpose or effect.

The Program Build Process

This section explains the correspondence between projects and makefiles. Normally, the build process is automatic, but you may encounter situations that require customized build options. Read this section to learn how the utilities work with PWB. The following diagram illustrates the PWB build process.

Figure 3.2 The PWB Build Process

When you save your project by choosing the Save button in the Edit Project dialog box, PWB uses the list of files along with the rules in the selected project template to scan for dependencies and write the project makefile.

When you choose the Build or Rebuild All command from the Project menu, PWB releases as much memory as possible and passes the makefile to NMAKE, which builds the project.

NMAKE stops at the end of the first build step that produces an error (as opposed to a warning) or at the end of a successful build. In either case, NMAKE returns the results of the build to PWB along with a log of any errors and warnings. For more information about NMAKE, see Chapter 16, "Managing Projects with NMAKE."

PWB saves the output of the build for you to view in the Build Results window or to step through when you choose the Next Error (SHIFT+F3), Previous Error (SHIFT+F4), and Goto Error commands on the Project menu. You can run the program, set program arguments, and debug the program by choosing commands in the Run menu.

If you have turned on the generation of browser information, PWB builds the browser database when you build the program. Once you have a browser database, you can use the commands in the Browse menu to navigate your program's source files and examine the structure of your program. For more information, see "Using the Source Browser" on page 88.

Extending a PWB Project

Makefiles that are not written by PWB often contain utility targets that are not used in the process of building the project itself. These targets are used to clean up intermediate files, perform backups, process documentation, or automate other tasks related to the project. You can extend a PWB makefile to perform these kinds of tasks by adding new rules. These additional rules must be placed in a special section of the project makefile.

In the following example you will add a section that creates a file with information about the project. This file has the same base name as the project and the extension .LST. It lists the files in the project and the major options used for the build. This example section can be used with any assembly-language PWB project.

Use the SHOW project to see how to add a custom section. If you have been following the tutorial, you have already made one custom edit to the SHOW.MAK file.

To add a custom section to the PWB makefile:

1. If the project is open, choose Close Project from the Project menu.

   This step is crucial because PWB disables modification of the project makefile until the project is closed or a different project is opened. (This restriction does not apply to non-PWB project makefiles.)

2. From the File menu, choose the Open command and open the SHOW.MAK file in the editor.
3. Press CTRL+END to move the cursor to the end of the makefile.
4. Type the following new comment line *exactly* as shown:

   ```
   # << User_supplied_information >>
   ```

   You must put the number sign (#) in column one and type the contents of the line exactly as shown, including capitalization. Failing to type this line accurately will make the project unrecognizable to PWB or will cause PWB to change your custom build information in unexpected ways.

   You can copy this line from Help rather than typing it in, if you wish. Press ALT+A, type `USI`, press F1, and then copy the line. Move back to the make file, and paste the line in at the end.

   NMAKE requires space between rules. Therefore, you should separate this line from the lines above it by one blank line. Similarly, you should leave at least one line between the separator and your custom build rules. For more information about NMAKE and the syntax of makefiles, see Chapter 16, "Managing Projects with NMAKE."

   This comment line is used by PWB as a separator. Anything above this comment is regarded as belonging to PWB, and you should not edit the information there. The exception is to add options to individual command lines, as described in "Changing Options for Individual Modules" on page 49. Anything in the makefile after the separator is your information, and PWB ignores it. NMAKE, however, processes the entire file.

Now that you have a separator to show PWB where your custom information starts, you can add the custom information. The separator and custom section is included in the following text, and can also be found in the EXTRA.TXT file in the SAMPLES directory:

```
# << User_supplied_information >>

# Example 'user section' for PWB project makefiles,
# used in the PWB Tutorial.
#
# NOTE: This is not a standalone makefile.
#       Append this file to makefiles created by PWB.
#
# This user section adds a new target to build a project
# listing that shows the build type, options, and a list
# of files in the project.
#

!IFNDEF PROJ
!ERROR Not a standalone makefile.
!ENDIF
!IF $(DEBUG)
BUILD_TYPE = debug
!ELSE
BUILD_TYPE = release
!ENDIF

# Project files and information-list target
#
$(PROJ).bld : $(PROJFILE)
    @echo <<$(PROJ).bld : Project Build Information
Build Type:        $(BUILD_TYPE)
Program Arguments: $(RUNFLAGS)
Project Files
    $(FILES: =^
    )
Assembler Options
    Global:  $(AFLAGS_G)
    Debug:   $(AFLAGS_D)
    Release: $(AFLAGS_R)
Link Options
    Global:  $(LFLAGS_G)
    Debug:   $(LFLAGS_D)
    Release: $(LFLAGS_R)
<<KEEP
```

The custom section of a PWB makefile can use any of the information defined by PWB. This example takes advantage of many macros defined by PWB. For example, the PROJFILE macro, which contains the name of the project makefile, is used as the dependent of the listing file so that the listing is rebuilt whenever the project makefile changes.

In addition, this custom section uses many features of NMAKE, including macros, macro substitution, preprocessing directives, and inline files. For more

information about NMAKE and makefiles, see Chapter 16, “Managing Projects with NMAKE.”

To rebuild using the custom options:

1. Choose Open Project from the Project menu and reopen the SHOW project.
2. From the Project menu, choose Build Target.
3. Type the name of the new target `SHOW.BLD` in the Target text box, and then choose OK.

   PWB informs you that the build options have changed and asks if you want to rebuild everything.
4. Choose Yes to confirm that you want to rebuild everything.

The project information file that is created shows the project name, indicates whether the build is a debug or release build, lists the files in the project, and lists the assembler and linker options used for the build.

Using a Non-PWB Makefile

PWB makefiles are highly structured and stylized makefiles that are generated from the rules in the project template and a list of files that you supply. Many projects have existing makefiles that PWB can’t read because they do not have this stylized structure. These makefiles are called non-PWB or “foreign” makefiles.

You can still take advantage of many of PWB’s project features with non-PWB makefiles. The features that cannot be used are shown as unavailable menu items. Note that a PWB makefile is not required to use the Source Browser—all you need to have is a browser database. For information on building a browser database, see “Building Databases for Non-PWB Projects” on page 94.

To use a non-PWB make file:

1. From the Project menu, choose Open Project.
2. Select the non-PWB make file to open.
3. Select the Use as a Non-PWB Makefile check box.

   The Open Project dialog box appears.
4. Choose OK.

Note A PWB makefile cannot be edited or modified when it is the open project. However, PWB does not disable modification of non-PWB makefiles. You can edit a non-PWB makefile, even when it belongs to the currently open project.

You can now use the Build, Rebuild All, and Build Target commands from the Project menu. The Build and Rebuild All commands work as they do with a PWB makefile by building the first target. However, the Language Options commands and the LINK Options command on the Options menu are unavailable. You set these kinds of options by editing the makefile.

When you close a non-PWB project, PWB saves the environment, window layout, and file history just as it does for a PWB project.

Where to Go from Here

This concludes the PWB tutorial section of this manual. If you wish, you can leave PWB by choosing Exit from the File menu (or by pressing ALT+F4).

Chapter 4, “User Interface Details,” explains how to start PWB, describes the elements of the user interface, and gives you an overview of the menus.

Chapter 5, “Advanced PWB Techniques,” explains search techniques (including regular-expression searching), describes how to use the browser, and shows how to write PWB macros.

Chapter 6, “Customizing PWB,” describes how to change the behavior of PWB to suit your needs.

Chapter 7, “PWB Reference,” contains an alphabetical reference to PWB menus, keys, functions, predefined macros, and switches.

CHAPTER 4

User Interface Details

This chapter summarizes the PWB user interface. It contains:

- General information on starting PWB.
- Instructions on how to use elements of the PWB screen.
- A description of the indicators on the status bar.
- A summary of every PWB menu command.
- Instructions on how to use menus and dialog boxes.

Starting PWB

You can start PWB in either of the following ways:

- From the the Windows operating system Program Manager
- From the operating-system command line

From the Command Line

To start PWB from the command line:

- At the operating-system prompt, type:

  PWB [[*options*]] [[*filename*]]

  PWB starts with its default startup sequence.

For a complete list of PWB options and their meanings, see “PWB Command Line” on page 131. Sometimes, you will want to modify the default startup sequence. The following procedures are examples of how you can start PWB to accommodate different circumstances.

To start PWB with an existing PWB project:

- Type `PWB /PP` *project*`.mak`

  PWB opens the specified project and the files that you were working on with the project.

To start PWB with the project you used in your last session:

- Type `PWB /PL`

  As with the previous option, the /PL option opens a project and arranges your screen as it was when you left PWB.

To start PWB quickly for editing a file such as CONFIG.SYS:

- Type `PWB /DAS /t CONFIG.SYS`

  This command suppresses autoloading of extensions and status files (/DAS). It also tells PWB not to remember `CONFIG.SYS` for the next PWB session (/t CONFIG.SYS).

Using the Windows Operating System Program Manager

Microsoft Windows offers features that can enhance program development, particularly if you plan to develop Windows-based applications. You can edit and build your application in an MS-DOS session and then immediately run it under the Windows operating system. See *Getting Started* for a full description of how to set up Windows operating system icons for MASM in the Windows Program Manager.

To start PWB with Windows, double-click the PWB icon.

You can add a Program Item to the Program Manager for each project you are working on. Use the PIF editor to open PWB.PIF, and then choose Save As on the File menu to create a .PIF file with the same base name as your project. Next, use the Optional Parameters text box to specify the /PF or /PP options and the name of the project makefile.

To run PWB in a window by default, you can change “Display Usage” in the PIF file to “Windowed” and (optionally) “Execution” to “Background.” Then, choose Project Templates on the Options menu. In the Build Rule edit field of the Customize Project Templates dialog box, type: `macro WXFLAGS "/w"` and select Set Build Rule. Choose OK.

Using the Windows Operating System File Manager

When programming, you are often concentrating on which file or project you want to work on and would prefer that the computer provide the right tool for the job. With the Windows File Manager, you can associate certain types of files with the commands that operate on those files. Therefore, when you double-click the filename in the File Manager, the right tool starts with the correct command-line options.

You can associate project makefiles (.MAK files) with the PWB .PIF file. Double-clicking a project makefile then starts PWB and opens that project, source files and all.

To associate PWB with .MAK files:

1. Select any file in the File Manager with the extension .MAK.
2. From the File menu, choose Associate.
3. Type the command `PWB.PIF` in the dialog box. (Make sure that your PWB.PIF file specifies a question mark (**?**) in the Optional Parameters text box.)

Now when you double-click a project makefile, the File Manager automatically starts PWB, and PWB opens that project.

Note Be sure you have set your PATH, INIT, and TMP environment variables prior to starting the Windows operating system so PWB can find all its files.

The PWB Screen

Figure 4.1 shows the PWB display. The table which follows it describes each of the user interface elements.

Figure 4.1 User Interface Elements

Name	Description
Menu bar	Lists available menus.
Menu	Lists PWB commands.
Desktop	Background area.
Icon	Displays a window in compact form.
Window	Contains source code; displays Help, browser results, build results, or error messages.
Scroll bars	Change position in file or list.
Status bar	Shows command buttons for the mouse and shortcut keys; summarizes commands and file and keyboard status.

Figure 4.2 shows a PWB window. The table which follows it describes each of a window's elements.

Figure 4.2 Window Elements

Name	Description
Window border	Moves window. Drag to move the window.
Close box	Closes the window. Click to close the window.
Window number	Identifies window. Press ALT+*number* to move to that window.
Window title	Indicates window contents, a filename, or pseudofile title.
Minimize box	Shrinks window to an icon. Click to minimize the window.
Maximize/Restore box	Enlarges window to maximum size or restores window to its original size.
Scroll up arrow	Scrolls up by lines. Click to scroll up.
Page up area	Scrolls up by pages. Click to page up.
Scroll box	Indicates relative position in the file. Drag to change position.
Page down area	Scrolls down by pages. Click to page down.
Scroll down arrow	Scrolls down by lines. Click to scroll down.
Size area	Sizes window. Drag to size the window.
Move bar	Moves window. Drag to move the window.

Figure 4.3 shows the PWB status bar. The table which follows it describes each of the status bar's elements.

Figure 4.3 Status Bar Elements

Name	Description
Message area	Shows command buttons for the mouse and shortcut keys, and summarizes commands.
Status	Indicates current file, editor, and keyboard status, as described in the following table.
Location	Shows the location of the cursor in the file.
Command buttons	Show common commands and shortcut keys. Click the button or press the key to execute the command.
Line	Indicates the line at the cursor. When scanning a file during a search or when loading a file, PWB displays the current line in the line indicator as specified by the **Noise** switch.
Column	Indicates the column at the cursor.

The status area of the status bar displays one of the following letters to indicate the corresponding status.

Letter	Description
T	File is temporary and is not recorded in the PWB status file.
R	File is no-edit (read-only); modification is disabled.
L	Line endings in the file are linefeed characters only.
M	File is modified.
P	File is a pseudofile.
A	Meta prefix (F9) is active.
X	Macro recording is turned on.
O	Overtype mode is enabled. In insert mode, no indicator appears.
C	CAPS LOCK is on.
N	NUM LOCK is on.

Figure 4.4 shows the Window menu with the PWB Windows cascaded menu pulled down. The table which follows it describes each element of a menu.

Figure 4.4 PWB Menu Elements

Name	Description
Menu	Displays a list of commands.
Menu command	Executes the command. When the command is dimmed, it is unavailable.
Shortcut key	Executes the command directly and bypasses the use of the menu. Press the key to execute the command.
Cascaded menu	Lists a group of related commands. The command for a cascaded menu has a small right arrow after the command. To open a cascaded menu, click the command or move the selection cursor to the command and press the RIGHT ARROW key. To close an open cascaded menu, press the LEFT ARROW key.
Access key	Executes the command. Press the highlighted letter key to execute the command.
Selection cursor	Indicates the selected command. Press the UP ARROW and DOWN ARROW keys to move the selection cursor. Press ENTER to execute the command.

PWB Menus

PWB commands are organized into menus; the menu names appear along the menu bar at the top of the screen. When a menu or command is selected, PWB displays a brief description of the selected menu on the status bar. To get more information about a menu or command, point the mouse cursor to the name and click the right mouse button, or highlight the name by using the arrow keys and then press F1.

File

The File menu provides commands to open, close, and save files. You can switch to any open PWB file or find a specific file on your disk. You can also print a selection, a file, or a list of files.

Command	Description
New	Start a new file
Open	Open an existing file
Find	Locate a file or list of files on disk
Merge	Merge one or more files into the current file
Next	Open the next file in the list of files specified on the command line
Save	Save the current file
Save As	Save the current file with a different name
Save All	Save all modified files
Close	Close the current file
Print	Print a selection, the current file, or a list of files
DOS Shell	Temporarily exit to the operating system
All Files	List all open files in PWB
Exit	Leave PWB

Edit

The Edit menu provides commands to manipulate text, set the selection mode, and record macros.

Command	Description
Undo	Reverse the effect of your recent edit
Redo	Reverse the effect of the last Undo
Repeat	Repeat the last edit
Cut	Delete selected text and copy it to the clipboard

Command	Description
Copy	Copy selected text to the clipboard
Paste	Insert text from the clipboard
Delete	Delete selected text without copying it to the clipboard
Set Anchor	Save the current cursor position
Select To Anchor	Select text from the anchor to the cursor
Stream Mode	Set stream selection mode
Box Mode	Set box selection mode
Line Mode	Set line selection mode
Read Only	Toggle the PWB no-edit state (to prevent accidental modification or to allow modification)
Set Record	Define a macro name and its shortcut key
Record On	Record commands for a macro

Search

The Search menu provides commands to perform single-file and multifile text and regular-expression searches. You can do single-file and multifile find-and-replace operations. You can define and jump to marks or go to specific lines.

Command	Description
Find	Search for an occurrence of a text string or pattern
Replace	Search for a string or pattern and replace it with another
Log	Turn multifile searching on and off
Next Match	Move to the next match
Previous Match	Move to the previous match
Goto Match	Go to the match at the cursor in the Search Results window
Goto Mark	Move to a mark or line number
Define Mark	Set a mark at the cursor
Set Mark File	Open or create a mark file

Project

The Project menu provides commands to open and create projects, build a project or selected targets in the project, and determine the location of build errors and messages.

Command	Description
Compile File	Compile or assemble the current source file
Build	Build the project
Rebuild All	Build all files in the project (even those that have not been modified)
Build Target	Build specific targets in the project
New Project	Create a new project
Open Project	Open an existing project
Edit Project	Change the list of files in the project
Close Project	Remove the current project from memory without changing its contents
Next Error	Move to the next error
Previous Error	Move to the previous error
Goto Error	Move to the error at the cursor in the Build Results window

Run

The Run menu provides commands to set arguments for the project's program, run and debug the program, run operating-system commands, and add or run custom Run menu commands.

Command	Description
Execute	Run the current program
Program Arguments	Specify commands passed to your program for Execute or Debug
Debug	Run CodeView for the current program
Run DOS Command	Perform any single DOS task without exiting PWB
Customize Run Menu	Add commands to the Run menu

The custom commands that you add to the Run menu appear after the Customize Run Menu command.

Options

The Options menu provides commands to set environment variables for use within PWB, customize the look and behavior of PWB, and assign keys to commands. For projects, you can set the build type, customize the project template, and set assembler and utility options.

Command	Description
Environment Variables	View and modify environment variables
Key Assignments	Assign keys that invoke functions and macros
Editor Settings	Change the setting of any PWB switch
Colors	Change screen colors
Build Options	Specify whether the program is built as a debug or release version; specify a build directory
Project Templates	Cascaded menu of commands for project templates
Language Options	Cascaded menu of compiler options commands

The Project Templates cascaded menu provides the following commands to manage project templates:

Command	Description
Set Project Template	Changes the run-time support and project template
Customize Project Template	Modify the current project template
Save Custom Project Template	Save the current template as a new, custom template
Remove Custom Project Template	Delete custom project templates

The Language Options cascaded menu provides the following commands for setting assembler and compiler options for any other languages that may be installed:

Command	Description
MASM Options	Set assembler options

Note Additional languages, such as Basic and FORTRAN, are listed when their PWB extensions are loaded. To load the Basic extension, rename PWBBASIC.XXT in the BIN subdirectory to PWBBASIC.MXT. For FORTRAN, do the same for PWBFORT.XXT.

The following commands appear when the utilities extension (PWBUTILS) is loaded:

Command	Description
LINK Options	Set linker options for your project
NMAKE Options	Set options for NMAKE when it builds the project
CodeView Options	Set options for CodeView when debugging the project

The following command appears when the browser extension (PWBROWSE) is loaded:

Command	Description
Browse Options	Define the way the Source Browser database is built

Browse

The Browse menu provides the commands for the PWB Source Browser. You can select a browser database. You can jump to specific definitions or symbols in your project and view complex relationships among program symbols. You can also view your program as an outline, function-call tree, or, if you are using Microsoft C++, you can even view it as a class-inheritance tree.

Command	Description
Open Custom	Open a custom browser database, open the project database, or save the current database
Goto Definition	Locate the definition of any symbol in your source code
Goto Reference	Locate the references to any name in the browser database
View Relationship	Query the browser database
List References	Display a list of functions that call each function and show the use of each variable, type, macro, or class
Call Tree (Fwd/Rev)	View which functions call other functions
Function Hierarchy	Display a program outline
Module Outline	Display an outline of program modules
Which Reference?	Display a list of possible references for the ambiguous reference at the cursor
Class Tree (Fwd/Rev)	View the class-inheritance tree (for the C++ language)
Class Hierarchy	View the hierarchy of classes (for the C++ language)
Next	Find the next definition or reference
Previous	Find the previous definition or reference
Match Case	Define whether or not browse queries are case sensitive

Window

The Window menu provides commands to manipulate and navigate windows in PWB.

Command	Description
New	Duplicate the active window
Close	Close the active window
Close All	Close all windows
Move	Start window-moving mode for the active window
Size	Start window-sizing mode for the active window
Restore	Restore a minimized or maximized window to normal size
Minimize	Shrink the active window to an icon
Maximize	Enlarge windows to maximum size
Cascade	Arrange windows to show all their titles
Tile	Arrange windows so that none overlap
Arrange	Organize windows in a useful configuration for viewing Help, source code, and Build Results
PWB Windows	Cascaded menu that lists the following special PWB windows:

PWB Window	Description
Build Results	View the results of builds
Search Results	View the results of logged searches
Print Results	View the results of print operations
Record	View, edit, save recorded macros
Clipboard	View the PWB clipboard
Help	Access the Help system
Browser Output	View the results of browser queries
1 *window1*	Move to window *n*
. . .	
5 *window5*	
All Windows	View a list of all open windows

The All Windows command does not appear until the full list of open windows is too long to fit on the menu.

Help

The Help menu contains commands to access the Microsoft Advisor Help system. You can see the index or table of contents for the system, get context-sensitive Help, and perform global plain-text searches in the Help.

Command	Description
Index	Display a list of available indexes
Contents	Display a table of contents for each component of the Help system
Topic	Display Help about the item or keyword at the cursor
Help on Help	Display information on how to use Help
Next	Display the next Help screen that has the same name as the topic you last viewed
Global Search	Search all open Help files for a string or regular expression
Search Results	View the results of the last global Help search
About	Display the PWB copyright and version number

Executing Commands

PWB commands appear in menus and as “buttons.” You can execute these commands in two ways:

- With a Microsoft Mouse or any fully compatible pointing device

  You perform mouse operations by “clicking”—moving the mouse cursor to the specified item and briefly pressing the left mouse button. “Double-click” by pressing the left button twice, quickly. Always use the left mouse button unless specifically instructed otherwise.

- With the keyboard

Choosing Menu Commands

To choose a menu command with the mouse:

1. Click the menu name to open the menu.
2. Click the command.

To choose a menu command from the keyboard:

1. Press the ALT key to activate the menu bar.
2. Press the highlighted character in the menu name (such as F for File).

   An alternative is:

1. Press the ALT key to activate the menu bar.
2. Use the RIGHT ARROW and LEFT ARROW keys to select a menu.
3. Press ENTER to open the menu.
4. Press the highlighted character in the command name (such as S for Save in the File menu), or use the UP ARROW and DOWN ARROW keys to select the command and then press ENTER.

There are several ways to close an open menu without executing a command.

To close a menu without executing a command:

- Do one of the following procedures:
  - Click outside of the menu.
  - Press ESC.
  - Press ALT twice.

When a menu command is dimmed (rather than black), it is unavailable. For example, when no windows are open, the Close command on the File menu is unavailable. If a command you want to use is unavailable, you must perform some other action or complete a pending action before you can invoke that command.

Shortcut Keys

Some commands are followed by the names of keys or key combinations. Press the shortcut key to execute the command immediately. You don't have to go through the menu. For example, press SHIFT+F2 to execute the Save command, which saves the current file.

All menu commands with shortcut keys and many other menu commands invoke predefined PWB macros to carry out their action. You can change the key or add new shortcut keys for these commands by assigning a key to the predefined macro. For a complete list of predefined macros and their corresponding menu commands, see "Predefined PWB Macros" on page 207. For more information on assigning keys, see "Changing Key Assignments" on page 109.

Many PWB functions have been assigned to keys besides those listed on the menus. Choose the Key Assignments command on the Options menu to view a list of functions and macros and their assigned keys.

Buttons

You can often execute commands by using buttons or boxes, which are areas of the screen that perform an action when you click them or select them from the keyboard. For example, the rectangle at the upper-left corner of a window is the "close box." Clicking this box with the mouse closes the window.

A command name surrounded by angle brackets (< >) appearing on the status bar or in a dialog box is a button. The following buttons are on the status bar when you first start PWB:

```
<General Help> <F1=Help> <Alt=Menu>
```

The General Help button brings up a screen that explains how to use the Help system. The other two buttons remind you of PWB functions: F1 summons Help, and ALT activates the menu bar. Clicking one of these buttons with the mouse performs the same function as pressing the key.

When you have opened more than one window, PWB displays the following buttons:

```
<F1=Help> <Alt=Menu> <F6=Window>
```

Click the Window button or press F6 to move to the next window.

When a menu is selected or a dialog box is displayed, an informative message appears on the status bar. While PWB displays this message, no buttons are available and clicking the status bar does nothing.

Dialog Boxes

When a menu command is followed by an ellipsis (**...**), PWB needs more information before executing the command. You enter this information in a dialog box that appears when you choose the command.

Dialog boxes can contain any of the items in Figure 4.5.

Figure 4.5 Dialog Box Elements

Option Button

A button that you select from a list of mutually exclusive choices. Click the one you want, press its highlighted letter, or use the arrow keys to move among the choices.

Text Box

An area in which you can type text. You can move the cursor within the text box by clicking the location with the mouse or by pressing the LEFT ARROW and RIGHT ARROW keys. You can toggle between insert and overtype mode by pressing the INS key. To select text for deletion, click and drag the mouse over the text or press SHIFT plus an arrow key. Press DEL to delete the text, or type new text to replace the highlighted text.

List Box

A box displaying a list of information (such as the contents of the current disk directory). If the number of items exceeds the visible area, click the scroll bar to move through the list or press PGUP, PGDN, or the arrow keys. You can move to the next item in the list that starts with a particular letter by typing that letter.

Combo Box

The combination of a text box and a drop-down list box. You can type the name of an item in the text box or you can select it from the list.

To open the list, click the highlighted arrow, or press ALT+DOWN ARROW or ALT+UP ARROW. You can then click the item or press the arrow keys to select the item you want. You can also press the first letter of an item to select it. When you have selected an item, click the highlighted arrow or press ALT+DOWN ARROW or ALT+UP ARROW to close the list.

Command Button

A button within angle brackets (< >) that invokes a command. Choose the OK button to accept the current options, or choose the Cancel button to exit the dialog box without changing the current options. Choose the Help button to see Help on the dialog box.

If one of the command buttons in a dialog box is highlighted, press ENTER to execute that command. You can also click a command button to execute that command. If a button contains an ellipsis (...), it indicates that another dialog box will appear when you choose it.

Check Box

An on/off switch. If the box is empty, the option is turned off. If it contains the letter X, the option is turned on. Click the box with the mouse, or press the SPACEBAR or the UP ARROW and DOWN ARROW keys to toggle a check box on
and off.

Key Box

A pair of braces (**{ }**) that allows you to enter a key by pressing the key. The key box is always followed by a text box where you can type the name of the key.

When the cursor is in the key box (between the braces), most keys lose their usual meaning, including ESC, F1, and the dialog box access keys. The key you press is interpreted as the key to be specified. Only TAB, SHIFT+TAB, ENTER, and NUMENTER retain their usual meaning. To specify one of these keys, type the name in the text box.

Figure 4.6 Key Box and Check Box

Clicking a dialog-box item either selects it (a text box, for example) or toggles its value (a check box or option button). You can also move among items with the TAB and SHIFT+TAB keys.

Dialog boxes usually contain access keys, identified by highlighted letters. Pressing an access key is equivalent to clicking that item with the mouse or moving to it by pressing TAB or SHIFT+TAB, and then pressing ENTER. Although some access keys are uppercase and others lowercase, dialog boxes are not case sensitive. Therefore, you can type either an uppercase or a lowercase character.

Note When the cursor is in a text box, access keys are interpreted as text. You must press ALT along with the highlighted letter. Pressing ALT is also required in list boxes because typing a letter by itself moves the cursor to the next item that starts with that letter.

CHAPTER 5

Advanced PWB Techniques

This chapter introduces you to some of the powerful features in the Programmer's WorkBench. It is not an exhaustive discussion of all the ways to use PWB. However, it can provide a starting point for you to begin your own investigation of PWB using the information in the Microsoft Advisor and in Chapter 7, "Programmer's WorkBench Reference."

This chapter contains:

- Find and search-and-replace techniques, including marks and regular expressions.
- How to use the PWB Source Browser.
- How to execute PWB functions and macros.
- An overview of PWB macros, macro recording, and the macro language.

Searching with PWB

PWB offers the following ways to search your files for information:

- Visually inspecting code, moving with the cursor or the PGUP and PGDN keys. This method is most effective either when you are familiarizing yourself with some old code or after you have switched from CodeView back to PWB and want to examine the local impact of a proposed change.
- Searching with text strings. When you have a specific string in mind (for example, `FileName`), you can find, in sequence, all the references to the name in your file.
- Searching with regular expressions. Regular expressions describe patterns of text. This is useful when you have a number of similarly named program symbols and you'd like to see them all in succession.

  For example, Windows API (application programming interface) names are made up of multiple words; the start of each new word is shown as a capital

letter (for example, GetTextMetrics). With this in mind, you might search for a string optionally starting with spaces and the letters "GetText" followed by any uppercase letter. This is expressed with a regular expression such as `*GetText[A-Z]`, which means zero or more spaces (using the `*` operator after a space), followed by `GetText`, followed by any uppercase letter (using a character class).

- Searching multiple files with text strings or regular expressions. A multifile search is called a "logged search." Instead of finding one match, PWB finds all matches in one operation. You can then browse the results of the search.
- Using the Source Browser. The Source Browser enables you to perform faster and more sophisticated searches than plain text searches because it maintains a complete database of relationships between program symbols. For example, to find all occurrences of `FileName` in your entire program (regardless of the number of files in the program), you can use the View References command from the Browse menu.

  The Source Browser has many more capabilities than just finding symbols. It can also produce call trees and perform sophisticated queries on the use-and-definition relationships among functions, variables, and classes in your program.

These searching techniques are discussed in detail in the following sections.

Searching by Visual Inspection

If you think you're close to the text you want to see, don't try a fancy search—use the PGUP or PGDN key. It's often faster. One trick you can use to speed up this method of searching is to leave a trail in the form of marks (names associated with file locations).

Using Marks

PWB lets you set named marks at any location in your file by using the Define Mark command from the Search menu or by using the **Mark** function. You can access these locations by name using the Goto Mark command or the **Mark** function.

For example, if you are writing code and want to leave certain sections until later, or if you are revising an existing program and don't fully understand all the algorithms, you might leave a mark at each location in the code you want to come back to. That way, you can work on some sections of the program first, and then come back to the marked areas after further research.

To save marks between PWB sessions, create a mark file using the Set Mark File command from the Search menu.

Using the Find Command

The Find command on the Search menu allows you to search a file using text strings and regular expressions. Searching forward uses the **Psearch** function (assigned to the F3 key), while searching backwards uses the **Msearch** function (assigned to the F4 key).

Find can help you locate any string of text in any file you specify. PWB usually searches the file you are currently editing. However, it can also search a list of files. This is particularly useful for finding all occurrences of a string in an entire project. The function used is called **Mgrep**.

The results of a multifile search are logged, that is, put into the Search Results window. To see the logged results of a search, choose Search Results from the PWB Windows cascaded menu. There are two ways to use the information that PWB puts into Search Results:

- You can look at the matches in sequence by choosing Next Match and Previous Match from the Search menu.
- You can go directly to a specific match by moving the cursor to the match listed in the Search Results window and choosing Goto Match from the Search menu. PWB then jumps to the location of the match.

The Match commands on the Search menu work with the Search Results window in exactly the same way that the Project menu's Next Error, Previous Error, and Goto Error commands work with the Build Results window. These Project menu commands are described in "Fixing Build Errors" on page 23.

To illustrate the logged-search technique, suppose you want to locate all instances of a software interrupt instruction in the SHOW project's source files.

To search all the source files in this project:

1. From the Search menu, choose Find.

   PWB brings up the Find dialog box.

2. Turn on Log Search check box.
3. Type `int` in lowercase.
4. Select the Match Case check box to exclude uppercase or mixed-case occurrences of the word.

5. Choose the Files button.

   PWB brings up the Search In Selected Files dialog box.

6. Type `SHOW*.ASM` in the File Name text box.

   This wildcard specifies all filenames beginning with `SHOW` and having the `.ASM` extension.

7. Choose the Add Pattern button to add the wildcard to the file list.
8. In the “Drives / Dirs” window, select the SAMPLES\SHOW subdirectory under the main MASM directory.
9. Return to the File Name text box by clicking the box or by pressing ALT+F.
10. Type `$INCLUDE:dos.inc` (with the environment variable, “INCLUDE,” all in caps). Preceding an environment variable name with a dollar sign causes the contents of that variable to be inserted into the search string. The INCLUDE variable normally contains the path to the directory where general-purpose include files are kept.
11. Press ENTER to add DOS.INC to the file list, or click Add / Delete.
12. Choose OK to start the search.

When PWB finishes the search, it displays the Log Search Complete dialog box.

From this dialog box you can:

- Choose View Results to open the Search Results window.
- Choose Cancel to close the dialog box.

Choose Cancel now (you will open the Search Results window later).

To go to the first match:

- From the Search menu, choose Next Match.

You can step sequentially through all occurrences of the string using the Next Match command. Choose Previous Match to move to the previous occurrence of the string. When you reach the end of Search Results, PWB displays the following message:

```
End of Search Results
```

Sometimes, you cannot focus the search narrowly enough to make a sequential scan of Search Results profitable. In this example, you wanted only instances of the software interrupt instruction, but PWB found many more occurrences of **int**. In these cases, you can examine the results of the search and skip the matches that aren't relevant.

To view the Search Results:

- To see all matches from the search, open the Search Results window. You can do this by choosing Search Results from the PWB Windows cascaded menu on the Window menu.

In the Search Results window, PWB displays the file, line, and column where the string was located. It also shows as much of the matching line as will fit in the window.

For example, if the instruction you were looking for is the Interrupt 10h in `PAGERR.ASM`, .you can jump directly to that location.

To jump directly to a match:

1. Put the cursor on the match.
2. From the Search menu, choose Goto Match.

   PWB opens the correct file if it is not already open and positions the cursor on the text you located.

You can use multifile searching regardless of whether the files that you want to search are open in PWB.

Using Regular Expressions

The PWB searching capabilities that you have used so far are useful when you know the exact text you are looking for. Sometimes, however, you have only part of the information that you want to match (for example, the beginning or end of the string), or you want to find a wider range of information. In such cases, you can use regular expressions.

Regular expressions are a notation for specifying patterns of text, as opposed to exact strings of characters. The notation uses literal characters and metacharacters. Every character that does not have special meaning in the regular-expression syntax is a literal character and matches an occurrence of that character. For example, letters and numbers are literal characters. A metacharacter is an operator or delimiter in the regular-expression syntax. For example, the backslash (`\`) and the asterisk (`*`) are metacharacters.

PWB supports two syntaxes for regular expressions: UNIX and non-UNIX. Each syntax has its own set of metacharacters. The UNIX metacharacters are `.\[]*+^$`. The non-UNIX metacharacters are `?\[]*+^$@#(){}`. Because it uses fewer metacharacters, the UNIX form is a little more verbose. However, it is more familiar to programmers who have experience with UNIX tools such as **awk** and **grep**. This book uses the UNIX syntax, but any expression that can be written with this syntax can also be written with the non-UNIX syntax.

The regular-expression syntax used by PWB depends on the setting of the **Unixre** switch (this is a Boolean switch, and UNIX is the default). You can change the **Unixre** switch by using the Editor Settings dialog box on the Options Menu.

Note PWB switches that take regular expressions always use UNIX syntax. They are independent from the **Unixre** switch.

Finding Text

In the multifile searching example, you learned how to locate every occurrence of **int** in the SHOW project. In a large project, finding every **int** would yield too many matches. To narrow the search, you can use a regular expression.

For this example, let's say you want to match any **int** instruction. You can specify this with a regular expression. The expression below matches text that:

- Begins with the keyword **int**
- Is followed by white space
- Is followed by one or more hex digits (characters between 0 and 9 or between A and F)

The syntax for this regular expression is shown in Figure 5.1.

Figure 5.1 Regular Expression Example

It illustrates the following important features of regular expressions:

1. Regular expressions can contain literal text. In this example, `int` is literal text and is matched exactly.
2. Regular expressions can contain predefined regular expressions. Here, `\:b` is shorthand for a pattern that matches one or more spaces or tabs (that is, white space). For a complete list of predefined regular expressions, see Appendix B.
3. You can use *classes* of characters in regular expressions. A class matches any one character in the class. For example, the class `[0-9a-f]` is the class of characters that contains the digits between 0 and 9 and the lowercase letters between A and F. The dash (–) defines a range of characters in a class.
4. The plus sign (+) after the class instructs PWB to look for one or more occurrences of any of the characters in the class. This is the key to regular expressions. You don't have to know exactly what the interrupt number is; all you have to do is describe what kind of characters make it up.

The pattern `int\:b[0-9A-F]` matches strings such as

```
        int     21h     ; DOS function interrupt
        int     3Ah
; Print 25 lines...
```

Figure 5.2 shows a more detailed way to write an expression that matches only an **int 20h** or an **int 21h**.

Figure 5.2 Complex Regular Expression Example

This expression is more precise than most searches require, but it is useful as an illustration of how to write a complex regular expression.

You can interpret this expression as follows:

1. Start at beginning of the line, which is specified by a caret (^) at the beginning of the regular expression. Using an initial caret is particularly helpful in a situation like this if your file uses space characters rather than tabs. Otherwise, when you begin your search criteria with \:b, the search will return one match for every space character preceding the matching text. For example, if your instruction column is indented eight spaces, searching for `\:bmov\:b` will return eight copies of every `mov` instruction, one for each of the preceding space characters. Including the inital caret, however, will result in only one match per line.
2. Skip any label on the line, without matching a comment line. The `[^; \t]` indicates a class made up of any characters that are *not* a semicolon, a space or a tab character. Within brackets, a caret (^) at the beginning of the class indicates an "inverse class," that is, one including all characters *except* the specified ones. Following the class is an asterisk, indicating that zero or more such characters may be present. In general, optional items are specified using the asterisk (`*`) operator, which indicates that zero or more of the preceding character should be matched. For example, the expression "`*`" means "match zero or more spaces."
3. Skip white space. The predefined UNIX regular expression "`\:b`" is equivalent to "`[ \t]+`", which requires that there be at least one space or tab.
4. Look for the "`int`" instruction as literal text.
5. Skip white space. The expression "`[ \t]+`" is equivalent to "`\:b`", and requires that there be one or more space or tab.
6. Skip optional "`0`" digits.
7. Look for a "`2`" digit as literal text.
8. Look for either a "`0`" digit or a "`1`" digit.
9. Look for an uppercase or lowercase "`h`" character.

This expression is so exact that it may take longer to write than the time it saves. The key to using regular expressions effectively is determining the minimal characteristics that make the text qualify as a match.

To find all int 20h and int 21h instructions:

1. From the Search menu, choose Find.
2. In the Find Text box, type `^\:bint\:b2[01]`.
3. Select the Regular Expression check box.
4. Choose the Files button.
5. Add the pattern `*.ASM` and the file `$INCLUDE:DOS.INC` to the file list.
6. Choose OK to start the search.

When the search is complete, choose View Results. You can see in the Search Results window that PWB matched only the int 20h and int 21h instructions.

Replacing Text

You can use regular expressions when changing text to achieve some extremely powerful results. A regular expression replacement can be a simple one-to-one replacement, or it can use "tagged" expressions. A tagged expression marks part of the matched text so that you can copy it into the replacement text.

For example, you can manipulate lists of files easily using regular expressions. This exercise shows how to get a clean list of files that is stripped of the size and time-stamp information.

To get a clean list of .ASM files in the current directory:

1. From the File menu, choose New.

   This gives you a new file for the directory listing.

2. Execute the function sequence **Arg Arg `!dir *.ASM` Paste**.

   The default key sequence for this command is to press ALT+A twice, type `!dir *.asm`, then press SHIFT+INS.

   Arg Arg introduces a text argument to the **Paste** function with an **Arg** count of two. The exclamation point (**!**) designates the text argument to be run as an operating-system command. Without the exclamation point, the text is the name of a file to be merged. If only one **Arg** is used, PWB inserts the text argument.

   PWB runs the **DIR** command and captures the output. When the **DIR** command is complete, PWB prompts you to press a key. When you press a key, PWB then inserts the results of the command at the cursor. For more information about this and other forms of the **Paste** function, see "Paste" in Chapter 7, "Programmer's WorkBench Reference."

3. From the Search menu, choose Replace.
4. In the Find Text box, type `\:b\:z \:z-.*$`

   This pattern means:

   - White space followed by
   - A number followed by
   - Exactly one space followed by
   - A number followed by
   - A dash (–) followed by

- Any sequence of characters, then
- End of the line

This string must be tied to the end of the line to prevent the search from finding anything too close to the beginning of the line.

5. Make sure there are no characters in the Replace Text text box.
6. Choose Replace All.

   PWB prompts you to verify that you want to replace text with an empty string.

7. Choose OK to confirm that you want to perform the empty replacement.

All the file-size, date, and time-stamp information is removed. Because you did not reuse any of the original text in the replacement, this is a simple regular expression replacement.

Choose Close from the File menu to discard the text you created in the previous exercise.

A more complicated task is backing up the .ASM files to a directory called LAST, which is assumed to be a subdirectory of the current directory. A batch file makes this easier. You can create such a batch file using regular expressions.

To create a batch file that copies the .ASM files to a subdirectory:

1. Create a list of .ASM files in the current directory as described in the previous example, but do not remove the file sizes, dates, and times.
2. Delete the heading printed by the **DIR** command.
3. From the Search menu, choose Replace.
4. In the Find Text text box, type `^\([^ ]+\)[ ]+\([^ ]+\).*`
5. This expression finds a string that starts at the beginning of the line (**^**). Placing parts of the expression inside the delimiters `\(` and `\)` is called "tagging."

   The first tagged expression (`\([^ ]+\)`) matches one or more characters that are not spaces. A leading caret in a class means "not."

   The pattern then matches one or more spaces (`[ ]+`), followed by the second tagged expression which matches one or more characters that are not spaces.

   The remainder of the line is matched by the wildcard (`.`), which matches any character, and the repeat operator (`*`). Matching the rest of the line is important because that is how this pattern removes everything after the filename. It discards these portions of the matched text.

6. In the Replace Text text box, type `COPY \1.\2 .\\LAST`
7. Select Replace All and choose OK to begin the find-and-replace operation.

PWB transforms each directory entry into a command to copy the file to the LAST subdirectory.

The word `COPY` is inserted literally. The text matched in the first tagged expression (the base name) replaces the expression `\1`. The period is inserted literally. The text matched by the second tagged expression (the filename extension) replaces the expression `\2`. The space is inserted literally. The text `.\\LAST` is inserted as `.\LAST`. Be sure to use two backslashes to indicate a literal backslash; otherwise, PWB expects a reference to a tagged expression such as `\1` and displays an error message.

You'll notice that the last two lines of the file are not useful in your batch file. They are the remnants of the summary statistics produced by the **DIR** command. Delete these two lines and you have a finished batch file.

Using the Source Browser

Another search technique is "browsing." Browsing uses information generated by the compiler to help you find pieces of code quickly. This section introduces you to some of the capabilities of the Source Browser. The browser is a handy tool for moving about in projects, large and small.

In addition to navigating through your program, you can use the browser to explore the relationships between parts of the project. The browser database contains full information about where each symbol is defined and used and about the relationships among modules, constants, macros, variables, functions, and classes. Note that the browser files can be very large.

Creating a Browser Database

Before you can use the PWB Source Browser, you must build a browser database. PWB helps you maintain this database automatically as a part of a normal project build.

To build a browser database:

1. Open the SHOW project using the Open Project command from the Project menu (this project is located in the SAMPLES\SHOW subdirectory of your main MASM directory).
2. From the Options menu, choose Browse Options.

   PWB displays the Browse Options dialog box.

3. Select the Generate Browse Information check box.
4. Choose OK.

   The browser changes the project makefile to build the project. It adds compiler options for creating browser information (.SBR files). It includes a BSCMAKE command which combines the .SBR files and creates a browser database (a .BSC file).

5. From the Project menu, choose Rebuild All.

   Rebuilding the entire project ensures that the database contains up-to-date information for all files in your program.

When the build completes, the following new files are on your disk:

- SHOW.BSC, the browser database
- SHOWUTIL.SBR, a zero-length “placeholder” for the SHOWUTIL module.
- PAGER.SBR, a placeholder for PAGER.
- SHOW.SBR, a placeholder for SHOW.

After adding each .SBR file’s contribution to the database, BSCMAKE truncates it and leaves the empty .SBR file on disk to provide an up-to-date target for later builds. Leaving these files on the disk ensures that a browser database is not rebuilt unless it is out of date with respect to its source files.

A PWB project is not required to create a browser database (although it is convenient). For information on how to build a browser database for non-PWB projects, see “Building Databases for Non-PWB Projects” on page 94.

Finding Symbol Definitions

When you are working on a program, it’s easy to forget where a particular variable, constant, or function is defined. You can use the Find command to locate occurrences of a symbol, but that offers little information about which one is the definition. To make such searches easier, you can choose Goto Definition from the Browse menu to jump directly to the definition of any symbol in your program.

The following procedure uses the SHOW project to demonstrate how powerful the browser can be.

To investigate the GetNamePos procedure:

1. From the Window menu, choose Close All.
2. Open SHOW.ASM.
3. Go to line 174 (from the Search menu, choose Goto Mark, type `174`, and press ENTER).
4. Move the cursor to the `GetNamePos` procedure.
5. From the Browse menu, choose Goto Definition.

   PWB displays the Goto Definition dialog box.

Notice that `GetNamePos` is highlighted and the defining file's name is displayed in the list box to the right. More than one defining file is listed if a name is defined in several scopes.

6. Choose OK.

   PWB opens SHOWUTIL.ASM and shows the definition of GetNamePos.

Showing the Call Tree

Often when analyzing an existing program's flow, or when looking for opportunities for optimization, it's useful to refer to a "call tree." A call tree is a view of your program that provides, for each function, a list of other functions called.

To generate a call tree of SHOW:

1. From the Browse menu, choose Call Tree.

   PWB displays the Display Tree dialog box.

2. Choose SHOW.ASM from the Modules list box.

   Notice that the Functions list box changes to show only the functions in SHOW.ASM.

3. Choose OK to see the call tree.

Three kinds of annotations appear in the call tree:

?
: A symbol followed by a question mark is used by your program but not defined in any of the program files in the browse database. These are often library functions.

[*n*]
: The number *n* between square brackets shows symbols that are used more than once. In the preceding example, `GetNamePos` is listed (under SHOW.ASM) as:

```
GetNamePos[3]
```

  This means that there are three references to `GetNamePos` in `SHOW`.

... (ellipsis)
: The ellipsis means that the full information for the function appears elsewhere in the call tree.

Finding Unreferenced Symbols

As you write, rewrite and maintain a program, you will occasionally remove function calls or references to global variables, leaving unused code or data space in your program. Since the browser database contains information about where every function and variable is referenced, you can easily identify ones that are not used. This section shows how to use the Source Browser to find and remove the extra code and data.

The system include files define many more functions than most programs use. Therefore, unreferenced functions in your program are easiest to find when using a browser database that does not contain the system include files. This example begins by building a browser database for SHOW that does not contain information defined by system include files.

To build the SHOW browser database:

1. From the Options menu, choose Browse Options.

   PWB displays the Browse Options dialog box.

2. In the Browse Options dialog box, select the Generate Browse Information, the Exclude System Include Files, and the Include Unreferenced Symbols check boxes.
3. Choose OK.

Now that the browse options are set, rebuild the project and browser database by choosing Rebuild All from the Project menu. With the updated browser database, you can obtain a list of references for functions and variables.

To get a list of references for function and variables:

1. From the Browse menu, choose List References.

   PWB displays the List References dialog box.

2. Select the Functions, Variables and Macros options, and then choose OK.

PWB opens the Browser Output window and creates the list of references. Each name is followed by a colon and a list of functions that refer to the name.

To find an unreferenced symbol:

- Search for the regular expression `:$` (colon, dollar sign).

  This pattern specifies a colon at the end of the line. It finds names that are followed by an empty list of references.

In the list of references created above for SHOW, a search for this expression will find no matches, since there are no unreferenced symbols.To find all unreferenced items with one search, you can perform a logged search and add only `<browse>` (the Browser Output pseudofile) to the file list. This is especially useful for large projects.

To go to the definition of an unreferenced symbol in the source:

1. Place the cursor on the symbol in question. From the Browse menu, choose Goto Definition.

   PWB automatically selects the definition of the symbol under the cursor. However, if the symbol begins with “@” or “?” or other punctuation characters, the nonalphabetic character is not automatically recognized as part of the symbol name. To include it, mark the entire name before choosing Goto Definition.

2. Choose OK.

PWB jumps to the definition of the selected symbol in the appropriate source file, where you can remove the unused function, macro or variable.

Advanced Browser Database Information

In the previous sections, you learned the basics of building a browser database and some useful applications of the Source Browser. In this section, you will find information on what goes into a browser database and how to estimate the disk requirements to build one. You will also learn how to build a database for non-PWB projects and how to build a single database for related projects.

Estimating .SBR and .BSC File Size

When you build a browser database, you first create an .SBR file for each source file in the project. Each of these files contains the following information:

- The name of the source file and the files it includes.
- Every symbol defined in the source file and the files it includes.

  These symbols are the names of all functions, types (including the names of all classes, structures, and enumerations and their members), macros (including symbols in the expanded macro), and variables in the file. These symbols also include all parameters and local variables for the functions.
- The location of all symbol definitions in the files.
- The location of all references to every symbol in the files.
- Linkage information.

This is a tremendous amount of information about your program and can therefore occupy a large quantity of disk space. The benefit is that the Source Browser provides fast, sophisticated access to this database of knowledge about your program.

For assembler source files, the .SBR file may be between a quarter and a half the size of the preprocessed source file (that is, the source file with comments removed, all files included, and all macros expanded).

You might assume that the resulting browser database (.BSC file) is approximately the sum of all the .SBR files. However, the browser database is the *union* of the information in the component .SBR files. This means that the .BSC file is usually not very large. Much of the information in the .SBR files is defined in include files, which are common to many modules in the project. The union of the .SBR files is relatively small because most of the include-file information is duplicated in each .SBR file.

Even for C or C++ programs, which tend to create much larger .BSC files, a good-sized program will seldom require a .BSC file larger than 500K.

Building Databases for Non-PWB Projects

The simplest way to build a browser database for non-PWB projects is to build the browser database separately from the project. You can use a makefile or a batch file for this purpose. The process requires only two steps:

1. Create an .SBR file for each module. The simplest way to do this is to run the compiler with the options to produce an .SBR file and no other files. For example, the ML command line:

   ```
   ML /Zs /W0 /Fr *.asm
   ```

   specifies that the compiler processes all .ASM files in the current directory, checks syntax only (`/Zs`) and issues no warnings (`/W0`). Therefore, no object files are produced. However, browser information (.SBR files) are generated (`/Fr`).

2. Combine the .SBR files into a browser database.

   The syntax for this command is:

   BSCMAKE *options* /**o***project*.**BSC *.sbr**

   For complete information on BSCMAKE options and syntax, see Chapter 19.

The process of creating a browser database changes little between projects. Therefore, you could use a batch file for many projects similar to the following example:

```
ECHO OFF
REM Require at least one command-line option
IF %1.==. GOTO USAGE

REM Compile to generate only .SBR files
ML /Zs /W0 /Fr *.asm

REM Build the browser database
BSCMAKE %2 %3 %4 %5 %6 %7 %8 /o%1.BSC *.sbr
GOTO END

:USAGE
REM Print instructions
ECHO -Usage: %0 project [option]...
ECHO -    project      Base name of browser database
ECHO -    [option]...  List of BSCMAKE options
:END
```

This batch file assumes that all the project sources are in the current directory. It requires that you specify the name of the browser database and allows BSCMAKE options. You may want to change this file to specify different BSCMAKE or assembler options.

If your project's sources are distributed across several directories, you must write a custom batch file or makefile to build the database. For more information on the BSCMAKE utility, see Chapter 19.

To use a custom browser database in PWB:

1. From the Browse menu, choose Open Custom.
2. Choose the Use Custom Database button.
3. Select your custom browser database and choose OK.

   If you want to save this database name permanently, choose Save Current Database.

4. Choose OK.

   The PWB Source Browser opens your custom database.

You can now browse your non-PWB project.

If you are using a makefile to build your project, you can choose Open Project from the Project menu and open it as a non-PWB project makefile. If the project makefile has the same base name as the browser database and resides in the same directory, PWB automatically opens the database when you open the project. For more information on using a non-PWB makefile for a project in PWB, see "Using a Non-PWB Makefile" on page 55.

Building Combined Databases

If you have two or more closely related projects, you can combine the browser databases for the projects. For example, if two large programs differ only in one or two modules so that most of the sources are shared between the two projects, it can be useful to browse both projects with a single browser database.

To build a combined browser database:

1. Generate the .SBR files for both projects.
2. Pass all of the .SBR files to BSCMAKE to build the combined database.

The resulting database is the inclusive-OR of the information in the two projects.

Executing Functions and Macros

The menus and dialog boxes in PWB provide access to almost everything you need to do to develop your projects. You can edit, search, and browse your source files. You can build, run, and debug your project, and you can view Help for the entire system. However, the visible display provides access to only part of the capabilities available in PWB. Behind the menu commands lie functions with many more options than you can access from the menus. Many functions and macros are not assigned to keys by default.

The sophisticated PWB user learns how to use the functions and predefined macros to perform the precisely correct action. Each function has several forms that are invoked with the combinations of the **Arg** and **Meta** prefixes. These two functions are used to introduce arguments and modify the action of PWB functions.

Arg (ALT+A)

The fundamental function in PWB. You use **Arg** to begin selecting text, introduce text and numeric function arguments, or modify the action of functions by increasing the Arg count.

To pass a text argument to a function, for example, press ALT+A, and then type the text. The text you type doesn't go into your file. The Text Argument dialog box appears when you type the first letter of the text.

You can then edit the text. PWB displays the current argument count and Meta state in the dialog box.

Notice that there is no OK button in this dialog box. Instead of choosing OK, press the key for the function you want to execute with this argument. Choose the Cancel button if you do not want to execute a function.

Meta (F9)

Modifies the action of a function in different ways from the various argument types. It generally toggles an aspect of the function's action.

For example, the text-deletion functions usually move the deleted text to the clipboard. However, when modified with **Meta**, they clear the text without changing the clipboard.

The combination of **Arg** and **Meta** greatly increases the number of variations available to each function. For example, the **Psearch** function can perform different search operations depending on how it is executed. **Psearch** can:

- Repeat the previous search (**Psearch**).
- Search for text (**Arg** *text* **Psearch**).
- Perform a case-sensitive text search (**Arg Meta** *text* **Psearch**).
- Search for a regular expression (**Arg Arg** *text* **Psearch**).
- Search for a case-sensitive regular expression (**Arg Arg Meta** *text* **Psearch**).

Because you can reassign keys to your preference, the PWB documentation cannot assume that a specific key executes a given function or macro. Therefore, the PWB documentation gives a sequence of functions or macros by name, followed by the same sequence of actions by key name. In this book, the key is the default key. In PWB Help, the displayed key is the one currently assigned to that function. When no key is assigned, PWB displays `unassigned`.

For example, to insert the definition of a macro at the cursor, you pass the name of the macro to the **Tell** function and modify **Tell**'s action with the **Meta** prefix. This sequence of actions is expressed as follows:

- Execute the function sequence **Arg Meta** *macroname* **Tell** (ALT+A F9 *macroname* CTRL+T).

If the **Tell** function is assigned to a different key, Help displays that key in place of CTRL+T.

Chapter 7, "Programmer's WorkBench Reference," contains complete descriptions of all forms of each function in PWB.

Executing Functions and Macros by Name

The most frequently used functions and macros are assigned to certain keys by default. For example, the **Paste** function is assigned to SHIFT+ENTER, **Linsert** is assigned to CTRL+N, and so on. Sometimes, however, you want to use a function or macro that is not assigned to a key. You can always assign a key by using the Key Assignments command or by using the **Assign** function. However, that is a lot of trouble for something you need only once. PWB allows you to execute a function or macro by name, rather than by pressing a key.

To execute a function or macro by name:

- Perform the function sequence **Arg** *function* **Execute** (ALT+A *function* F7).

In other words, press ALT+A (execute the **Arg** function), type the name of the function or macro, and then press F7 (invoke the **Execute** function).

The argument to **Execute** doesn't have to be a single function or macro name. It can be a list of functions and macros. The argument is really a temporary, nameless macro. This means that you can do anything in an argument to **Execute** that you can do in a macro. PWB follows the rules for macro syntax and execution. You can define labels, test function results, and loop.

Warning When executed from a macro, PWB functions that display a yes-or-no prompt assume a "Yes" response. To restore the prompt, use the macro prompt directive (<). For more information, see "Macro Prompt Directives" in PWB Help.

Writing PWB Macros

The Programmer's WorkBench, like other editors designed for programmers, provides a macro language so that you can customize and extend the editor or automate common tasks. You can create macros in one of the following ways:

- By recording actions you perform. The recording mechanism allows you to perform a procedure once, while PWB is recording. After you've recorded it, you can execute the macro to repeat the recorded procedure.
- By manually writing macros. This technique is less automatic but does allow you to write more powerful macros.

These two techniques are not mutually exclusive. You can start by recording a macro that approaches the steps you want to perform, then edit it to expand its functionality or handle different situations.

When Is a Macro Useful?

Macros are useful for automating procedures you perform frequently. You may also write macros that automate tedious one-time tasks.

Of course, not every task is a good candidate for automation. It might take longer to write the macro than to do the task by hand. If you don't expect to perform a task often, don't automate it. Also, automated editing procedures introduce an element of risk. You might not foresee situations that your macro can encounter. Incorrect macros can sometimes be destructive.

A little experience with macros and some careful testing will enable you to create a good set of macros for your own use.

Recording Macros

Recording actions you perform with the mouse or at the keyboard can be a powerful way to write a macro. You turn on recording and perform the actions that you want the macro to execute. You can concentrate on the task that you want to automate, instead of concentrating on the syntax of the macro language.

For example, if you occasionally reverse characters when you type quickly, a macro to transpose them is useful. Before recording a macro to transpose characters, you should think about what you are going to do while recording the macro. To transpose characters, you will select the character at the cursor, cut it onto the clipboard, move over one character, and then paste the character you cut.

To record a macro that transposes characters:

1. From the Edit menu, choose Set Record.

   PWB brings up the Set Macro Record dialog box.

2. In the Name text box, type `Transpose`.
3. Click the mouse in the key box (between the braces **{ }**), or press TAB until the cursor is in the key box.
4. Press CTRL+SHIFT+T (for transpose).

PWB automatically fills in the name of the key you pressed.

5. Press TAB to leave the key box, and then choose OK.

   PWB closes the Set Macro Record dialog box. When you turn on macro recording, PWB records a macro called Transpose and associates it with SHIFT+CTRL+T.

Important The Set Macro Record command does *not* start the macro recorder. It only specifies the name and key association for the macro you are going to record.

---

6. From the Edit menu, choose Record On.

   When you choose Record On, the macro recorder starts. To indicate that the macro recorder is running, PWB displays the letter X on the status bar. Notice that the Project, Options, and Help menus are unavailable while PWB is recording a macro.

7. Select the character at the cursor by holding down the SHIFT key and pressing the RIGHT ARROW key.
8. Press SHIFT+DEL to cut the character onto the clipboard.
9. Press the RIGHT ARROW key to move the cursor to the new location for the character.

10. Press SHIFT+INS to paste the character from the clipboard back into the text.
11. From the Edit menu, choose Record On to stop the macro recorder.

Press SHIFT+CTRL+T to switch the character at the cursor with the character to the right. You can now use the new macro and key assignment for the rest of the PWB session.

To edit the macro:

- From the Window menu, choose Record from the PWB Windows cascaded menu.

  PWB opens the Record window.

The Record window shows the definition of the `Transpose` macro that you just recorded. You can edit the definition to change the way the macro works. For example, you decide that the macro should reverse the character at the cursor with the character to the left, instead of the character to the right.

To redefine the macro:

1. Change the macro to read as follows:

   ```
   Transpose:=select left delete left paste
   ```

2. Move the cursor to the macro definition.

3. Press ALT+=, the default key for the **Assign** function.

   Assigning the macro replaces the previous definition of `Transpose` with the new definition.

4. Return to the file you were originally viewing.

Up to this point, the macro exists only in memory. To use your recorded macro for subsequent PWB sessions, you must save the definition of the macro to disk.

To save the macro:

1. If the Record window is not open, choose Record from the PWB Windows cascaded menu.

   PWB opens the Record window.

2. From the File menu, choose Save.

   PWB inserts the macro definition and the key assignment into your TOOLS.INI file for future sessions. When you leave PWB, you are prompted to save TOOLS.INI. Your changes are not permanent until you actually save TOOLS.INI.

Flow Control Statements

Recorded macros have the inherent limitation of playing back one fixed sequence of commands. Often you need a macro to execute repeatedly until some condition is satisfied. This requires that you use flow control statements to govern the actions your macro takes.

All editor functions return a true or false value. The macro flow control operators that use these values are:

Operator	Meaning
+>*label*	Branch to *label* if last function yields TRUE
–>*label*	Branch to label if last function yields FALSE
=>*label*	Branch unconditionally to label
:>*label*	Define *label*

These rudimentary operators are not as sophisticated as a high-level language's IF statement or FOR loop. They are more like an assembly language's conditional jump instruction. However, they provide the essential capabilities needed for writing loops and other conditional constructs.

Flow Control Example

If you frequently perform multiple-window editing, a macro that restores the display to a single window can be helpful. Such a macro requires the following logic:

1. Switch to the next window.
2. If the switch is not successful (meaning that only one window is present), end the macro.
3. If the switch is successful (another window is present), close that window and go back to step one.

This macro will be called CloseWindows and assigned to SHIFT+CTRL+W.

To create the CloseWindows macro:

1. From the File menu, choose All Files.

   PWB displays the All Files dialog box.

   Notice that your TOOLS.INI file is in the list of open files, even though you did not explicitly open it. PWB opens TOOLS.INI to load its configuration information (unless when you specify /DT on the PWB command line).

2. Select TOOLS.INI file in the list of open files.
3. Choose OK.

   PWB opens a window and displays your TOOLS.INI file.

4. Find the section of TOOLS.INI that begins with `[pwb]`. This is the section where PWB keeps its startup configuration information.
5. In the PWB section, type the following two new lines:

   ```
   CloseWindows:= :>Loop Openfile -> Meta Window Window =>Loop
   CloseWindows: SHIFT+CTRL+W
   ```

   If you want these definitions to take effect immediately, select both lines and press ALT+= to execute the Assign function. You can also assign the definitions one at a time.

6. Choose Save from the File menu to make this macro and key assignment part of your TOOLS.INI file.

   The next time you start PWB, the CloseWindows macro is defined and assigned to the SHIFT+CTRL+W key.

The first line you typed uses the **:=** operator to associate the macro definition with the name “CloseWindows.” After the operator is the list of functions and macro operators that specify what the macro is to do. The second line is a

separate statement that uses the `:` operator to assign the macro to the SHIFT+CTRL+W key.

The CloseWindows macro works as follows:

1. `Loop` defines a label called `Loop`. There cannot be a space between the `:>` operator and the label name.
2. `Openfile` switches to the window under the active window.
3. The `->` operator examines the return value from the **Openfile** function. If the function returns false because there is no other window, the `->` operator exits the macro.
4. The phrase `Meta Window` closes the active window.
5. `Window` returns to the window you started from.
6. `Loop` unconditionally transfers control back to the `Loop` label and starts the sequence again.

When this macro is defined, you can press SHIFT+CTRL+W whenever you want to close all windows except the active window.

User Input Statements

PWB macros can prompt for input. This helps you write more general macros. For example, you might keep a history of the changes you make to a file at the top in a format similar to the following:

```
//** Revision History **
//15-Nov-1991:IAD:Add return value for DoPrint
//31-Oct-1991:IAD:Implement printing primitives
```

To facilitate entering the revision history in reverse chronological order and to make it easy to keep track of where you were in the source file, you can write a macro to perform the following steps:

1. Set a mark at the cursor for future reference.
2. Insert a revision history header at the beginning of the file if one is not present.
3. Insert the current date.
4. Prompt for initials and insert them just below the header.
5. Prompt for comments and insert them after the initials.
6. Return to the saved position in the file.

Note that while this macro is executing, you can choose the Cancel button in the dialog boxes that prompt for initials and comments. The macro must handle these cases and gracefully back out of the changes to the file.

To enter this macro in TOOLS.INI:

1. Open TOOLS.INI for editing.
2. Type the following macros and key assignment in the `[pwb]` section of TOOLS.INI:

```
LineComment:=“//  "
RevHead:= "** Revision History **"
RevComment:=  \
  Arg Arg "Start" Mark                  \
  Begfile Arg RevHead Psearch +>Found  \
  Linsert LineComment RevHead           \
:>Found  \
  Down Linsert Begline LineComment Curdate " ("     \
  Arg "Initials" Prompt ->Quit Paste Endline ") "  \
  Arg "Comment"  Prompt ->Quit Paste =>End        \
:>Quit Meta Ldelete  \
:>End Arg "Start" Mark
RevComment:Ctrl+H
```

   There are at least two spaces before the backslash at the end of each line. The backslashes are line-continuation characters. They allow you to write a macro that is more than one line long. In this case, line continuations format the macro in a readable way. To further assist in readability, you can indent the parts of the macro which define the actual keystrokes, as in the preceding example.

3. Choose Save from the File menu to save your changes.
4. To reinitialize PWB, execute the **Initialize** function by pressing SHIFT+F8.

   PWB discards all of its current settings and rereads the PWB section of TOOLS.INI. The same effect can be achieved by quitting and restarting PWB.

The following discussion analyzes the workings of the definitions you added to TOOLS.INI. It repeats one or two lines from the text you typed and describes how each line works. You may want to refer to the full definition as you follow along.

The first two lines

```
LineComment:="//"
RevHead:= "** Revision History **"
```

define two utility macros that are used by the main RevComment macro. They define strings that are used several times in RevComment.

The third line

```
RevComment:=  \
```

declares the name of the macro. The succeeding lines define the action of the RevComment macro.

The first line of the definition

```
Arg Arg "Start" Mark                \
```

sets a mark named “Start” at the cursor so that the macro can restore the cursor position after inserting the comments at the beginning of the file.

The next line

```
Begfile Arg RevHead Psearch +>Found  \
```

moves to the beginning of the file (**Begfile**), then searches forward for the revision-history header. If the header is found, PWB branches to the `Found` label; otherwise, it executes the next line.

```
Linsert LineComment RevHead         \
```

If the macro is here, the header was not located in the file. The **Linsert** function creates a new line, and PWB types the revision-history header. The macro continues with the line:

```
:>Found  \
```

This line defines the `Found` label. At this point in the macro, the cursor is on the line with the header. The next lines insert the new revision information, starting with the following line:

```
Down Linsert Begline LineComment Curdate " ("    \
```

PWB moves the cursor down one line (**Down**), inserts a new line (**Linsert**), moves to the beginning of the line (**Begline**), and calls the LineComment macro to designate the line as a comment. PWB then types the current date (**Curdate**) and an open parenthesis.

The macro prompts for initials:

```
Arg "Initials" Prompt ->Quit Paste Endline ") "  \
```

The macro uses the **Prompt** function to get your initials. If you choose the Cancel button, the function returns false, so the macro branches to the label `Quit`. If you choose the OK button, the text you typed in the dialog box is passed to the **Paste** function, which inserts the text. The macro moves the cursor to the end of the line (**Endline**) and types a closing parenthesis.

The code on this line explicitly handles the case when you cancel the prompt (the false condition). The phrase `->Quit` causes PWB to skip to the label `Quit` when **Prompt** returns false.

If you use the **Prompt** function and you do not handle the false condition, a null argument (a text string with zero length) is passed to the next function.

Therefore, a phrase like `Arg "Que?" Prompt Paste` pastes either the input or nothing, depending on whether you choose the OK or Cancel button. Passing a null argument to **Paste** is harmless, but some functions require an argument. In these cases, you can use the `->` operator to terminate the macro.

The RevComment macro uses an explicit label so that it can end the macro without an error when you choose the Cancel button. The next line of the macro is almost the same as the previous line in the macro.

```
Arg "Comment"  Prompt ->Quit Paste =>End         \
```

On this line, if the paste is carried out, an unconditional branch is taken to the label `End` and passes over the Quit branch, which is defined on the next line.

```
:>Quit Meta Ldelete  \
```

The Quit branch is taken when you cancel a prompt. The macro has to clean up the text already inserted by the macro. The **Meta Ldelete** function deletes the incomplete line that would have been the revision-history entry. The next line defines the last step of the macro.

```
:>End Arg "Start" Mark
```

The `End` label defines the entry point for the common cleanup code. This line restores the cursor to the initial position when you invoked the macro. Because this line does not end in a line-continuation character (\), it is the end of the RevComment macro.

The last line that you typed is not part of the RevComment macro. It is a separate TOOLS.INI entry.

```
RevComment:Ctrl+H
```

This line assigns the CTRL+H key to the RevComment macro.

You can polish this macro by adding `Arg "Start" Meta Mark` to the end of the macro. This phrase deletes the mark. A better alternative is to use the **Savecur** and **Restcur** functions instead of named marks. However, this example uses named marks to illustrate how to use them in a macro.

CHAPTER 6

Customizing PWB

PWB is a completely customizable development environment. You can modify PWB in the following ways:

- Changing mapping of keystrokes to actions.
- Changing default behavior of PWB (for example, how tabs are handled or if PWB automatically saves files).
- Changing the colors of parts of the PWB display.
- Adding new commands to the Run menu.
- Programming new editor actions (macros).

  Instructions on how to write macros are in "Writing PWB Macros" on page 98.

In addition to the customizations that you can make by using the commands in the Options menu, you can also customize PWB by editing the TOOLS.INI file.

Note Another category of customization that is not covered in this book is how to write PWB extensions. An extension is a dynamically loaded module that can access PWB's internal functions. Extensions can do much more than macros. To learn more about writing PWB extensions, see the Microsoft Advisor Help system (choose "PWB Extensions" from the main Help table of contents).

Changing Key Assignments

PWB maps actions (functions and macros) to keys. You can assign any of these actions to keys other than the default keys.

For example, **Exit** is a PWB function. Its default key assignment is F8. A BRIEF user may prefer to use ALT+X to leave the editor.

To make ALT+X execute the Exit function:

1. From the Options menu, choose Key Assignments.

   PWB displays the Key Assignments dialog box.

2. Select Exit in the Macro/Function List box, or type `exit` in the Macro/Function Name text box.
3. Move the cursor to the New Key box between the braces (**{}**) by clicking between the braces or by pressing ALT+K.
4. Press ALT+X.

   PWB types `ALT+X` in the text box after the braces and displays the name of the macro or function that ALT+X is currently assigned to. With the default settings, you can see that ALT+X is assigned to the **Unassigned** function. Pressing a key in the key box is a quick way to find out the name of the function assigned to the key.

Note When the cursor is in the key box (between the braces), most keys lose their usual meaning, including ESC, F1, and the dialog box access keys. The key you press is interpreted as the key to be assigned. Only TAB, SHIFT+TAB, ENTER, and NUMENTER retain their usual meaning. To assign one of these keys, type the name of the key in the text box.

---

5. Press TAB to move the cursor out of the key box.
6. Choose Assign.

PWB assigns Exit to the ALT+X key. Note that Exit is still assigned to the F8 key. Functions can be assigned to many keys.

7. Choose OK.

Important To change a key, you *must* choose the Assign button. The OK button dismisses only the dialog box. It does not perform any other action. This design allows you to assign many keys in one session with the dialog box.

The change remains in effect for the duration of the session.

To make a permanent key assignment:

1. From the Options menu, choose Key Assignments.
2. Choose Save.

   PWB displays the Save Key Assignments dialog box, which lists all of the unsaved assignments that you have made during the PWB session by using the Key Assignments dialog box.
3. Delete any settings that you do not want to save.
4. Choose OK.

PWB writes your new settings into the `[PWB]` section of TOOLS.INI for subsequent sessions. When you exit PWB, you are prompted to save TOOLS.INI. Your changes are not permanent until you actually save the file to disk.

If you already know the function name, you can make a quick assignment for the current session by using the **Assign** function instead of going through the Key Assignments dialog box.

To assign a key using the Assign function:

- Execute the function sequence:

  Arg *function*:*key* **Assign** (ALT+A *function*:*key* ALT+=).

  For example, to assign **Exit** to ALT+X:

  1. Press ALT+A to execute **Arg**.
  2. Type `exit:ALT+X`
  3. Press ALT+= to execute **Assign**.

  The assignment is in effect for the rest of the PWB session.

The key assignments you make by using the **Assign** function are not listed in the Save Key Assignments dialog box.

To discover the name of the function or macro that is currently assigned to a key, use the Key Assignments dialog box (as previously described) or use the **Tell** function.

To find a current key assignment using Tell:

1. Press CTRL+T to execute **Tell**.

   PWB displays the prompt:

2. Press the key you want to find out about.

   If you press F10, PWB displays the function assigned to the F10 key (Openfile).

The **Tell** function has many other uses in addition to displaying key assignments. For more information on **Tell**, see page 202.

Changing Settings

When you first use PWB, you don't have to specify the tab stops, whether the editor starts in insert or overtype mode, and so on. These settings (called "switches") are all covered by defaults. PWB's default behavior can be extensively customized by changing the values of PWB switches.

Switches fall into three categories:

- Boolean switches. True/false or on/off switches that can also be specified as yes/no or 0/1. An example of a Boolean switch is **Autosave**, which governs whether PWB saves a file when you switch to a different one.
- Numeric switches. An example of a numeric switch is **Undocount**, which determines the maximum number of editing actions you can undo.
- Text switches. Examples of a text switch are **Markfile**, the name of the file in which to store marks, **Tabstops**, a list of tab-stop intervals, and **Readonly**, the operating-system command for PWB to run when saving a read-only file.

To change the setting for Tabstops:

1. From the Options menu, choose Editor Settings.

   PWB displays the Editor Settings dialog box.

2. **Tabstops** is a text switch (not a numeric switch as you might expect), so select the Text option button.

3. Select **Tabstops** in the Switch List box.

   PWB shows the current setting for **Tabstops** in the Switch text box at the top of the dialog box.

4. Move to the Switch text box by clicking in the box or by pressing ALT+S.

   PWB selects only the switch value, instead of the entire text.

5. Type the new setting:

   ```
   3 4 7 8
   ```

   This setting defines a tab stop at columns 4, 8, 15, and every eight columns thereafter. At this point, the Editor Settings dialog box should look like:

6. Choose the Set Switch button to change the setting of the **Tabstops** switch.

7. Choose OK.

Important To change a setting you *must* choose the Set Switch button. The OK button only dismisses the dialog box. It does not perform any other action. This design allows you to set many switches in one session with the dialog box.

The new tab stops you set are used for the current session. If you want to use this setting permanently, you must choose the Save button in the Editor Settings dialog box. This changes your TOOLS.INI file in the same way as for key assignments.

You can make temporary switch assignments for the current session by using the **Assign** function. You do this in the same way as for a key assignment by typing **Arg** *switch*:*value* **Assign** (ALT+A *switch*:*value* ALT+=).

You may be curious about the Switch Owner box that you did not use in this example. The Switch Owner is either PWB or a PWB extension such as PWBHELP (the extension that provides the Microsoft Advisor in PWB). Type or select a switch owner to set switches for that extension. Each extension has its own section in TOOLS.INI.

Note When you choose Set Switch, most switch settings take effect immediately. However, changes to the **Height** switch do not take effect until you choose OK.

Customizing Colors

You can change the color of almost any item in the PWB interface. For a table showing the names and meanings of PWB's color settings, see page 252 in Chapter 7, the "Programmer's WorkBench Reference."

Some displays show a brilliant green for the left and right triangular symbols surrounding buttons in Help.

To change the light green to light cyan:

1. From the Options menu, choose Colors.

   PWB displays the Colors dialog box.

2. Select **Helpitalic** in the Color list box.
3. Select Cyan in the Foreground list box.
4. Choose Set Color.

To verify your change, press F1. The green symbols in help are now light cyan blue. While you are viewing Help, you can find out what parts of PWB the rest of the color names determine. To leave Help, choose the Cancel button or press ESC. PWB returns you to the Colors dialog box.

The Bright Fore and Bright Back check boxes determine if the given color is the usual version of the color or the bright version of the color. Bright black, for example, is usually a dark gray color.

If you want to save your new colors for subsequent sessions, choose the Save button. PWB displays the Save Colors dialog box where you can delete modifications that you don't want to save. When you choose OK in the Save Colors dialog box, PWB modifies TOOLS.INI to record your changes.

Adding Commands to the Run Menu

You can add up to six commands to the Run menu to integrate your own utilities into PWB. A command is the name of any executable (.EXE or .COM) file, batch (.BAT) file, or built-in operating-system command such as DIR or COPY.

Suppose you use an outline processor to keep project notes. You can start the outline processor from PWB's Run menu.

To add a command to the Run menu:

1. From the Run menu, choose Customize Run Menu.
2. Choose the Add button.

   PWB displays the Add Custom Run Menu Item dialog box for you to describe your custom menu item:

3. Type `Project ~Notes...` in the Menu Text box.

   The tilde (~) before the letter N indicates the highlighted access letter for the menu command. The ellipsis (`...`) uses the standard convention to indicate that the command will require more information before it is completed. An ellipsis is commonly associated with a dialog box command but can be used in this context as well.

4. Specify the full path to the outlining program, OUTLINE.EXE, in the Path Name text box. (The program name OUTLINE.EXE is for example purposes only. Substitute the name of your own outliner or other program in its place.)
5. Specify the arguments you want to pass to the outliner in the Arguments text box: `%|dpfF.log`.

   This example illustrates a powerful feature of PWB: its ability to extract parts of the filename to form a new name for customized menu items. The specification `%|dpfF` extracts the drive (`d`), path (`p`), and base name (`f`) of the current file. Anything after `F` is added to the end of the name.

   For example, if the current file is C:\SOURCE\COUNT.ASM, the argument that PWB passes to the program is C:\SOURCE\COUNT.LOG.

6. In the Help Line text box, type the explanatory message that appears on the status bar when you browse this menu item:

   `Run the OUTLINE program`

7. Choose OK to confirm your entries.

PWB adds the command to your Run menu and modifies TOOLS.INI to save the new item. You can now access your outline processor directly from the Run menu.

Note You can add other text processing or word processing programs to the Run menu. If you change the current file using another program, PWB prompts you to update the file or to ignore the changes made by the other program.

How PWB Handles Tabs

The following functions and switches control how PWB handles tabs:

Name	Type	Description
Realtabs	Switch	Determines if PWB preserves tabs on modified lines
Entab	Switch	The white space translation method
Tabalign	Switch	The alignment of the cursor within a tab field
Filetab	Switch	The width of a tab field
Tabdisp	Switch	The fill-character for displaying tab fields
Tab	Function	Moves the cursor to the next tab stop
Backtab	Function	Moves the cursor to the previous tab stop
Tabstops	Switch	Tab positions for Tab and Backtab

For detailed information on each function and switch, see Help or Chapter 7, "Programmer's WorkBench Reference." For instructions on how to set a switch see "Changing Settings" on page 112. For instructions on how to assign a function to a key, see "Changing Key Assignments" on page 109.

To understand how PWB handles tabs, you need to know only a few facts:

- The **Tab** (TAB) and **Backtab** (SHIFT+TAB) cursor-movement functions and the **Tabstops** switch have nothing to do with tab characters. They affect cursor movement, rather than the handling of tab characters, and are not discussed further here. For more information on these items, see Chapter 7, "Programmer's WorkBench Reference."
- PWB never changes any line in your file unless you explicitly modify it (lines longer than PWB's limit of 250 characters are the exception).

  Some text editors translate white space (that is, entab or detab) when they read and write the file. PWB does not translate white space when it reads or writes a file. This is to be compatible with source-code control systems that would detect the translated lines as changed lines.
- PWB translates white space according to the **Entab** switch only when you modify a line.
- **Tabalign** has an effect only when **Realtabs** is set to yes.
- A "tab break" occurs every **Filetab** columns.
- When PWB displays a tab in the file, it fills from the tab character to the next tab break with the **Tabdisp** character.

  Figure 6.1 illustrates how PWB displays tabs.

Figure 6.1 How PWB Displays Tabs

- When translating white space, PWB preserves the exact spacing of text as it is displayed on screen.

To set the width of displayed tabs, change the setting of the **Filetab** switch.

To tell PWB to translate white space on lines that you modify, set the **Realtabs** switch to `no` and the **Entab** switch to a nonzero value, according to the translation method that you want to use. The **Entab** switch takes one of the following values:

Entab	Translation Method
`0`	Translate white space to space characters
`1`	Translate white space outside of quotation-mark pairs to tabs
`2`	Translate white space to tabs

To preserve white space exactly as you type it, set the **Realtabs** switch to `yes` and the **Entab** switch to `0`.

When **Realtabs** is yes, the **Tabalign** switch comes into effect. When **Tabalign** is set to yes, PWB automatically repositions the cursor onto the physical tab character in the file, similar to the way a word processor positions the cursor. When **Tabalign** is set to `no`, PWB allows the cursor to be anywhere in the tab field.

If you want the TAB key to type a tab character, assign the TAB key to the **Graphic** function. Note that when a dialog box is displayed, the TAB key always moves to the next option. You can always use the following method to type a tab character, whether you are in a dialog box or an editing window.

To type a literal tab character in your text or in a dialog box:

1. Execute the **Quote** function (press CTRL+P).
2. Press TAB.

Examples

The following example sets up tabs so that they act the same as in other Microsoft editors, such as QuickC or Word:

```
realtabs:yes
tabalign:yes
graphic:tab
trailspace:yes
entab:0
```

The **Trailspace** switch is needed so that the TAB key will have an effect on otherwise blank lines.

To save your file so that it does not include any actual tab characters (ASCII 9), use the following settings:

```
realtabs:no
entab:0
tabstops:3
```

The **Tabstops** value determines the number of spaces inserted for each press of the tab key.

Another example of a common tab configuration is one in which the TAB key inserts a tab in insert mode but moves over text to the next tab stop when the editor is in overtype mode.

First, use the following tab settings:

```
realtabs:yes
tabalign:yes
```

Then insert the following macro into the PWB section of your TOOLS.INI:

```
;Insert mode and overtype mode tabbing
TabIO:= Insertmode +>over Insertmode "\t" =>  \
        :>over Insertmode Tab
TabIO:TAB
```

For more information on PWB macros see “Writing PWB Macros” on page 98.

PWB Configuration

PWB keeps track of three kinds of information between sessions in these three files:

File	Information Saved
TOOLS.INI	Configuration and customizations, such as key assignments, colors, and macro definitions

CURRENT.STS	The editing environment used most recently
project.STS	The editing and building environment for a project

TOOLS.INI is described in the next section: "The TOOLS.INI File." For more information about CURRENT.STS, see "Current Status File CURRENT.STS" on page 128, and for more information about the *project*.STS files, see "Project Status Files" on page 129.

When you start PWB, it reads the TOOLS.INI file, loads PWB extensions, and reads the CURRENT.STS or project status file in the following order:

1. PWB reads the `[PWB]` section of TOOLS.INI (except when PWB is started using the /D or /DT command-line options). For more information on tagged sections, see "TOOLS.INI Section Tags."

   If the `[PWB]` section contains **Load** switches, PWB loads the specified extension when each switch is encountered. When PWB loads an extension, it also reads the extension's tagged section of TOOLS.INI, if any. For example, when the Help extension is loaded, PWB reads the `[PWB-PWBHELP]` section of TOOLS.INI.
2. PWB autoloads extensions (except when the /D or /DA option is used to start PWB).

   The automatic loading of PWB extensions is described in the next section, "Autoloading Extensions."
3. PWB reads the TOOLS.INI operating-system tagged section (except when /D or /DT is used).
4. PWB reads the CURRENT.STS status file (except when /D or /DS is used to start PWB).
5. PWB reads the TOOLS.INI tagged section for the file extension of the current file (except when /D or /DT is used to start PWB).
6. PWB runs the **Autostart** macro if it is defined in TOOLS.INI (except when /D or /DT is used).

Autoloading Extensions

PWB automatically loads extensions if they follow a specific naming convention and reside in a certain directory. For extensions that follow the convention, it is not necessary to put load statements in TOOLS.INI.

PWB searches the directory where the PWB executable file is located for filenames with the following pattern:

```
PWB*.MXT
```

PWB loads as many extensions with names of this form as it finds. When PWB loads an extension, it also loads the extension's tagged section of TOOLS.INI.

To suppress extension autoloading, use the /DA option on the PWB command line.

Important Do not rename editor extensions. PWB and some extensions may assume the predefined filename.

The TOOLS.INI File

PWB, like other Microsoft tools, stores information in a file called TOOLS.INI. This file retains information about how you want PWB to work under various circumstances. PWB expects to find this file in the directory specified by your INIT environment variable.

TOOLS.INI is a text file. You can edit it using PWB or any other text editor. PWB also can store information directly to TOOLS.INI when, for example, you choose the Save Colors button in the Colors dialog box. PWB modifies this file when you save a recorded macro, a changed switch, a new key assignment, a custom browser database, or a custom project template.

TOOLS.INI Section Tags

The TOOLS.INI file is divided into sections, separated by “tags.” These tags are specified in the form:

[*tagname*]

The *tagname* is the base name of an executable file, such as NMAKE, CVW, or PWB. The tag defines the start of a TOOLS.INI section that contains settings for the indicated tool.

PWB extends this simple syntax to enable you to take different action depending on the operating system or the current file’s extension. The extended syntax is:

[**PWB**-*modifier*]

The modifier can be the base name of a PWB extension, an operating system’s identifier, or a filename extension for files that you edit.

Operating-System Tags

The following table lists the operating-system tags for various operating environments. If you are running the Windows operating system, use the tag for the version of MS-DOS that you are running.

Tag	Operating Environment
[PWB-4.0]	MS-DOS versions 4.0 and 4.01
[PWB-5.0]	MS-DOS version 5.0

Be sure to use the correct version number for your operating system.

Filename-Extension Tags

The operating-system tags are read only once at startup. PWB reads the filename-extension tagged sections each time you switch to a file with that extension. For example, suppose that you want the tab stops for MASM files to be every eight columns, and every five columns for text files.

To set tab options based on filename extension:

1. Open your TOOLS.INI file in an editing window.
2. Create a MASM section by typing the tag:

   ```
   [PWB-.ASM PWB-.INC ]
   ```

3. Create a text file section by typing the tag:

   ```
   [PWB-.TXT]
   ```

4. Put the appropriate **Tabstops**, **Entab**, and **Realtabs** switches in each section. The lines that begin with a semicolon are comments.

   ```
   [PWB-.ASM PWB-.INC
   ; Set the tab stops for MASM to 8
   tabstops : 8
   ; Translate white space to tabs
   entab    : 1
   realtabs : no

   [PWB-.TXT]
   ; Set the tab stops for text files to 5
   tabstops : 5
   ; Translate white space to spaces
   entab    : 0
   realtabs : no
   ```

Depending on whether the current file is a MASM (.ASM or .INC) file or a text (.TXT) file, the tab stops are set at 8 or 5 columns, respectively.

PWB reads multiple sections and applies the appropriate settings. You can use this to your advantage by storing all your general settings in the `[PWB]` section and storing differences in separate tagged sections.

Filename-extension tagged sections are useful for the kinds of files you edit most frequently. However, it's impossible to define settings for every conceivable extension. To handle this case, PWB provides a special extension (`..`) that means "all extensions not defined elsewhere in TOOLS.INI."

For example, to set tab stops to 5 for all files except MASM files, modify the preceding example to use the `[PWB-..]` tag in place of `[PWB-.TXT]`.

Note

When you choose the Save button in the Key Assignments, Editor Settings, and Colors dialog boxes, and when you save a recorded macro or custom Run menu command, PWB saves the setting in the main section. If the setting is for a PWB extension, it is saved in that extension's tagged section. PWB never modifies or writes settings in a filename-extension or operating-system section.

Named Tags

You can define tagged sections of TOOLS.INI that you load manually. Use manually loaded sections to make special key assignments, to load complex or rarely used macros, or to use a special PWB configuration under a particular circumstance.

The syntax for a manually-loaded section tag is:

[**PWB**-*name*]

Where *name* is the name of the tagged section. A single section of TOOLS.INI can be given several tag names. These tags have the form:

[**PWB**-*name1* **PWB**-*name2*...]

When you want to use the settings defined in one of these named sections, pass the name of the section to the **Initialize** function (SHIFT+F8).

To read a tagged section of TOOLS.INI:

- Execute **Arg** *name* **Initialize** (ALT+A *name* SHIFT+F8)

You can use this method to read any tagged section, including the automatically loaded sections.

Note When you execute Initialize with no arguments, PWB clears all the current settings before reading the `[PWB]` section, including settings that you have made for specific PWB extensions. PWB does not reread the operating-system or other additional sections of TOOLS.INI. To reread the main section without clearing other settings that you want to remain in effect, label the main PWB section with the tag `[PWB PWB-main]`. You can then use **Arg `main`** **Initialize** to recover your startup settings, instead of using **Initialize** with no arguments.

TOOLS.INI Statement Syntax

Within each TOOLS.INI section you place a series of comments or statements. Each statement is a macro definition, key assignment, or switch setting, and

must be stated on a single logical line. Statements can be continued across lines by using line-continuations.

General Macro Syntax

The general syntax for a macro definition is:

name := *definition*

PWB does not reserve any names. Therefore, be careful not to redefine a PWB function. For more information about how to write macros, see “Writing PWB Macros” on page 98.

General Key Syntax

The general syntax for a key assignment is:

name : *key*

The *name* is the name of a function or macro, and the *key* is the name of a key. To see how to write a given key, use the **Tell** function as described in “Changing Key Assignments” on page 109.

Note that certain keys have fixed meanings when the cursor is in a dialog box or in the Help window. You can assign one of these keys to a function or macro, but the fixed meaning is used in a dialog box or the Help window.

The following keys have fixed meanings:

Key	Dialog Box	Help Window
ESC	Choose Cancel	Close the Help window
F1	See Help on the dialog box (choose Help)	See Help on the current item
TAB	Move to the next option	Move to the next hyperlink
SHIFT+TAB	Move to the previous option	Move to the previous hyperlink
SPACEBAR	Toggle the setting of the current option	Activate the current hyperlink
ENTER, SHIFT+ENTER, NUMENTER, SHIFT+NUMENTER	Choose the default action	Activate the current hyperlink

Note The Windows operating system or a terminate-and-stay-resident (TSR) program may override PWB's use of specific keys. PWB has no knowledge of keys that are reserved by these external processes. PWB lists these keys as available keys in the Key Assignments dialog box and allows you to assign functions to these keys, but you may not be able to use them. See the documentation for your operating environment to see what keys are reserved by the system.

General Switch Syntax

The general syntax for a switch setting is:

switch : *value*

The exact syntax for the switch value depends on the switch. See Chapter 7, "PWB Reference," for more information about each switch.

Line Continuation

All statements in TOOLS.INI must be stated on a single logical line. A logical line can be written on several physical lines by using the TOOLS.INI line-continuation character, the backslash (\).

The backslash must be preceded by a space to be treated as a line-continuation character. Precede the backslash by two spaces if you want the concatenated statement to contain a space at that location. If the backslash is preceded by a tab, PWB treats the tab as if it were two spaces. The backslash should be the last character on the line except for spaces or tabs.

The backslash in the following statement is *not* a line continuation.

```
Qreplace:CTRL+\
```

However, the backslash at the end of the first line below *is* a line continuation.

```
findtag:=Arg Arg "^\\[^\\]+\\]" Psearch ->nf  \
  Arg Setwindow => :>nf Arg "no tag" Message
```

In this example, the backslash is preceded by two spaces. The first space is included to separate `->nf` from `Arg` in the concatenated macro definition. The second space identifies the backslash that follows it as the line-continuation character.

Comments

In the TOOLS.INI file, PWB treats the text that follows a semicolon (**;**) up to the end of the line as a comment. To specify the beginning of a comment, you must place the semicolon at the beginning of a line or following white space.

For example, the first semicolon in the following statement is part of a command, and the second semicolon begins a comment.

```
Printcmd:lister -t4 %s -c;   ;Print using lister program
```

In the following example, the first semicolon is a key name, and the second semicolon begins a comment.

```
Sinsert:CTRL+; ;Stream insertion: CTRL plus semicolon
```

Semicolons inside a quoted string do not begin a comment.

Environment Variables

The INIT environment variable tells PWB where to find the TOOLS.INI file and where to store the CURRENT.STS file. In general, the INIT, TMP, LIB, INCLUDE, HELPFILES, and PATH environment variables must all be properly set for your development environment to work smoothly.

To set the INIT environment variable from the command line:

- Type `SET INIT=C:\INIT`

The operating-system **SET** command sets the environment variable to contain the string `C:\INIT`. This example presumes that you want to store your initialization files in C:\INIT. You could use any other directory. Make sure that the INIT environment variable lists a single directory. Multiple directories in INIT can cause inconsistent behavior.

The following list outlines how the environment works:

- The environment is always inherited from the parent process. The parent is the process that starts the current process. In MS-DOS, the parent is often COMMAND.COM or the Windows operating system.
- Inheritance of environment variables is a one-way process. A child inherits from its parent. You can make changes to the environment in a child (when you use the Environment Variables command in PWB, for example), but they are not passed back to the parent. This means that any changes to environment variables that you make while shelled out are lost when you return to PWB.
- Each MS-DOS session under the Windows operating system inherits its environment from the Windows operating system. Changes made to the environment in one session do not affect any other session.

The best way to make sure your environment is set properly is to explicitly set it in one of your startup files. These are:

- CONFIG.SYS

- AUTOEXEC.BAT

PWB can save the complete table of environment variables for each project. You can then use the Environment Variables command from the Options menu to change environment variables for individual projects.

If you prefer that PWB save the environment variables for all PWB sessions or use the current operating-system environment when it starts up, change the **Envcursave** and **Envprojsave** switches. For more information on these switches, see the "Programmer's WorkBench Reference" on pages 259 and 260.

Current Status File CURRENT.STS

The first time you run PWB or CodeView, it creates a CURRENT.STS (current status) file in your INIT directory. If there is no INIT directory, PWB and CodeView create the file in the current directory.

CURRENT.STS keeps track of the following items for PWB:

- Open windows, including their size and position and the list of open files in each window
- Screen height
- Window style
- Find string
- Replace string
- The options used in a find or find-and-replace operation, such as the use of regular expressions
- Optionally, all environment variables

PWB and CodeView share the current location and filename for the active window. When you leave CodeView after a debugging session and return to PWB, PWB positions the cursor at the place where you stopped debugging. For more information on the items that CodeView saves in CURRENT.STS, see "The CURRENT.STS State File" on page 316.

The next time you run PWB, it reads CURRENT.STS and restores the editing environment to what it was when you left PWB. For more information on how PWB uses environment variables, see "Environment Variables" on page 127.

The status files are plain text files. You can load one into an editor and read it. However, you might corrupt the file if you try to modify it. There is no need to modify it because PWB keeps it updated for you. No harm occurs if you delete CURRENT.STS. However, you will have to manually reopen the files you were working on.

Project Status Files

For each project, PWB creates a project status file. PWB stores this file in the project directory and gives it the name *project*.STS, where *project* is the base name of the project.

Project status files contain the same kind of information that CURRENT.STS contains, except on a per-project basis. This scheme allows PWB to keep track of your screen layout, file history, and environment variables for each project. The project status files also contain the current project template, language and utility options, build directory, and the program's run-time arguments.

The main difference between the two status files is that the CURRENT.STS file supplies default status information—settings that PWB uses when you have not set a project. PWB uses the project's status file when you open that project.

PWB can also save all environment variables, including PATH, INCLUDE, LIB, and HELPFILES, depending on how the **envcursave** and **envprojsave** switches are set. For more information, see "Environment Variables" on page 127.

Important While it is harmless to delete CURRENT.STS, you should *not* delete project status files. They contain important information for building and updating your project. If you delete a project status file, you may need to delete the project makefile and start over.

CHAPTER 7

Programmer's WorkBench Reference

PWB Command Line

Syntax PWB [[*options*]] [[**/t**]] *files*

Options Use the following case-insensitive options when starting PWB:

/D[[**S**|**T**|**A**]]...
Disables PWB loading the initialization files or PWB extensions as indicated by the following letters:

Letter	Meaning
S	Disable reading the status file CURRENT.STS
T	Disable reading TOOLS.INI
A	Disable PWB extension autoload

The /D option alone disables loading all the PWB extension and initialization files. See: **Autoload**.

Note If you start PWB with the /DT option, this means that PWB options you change during the session cannot be saved.

/PP *makefile*
Opens the specified PWB project.

/PF *makefile*
Opens the specified non-PWB project (foreign makefile).

/PL
Resets the last project. Use this option to start PWB in the same state you last left it. You can set this option as the default by setting the **Lastproject** switch to yes.

/E *command*

Executes the given command or sequence of commands as a macro upon startup.

If *command* contains a space, *command* should be enclosed in double quotation marks (**"**). A single command need not be quoted. If *command* uses literal quotation marks, place a backslash (\) before each mark. To use a backslash, precede it with another backslash.

/R

PWB starts in no-edit mode. You cannot modify files in this mode. See: **Noedit**.

/M {*mark* | *line*}

PWB starts at the specified location. See: **Mark**.

[[[**/T**]] *file*]]...

Tells PWB to load the given files on startup. If you specify a single file, PWB loads it. If you specify multiple files, PWB loads the first file; then when you use File Next or the **Exit** function, PWB loads the next file in the list.

If a /T precedes a filename or wildcard, PWB loads each file as a temporary file. PWB does not include temporary files in the list of files saved between sessions.

Note No other options can follow /T on the PWB command line. You must specify /T for each file you want to be temporary.

PWB Menus and Keys

Many PWB menu commands activate PWB functions or predefined macros. The menu commands that are attached to functions and macros are listed in the tables that follow. To assign a shortcut key for one of these menu commands, use the Key Assignments command on the Options menu and assign a key to the corresponding function or macro. For details on using the Key Assignments dialog box, see “Changing Key Assignments” on page 109.

Names beginning with an underscore (_pwb...) are macros. Names without an underscore are functions.

Table 7.1 File Menu and Keys

Menu Command	Macro or Function	Default Keys
New	_pwbnewfile	Unassigned
Close	_pwbclosefile	Unassigned

Next	_pwbnextfile	Unassigned
Save	_pwbsavefile	SHIFT+F2
Save All	_pwbsaveall	Unassigned

Table 7.1 File Menu and Keys (*continued*)

Menu Command	Macro or Function	Default Keys
DOS Shell	_pwbshell	Unassigned
n file	_pwbfile*n*	Unassigned
Exit	_pwbquit	ALT+F4

Table 7.2 Edit Menu and Keys

Menu Command	Macro or Function	Default Keys
Undo	_pwbundo	Unassigned
Redo	_pwbredo	Unassigned
Repeat	_pwbrepeat	Unassigned
Cut	Delete	SHIFT+DEL, SHIFT+NUM-
Copy	Copy	CTRL+INS, SHIFT+NUM*
Paste	Paste	SHIFT+INS, SHIFT+NUM+
Delete	_pwbclear	DEL
Set Anchor	Savecur	Unassigned
Select To Anchor	Selcur	Unassigned
Stream Mode	_pwbstreammode	Unassigned
Box Mode	_pwbboxmode	Unassigned
Line Mode	_pwblinemode	Unassigned
Record On	_pwbrecord	Unassigned

Table 7.3 Search Menu and Keys

Menu Command	Macro or Function	Default Keys
Log	_pwblogsearch	Unassigned
Next Match (Logging on)	_pwbnextlogmatch	SHIFT+CTRL+F3
Next Match (Logging off)	_pwbnextmatch	Unassigned
Previous Match (Logging on)	_pwbpreviouslogmatch	SHIFT+CTRL+F4
Previous Match (Logging off)	_pwbpreviousmatch	Unassigned
Goto Match	_pwbgotomatch	Unassigned

Table 7.4 Project Menu and Keys

Menu Command	Macro or Function	Default Keys
Compile File	_pwbcompile	Unassigned
Build	_pwbbuild	Unassigned
Rebuild All	_pwbrebuild	Unassigned
Close	_pwbcloseproject	Unassigned
Next Error	_pwbnextmsg	SHIFT+F3
Previous Error	_pwbprevmsg	SHIFT+F4
Goto Error	_pwbsetmsg	Unassigned

Table 7.5 Run Menu and Keys

Menu Command	Macro or Function	Default Keys
command1	_pwbuser1	[ALT+F*n*]
command2	_pwbuser2	[ALT+F*n*]
command3	_pwbuser3	[ALT+F*n*]
command4	_pwbuser4	[ALT+F*n*]
command5	_pwbuser5	[ALT+F*n*]
command6	_pwbuser6	[ALT+F*n*]
command7	_pwbuser7	[ALT+F*n*]
command8	_pwbuser8	[ALT+F*n*]
command9	_pwbuser9	[ALT+F*n*]

Table 7.6 Browse Menu and Keys

Menu Command	Macro or Function	Default Keys
Goto Definition	Pwbrowsegotodef	Unassigned
Goto Reference	Pwbrowsegotoref	Unassigned
View Relationship	Pwbrowseviewrel	Unassigned
List References	Pwbrowselistref	Unassigned
Call Tree (Fwd/Rev)	Pwbrowsecalltree	Unassigned
Function Hierarchy	Pwbrowsefuhier	Unassigned
Module Outline	Pwbrowseoutline	Unassigned
Which Reference	Pwbrowsewhref	Unassigned
Class Tree (Fwd/Rev)	Pwbrowsecltree	Unassigned
Class Hierarchy	Pwbrowseclhier	Unassigned

Table 7.6 Browse Menu and Keys (*continued*)

Menu Command	Macro or Function	Default Keys
Next	Pwbrowsenext	CTRL+NUM+
Previous	Pwbrowseprev	CTRL+NUM-

Table 7.7 Window Menu and Keys

Menu Command	Macro or Function	Default Keys
New	_pwbnewwindow	Unassigned
Close	_pwbclose	CTRL+F4
Close All	_pwbcloseall	Unassigned
Move	_pwbmove	CTRL+F7
Size	_pwbresize	CTRL+F8
Restore	_pwbrestore	CTRL+F5
Minimize	_pwbminimize	CTRL+F9
Maximize	_pwbmaximize	CTRL+F10
Cascade	_pwbcascade	F5
Tile	_pwbtile	SHIFT+F5
Arrange	_pwbarrange	ALT+F5
n file	_pwbwindow*n*	ALT+*n*

Table 7.8 Help Menu and Keys

Menu Command	Macro or Function	Default Keys
Index	_pwbhelp_index	Unassigned
Contents	_pwbhelp_contents	SHIFT+F1
Topic	_pwbhelp_context	F1
Help on Help	_pwbhelp_general	Unassigned
Next	_pwbhelp_again	Unassigned
Search Results	_pwbhelp_searchres	Unassigned

PWB Default Key Assignments

PWB's default keys assignments are shown in table 7.9. In each position having the text `Unassigned`, you can assign a function or macro to that key without taking away a default keystroke. You cannot assign keys for positions that are

empty. These can usually be expressed in a different way. For example, CTRL+{ is expressed as SHIFT+CTRL+[.

Table 7.9 PWB Default Key Assignments

Key	Plain	SHIFT	ALT	CTRL	CTRL+SHIFT
!	—	—	—	—	—
#	Graphic	—	—	—	—
$	Graphic	—	—	—	—
%	Graphic	—	—	—	—
&	Graphic	—	—	—	—
(	Graphic	—	—	—	—
*	Graphic	—	—	—	—
+	Graphic	—	—	—	—
,	Graphic	—	Unassigned	—	—
-	Graphic	—	Unassigned	Unassigned	—
.	Graphic	—	Unassigned	Unassigned	—
/	Graphic	—	Unassigned	Unassigned	—
0	Graphic	—	Unassigned	Unassigned	—
1	Graphic	—	_pwbwindow1	Unassigned	—
2	Graphic	—	_pwbwindow2	Unassigned	—
3	Graphic	—	_pwbwindow3	Unassigned	—
4	Graphic	—	_pwbwindow4	Unassigned	—
5	Graphic	—	_pwbwindow5	Unassigned	—
6	Graphic	—	_pwbwindow6	Unassigned	—
7	Graphic	—	_pwbwindow7	Unassigned	—
8	Graphic	—	_pwbwindow8	Unassigned	—
9	Graphic	—	_pwbwindow9	Unassigned	—
:	Graphic	—	Unassigned	—	Unassigned
;	Graphic	—	Unassigned	Unassigned	—
<	Graphic	—	Unassigned	—	Unassigned
=	Graphic	—	Assign	Unassigned	—
>	Graphic	—	Unassigned	—	Unassigned
@	Graphic	—	—	—	Unassigned
A	Graphic	Graphic	Arg	Mword	Unassigned
B	Graphic	Graphic	(Browse menu)	Unassigned	Unassigned
C	Graphic	Graphic	Unassigned	Ppage	Unassigned
D	Graphic	Graphic	Unassigned	Right	Unassigned

Table 7.9 PWB Default Key Assignments (*continued*)

Key	**Plain**	SHIFT	ALT	CTRL	CTRL+SHIFT
E	Graphic	Graphic	(Edit menu)	Up	Unassigned
F	Graphic	Graphic	(File menu)	Pword	Unassigned
G	Graphic	Graphic	Unassigned	Cdelete	Unassigned
H	Graphic	Graphic	(Help menu)	Unassigned	Unassigned
I	Graphic	Graphic	Unassigned	Unassigned	Unassigned
J	Graphic	Graphic	Unassigned	Sinsert	Unassigned
K	Graphic	Graphic	Unassigned	Unassigned	Unassigned
L	Graphic	Graphic	Unassigned	Replace	Unassigned
M	Graphic	Graphic	Unassigned	Mark	Unassigned
N	Graphic	Graphic	Unassigned	Linsert	Unassigned
O	Graphic	Graphic	(Options menu)	Lasttext	Unassigned
P	Graphic	Graphic	(Project menu)	Quote	Unassigned
Q	Graphic	Graphic	Unassigned	Unassigned	Unassigned
R	Graphic	Graphic	(Run menu)	Mpage	Record
S	Graphic	Graphic	(Search menu)	Left	Sethelp
T	Graphic	Graphic	Unassigned	Tell	Unassigned
U	Graphic	Graphic	Unassigned	Lastselect	Unassigned
V	Graphic	Graphic	Unassigned	Insertmode	Unassigned
W	Graphic	Graphic	(Window menu)	Mlines	Unassigned
X	Graphic	Graphic	Unassigned	Down	Unassigned
Y	Graphic	Graphic	Unassigned	Ldelete	Unassigned
Z	Graphic	Graphic	Unassigned	Plines	Unassigned
[	Graphic	—	Unassigned	Pbal	Unassigned
\	Graphic	—	Unassigned	Qreplace	Unassigned
]	Graphic	—	Unassigned	Setwindow	Unassigned
^	Graphic	—	—	—	Unassigned
_	Graphic	—	—	—	Unassigned
{	Graphic	—	Unassigned	—	—
|	Graphic	—	Unassigned	—	—
}	Graphic	—	Unassigned	—	—
~	Graphic	—	Unassigned	—	Unassigned
F1	_pwbhelp-_context	_pwbhelp-_contents	_pwbhelp_back	Pwbhelpnext	Unassigned
F2	Setfile	_pwbsavefile	Unassigned	Unassigned	Unassigned

Table 7.9 PWB Default Key Assignments (*continued*)

Key	Plain	SHIFT	ALT	CTRL	CTRL+SHIFT
F3	Psearch	_pwbnextmsg	Unassigned	Compile	_pwbnext-logmatch
F4	Msearch	_pwbprevmsg	_pwbquit	_pwbclose	_pwbprevious-logmatch
F5	_pwbcascade	_pwbtile	_pwbarrange	_pwbrestore	Unassigned
F6	Selwindow	_pwb-prevwindow	Unassigned	Winstyle	Unassigned
F7	Execute	Refresh	Unassigned	_pwbmove	Unassigned
F8	Exit	Initialize	Unassigned	_pwbresize	Unassigned
F9	Meta	Shell	Unassigned	_pwbminimize	Unassigned
F10	Openfile	Unassigned	Unassigned	_pwbmaximize	Unassigned
F11	Unassigned	Unassigned	Unassigned	Unassigned	Unassigned
F12	Unassigned	Unassigned	Unassigned	Unassigned	Unassigned
F13	Unassigned	Unassigned	Unassigned	Unassigned	Unassigned
F14	Unassigned	Unassigned	Unassigned	Unassigned	Unassigned
F15	Unassigned	Unassigned	Unassigned	Unassigned	Unassigned
F16	Unassigned	Unassigned	Unassigned	Unassigned	Unassigned
LEFT	Left	Select	Unassigned	Mword	Select
RIGHT	Right	Select	Unassigned	Pword	Select
UP	Up	Select	Unassigned	Mlines	Unassigned
DOWN	Down	Select	Unassigned	Plines	Unassigned
INS	Insertmode	Paste	Unassigned	Copy	Unassigned
DEL	_pwbclear	Delete	Unassigned	Unassigned	Unassigned
HOME	Begline	Select	Unassigned	Begfile	Select
END	Endline	Select	Unassigned	Endfile	Select
ENTER	Emacsnewl	Newline	Unassigned	Unassigned	Unassigned
BKSP	Emacscdel	Emacscdel	Undo	Unassigned	Undo
ESC	Cancel	Unassigned	Unassigned	Unassigned	Unassigned
GOTO	Home	Unassigned	Unassigned	Unassigned	Unassigned
NUM*	Graphic	Copy	Unassigned	Unassigned	Unassigned
NUM+	Graphic	Paste	Unassigned	Pwbrowsenext	Unassigned
NUM-	Graphic	Delete	Unassigned	Pwbrowseprev	Unassigned
NUM/	Graphic	——	Unassigned	Unassigned	Unassigned
NUM-ENTER	Emacsnewl	Newline	Unassigned	Unassigned	Unassigned

Table 7.9 PWB Default Key Assignments (*continued*)

Key	Plain	SHIFT	ALT	CTRL	CTRL+SHIFT
PGUP	Mpage	Select	Unassigned	Unassigned	Select
PGDN	Ppage	Select	Unassigned	Unassigned	Select
TAB	Tab	Backtab	Unassigned	Unassigned	Unassigned

Note on Available Keys

PWB allows you to assign functions and macros to almost any key combination. However, some keys have a fixed meaning in certain circumstances or operating environments. PWB lists these key as available keys in the Key Assignments dialog box, and PWB allows you to assign a command to the key. However, when the circumstance holds, or you are running PWB in a specific environment, certain keys have a fixed meaning that overrides any assignment that you make.

Help Window

In the Help window, the following keys have a fixed meaning:

Key	Meaning
ESC	Close the Help window
TAB	Move to next hyperlink
SHIFT+TAB	Move to previous hyperlink
ENTER	Activate current hyperlink
NUMENTER	Activate current hyperlink
SHIFT+ENTER	Activate current hyperlink
SHIFT+NUMENTER	Activate current hyperlink
SPACE	Activate current hyperlink

Dialog Boxes

In dialog boxes, all keys have predetermined meanings. Your assignments have no effect when a dialog box is displayed. In particular, note the following keys:

Key	Meaning
ESC	Choose Cancel
ENTER	Choose the active command button
F1	Choose Help
TAB	Move to the next option or command
SHIFT+TAB	Move to the previous option or command

Key	Meaning
SPACE	Toggle active option
CTRL+P	When used in a text box, inserts the next key as a literal value. Use this key to type a literal tab character.

The Text Argument dialog box is an exception. All keys except ESC (Cancel) and F1 (Help) have their assigned meaning.

Microsoft Windows

When running PWB with the Windows operating system, some keys are reserved for use by Windows. You can override these reserved keys by setting options in a PIF file.

Key	Default Meaning in the Windows operating system
ALT+ESC	Switch to the next window in the Windows operating system
CTRL+ESC	Switch to the the Windows operating system Task Manager
ALT+TAB	Switch to the next application
ALT+SPACE	Activate the current window's system menu
ALT+ENTER	Shift application between full screen and window

PWB Functions

PWB provides a rich variety of editing, searching, and project-management capabilites in the form of functions. Most of PWB's menus and dialogs call these functions (or macros that use these functions) to perform their actions. You can write your own macros that use these capabilities in ways that precisely suit your needs. You can also execute every function directly, either by pressing a key or by using the **Execute** function.

Table 7.10 summarizes PWB functions. Most functions can be executed in different ways to perform related actions. Complete details are given in the A-to-Z reference that follows the table.

Table 7.10 PWB Functions

Function	Description	Keys
Arg	Begin a function argument	ALT+A
Arrangewindow	Arrange windows or icons	Unassigned
Assign	Define a macro or assign a key	ALT+=
Backtab	Move to previous tab stop	SHIFT+TAB
Begfile	Move to beginning of file	CTRL+HOME
Begline	Move to beginning of line	HOME

Table 7.10 PWB Functions (*continued*)

Function	Description	Keys
Cancel	Cancel arguments or current operation	ESC
Cancelsearch	Cancel background search	Unassigned
Cdelete	Delete character	CTRL+G
Clearmsg	Clear Build Results	Unassigned
Clearsearch	Clear Search Results	Unassigned
Closefile	Close current file	Unassigned
Compile	Compile and build	CTRL+F3
Copy	Copy selection to the clipboard	CTRL+INS, SHIFT+NUM*
Curdate	Today's date (*dd-Mmm-yyyy*)	Unassigned
Curday	Day of week (`Tue`)	Unassigned
Curtime	Current time (*hour*:*minute*:*second*)	Unassigned
Delete	Delete selection	SHIFT+DEL, SHIFT+NUM-
Down	Move down one line	CTRL+X, DOWN
Emacscdel	Delete character	BKSP, SHIFT+BKSP
Emacsnewl	Start a new line	ENTER, NUMENTER
Endfile	Move to end of file	CTRL+END
Endline	Move to end of line	END
Environment	Set or insert environment variable	Unassigned
Execute	Execute macros and functions by name	F7
Exit	Advance to next file or leave PWB	F8
Graphic	Type character	(many)
Home	Move to window corner	GOTO
Information	(Obsolete)	—
Initialize	Reinitialize	SHIFT+F8
Insert	Insert spaces or lines	Unassigned
Insertmode	Toggle insert/overtype mode	CTRL+V, INS
Lastselect	Recover last selection	CTRL+U
Lasttext	Recover last text argument	CTRL+O
Ldelete	Delete lines	CTRL+Y
Left	Move left	CTRL+S, LEFT
Linsert	Insert lines or indent line	CTRL+N
Logsearch	Toggle search logging	Unassigned

Table 7.10 PWB Functions (*continued*)

Function	Description	Keys
Mark	Set, clear, or go to a mark or line number	CTRL+M
Maximize	Enlarge window to full size	Unassigned
Menukey	Activate menu	ALT
Message	Display a message or refresh the screen	Unassigned
Meta	Modify the action of a function	F9
Mgrep	Search across files for text or pattern	Unassigned
Minimize	Shrink window to an icon	Unassigned
Mlines	Scroll down by lines	CTRL+UP, CTRL+W
Movewindow	Move window	Unassigned
Mpage	Move up one page	CTRL+R, PGUP
Mpara	Move up one paragraph	Unassigned
Mreplace	Multifile replace with confirmation	Unassigned
Mreplaceall	Multifile replace	Unassigned
Msearch	Search backward for pattern or text	F4
Mword	Move back one word	CTRL+A, CTRL+LEFT
Newfile	Create a new pseudofile	Unassigned
Newline	Move to the next line	SHIFT+ENTER, SHIFT+NUMENTER
Nextmsg	Go to build message location	Unassigned
Nextsearch	Go to search match location	Unassigned
Noedit	Toggle the no-edit restriction	Unassigned
Openfile	Open a new file	F10
Paste	Insert file or text from clipboard	SHIFT+INS, SHIFT+NUM+
Pbal	Balance paired characters	CTRL+[
Plines	Scroll up by lines	CTRL+DOWN, CTRL+Z
Ppage	Move down one page	CTRL+C, PGDN
Ppara	Move down one paragraph	Unassigned
Print	Print file or selection	Unassigned
Project	Set or clear project	Unassigned
Prompt	Request text argument	Unassigned
Psearch	Search forward for pattern or text	F3
Pwbhelp	Help topic lookup	Unassigned

Table 7.10 PWB Functions (*continued*)

Function	Description	Keys
Pwbhelpnext	Relative help topic lookup	CTRL+F1
Pwbhelpsearch	Global full-text help search	Unassigned
Pwbrowse1stdef	Go to first definition	Unassigned
Pwbrowse1stref	Go to first reference	Unassigned
Pwbrowsecalltree	Browse Call Tree (Fwd/Rev)	Unassigned
Pwbrowseclhier	Browse Class Hierarchy	Unassigned
Pwbrowsecltree	Browse Class Tree (Fwd/Rev)	Unassigned
Pwbrowsefuhier	Browse Function Hierarchy	Unassigned
Pwbrowsegotodef	Browse Goto Definition	Unassigned
Pwbrowsegotoref	Browse Goto Reference	Unassigned
Pwbrowselistref	Browse List References	Unassigned
Pwbrowsenext	Browse Next	CTRL+NUM+
Pwbrowseoutline	Browse Module Outline	Unassigned
Pwbrowsepop	Go to previously browsed location	Unassigned
Pwbrowseprev	Browse Previous	CTRL+NUM-
Pwbrowseviewrel	Browse View Relationship	Unassigned
Pwbrowsewhref	Browse Which Reference?	Unassigned
Pwbwindow	Open a PWB window	Unassigned
Pword	Move forward one word	CTRL+F, CTRL+RIGHT
Qreplace	Replace with confirmation	CTRL+\
Quote	Insert literal key	CTRL+P
Record	Toggle macro recording	SHIFT+CTRL+R
Refresh	Reread or discard file	SHIFT+F7
Repeat	Repeat the last editing operation	Unassigned
Replace	Replace pattern or text	CTRL+L
Resize	Resize window	Unassigned
Restcur	Restore saved position	Unassigned
Right	Move right	CTRL+D, RIGHT
Saveall	Save all modified files	Unassigned
Savecur	Save cursor position	Unassigned
Sdelete	Delete streams	Unassigned
Searchall	Highlight occurrences of pattern or text	Unassigned
Selcur	Select to saved position	Unassigned

Table 7.10 PWB Functions (*continued*)

Function	Description	Keys
Select	Select text	SHIFT+PGUP, SHIFT+CTRL+PGUP, SHIFT+PGDN, SHIFT+CTRL+PGDN, SHIFT+END, SHIFT+CTRL+END, SHIFT+HOME, SHIFT+CTRL+HOME, SHIFT+LEFT, SHIFT+CTRL+LEFT, SHIFT+UP, SHIFT+RIGHT, SHIFT+CTRL+RIGHT, SHIFT+DOWN
Selmode	Change selection mode: box	Unassigned
Selwindow	Move to window	F6
Setfile	Open or change files	F2
Sethelp	Opens, closes, and lists help files	SHIFT+CTRL+S
Setwindow	Adjust file in window	CTRL+]
Shell	Start a shell or run a system command	SHIFT+F9
Sinsert	Insert a stream of blanks or break line	CTRL+J
Tab	Move to the next tab stop	TAB
Tell	Show key assignment or macro definition	CTRL+T
Unassigned	Remove a function assignment from a key	(All unassigned keys)
Undo	Undo and redo editing operations	ALT+BKSP, SHIFT+CTRL+BKSP
Up	Move up	CTRL+E, UP
Usercmd	Execute a custom Run menu command	Unassigned
Window	Move to next or previous window	Unassigned
Winstyle	Add or remove scroll bars	CTRL+F6

Cursor-Movement Commands

PWB provides the following commands to navigate through text. In addition to the commands in the PWB editor, the Source Browser provides powerful commands to navigate through the source of your programs.

Table 7.11 Cursor-Movement Commands

Cursor Movement	Command	Keys
Up one line	Up	UP
Down one line	Down	DOWN
Left one column	Left	LEFT
Right one column	Right	RIGHT
Upper-left corner of window	Home	HOME
Top of window	Meta Up	F9 UP
Bottom of window	Meta Down	F9 DOWN
Leftmost column in window	Meta Left	F9 LEFT
Rightmost column in window	Meta Right	F9 RIGHT
Lower-right corner of window	Meta Home	HOME
Up one window	Mpage	PGUP
Down one window	Ppage	PGDN
Column one	Meta Begline	F9 HOME
One column past window width	Meta Endline	F9 END
Back one word	Mword	CTRL+LEFT
Forward one word	Pword	CTRL+RIGHT
Beginning of line	Begline	HOME
End of line	Endline	END
Next paragraph	Ppara	Unassigned
Previous paragraph	Mpara	Unassigned
End of paragraph	Meta Ppara	F9 Unassigned
End of previous paragraph	Meta Mpara	F9 Unassigned
Beginning of file	Begfile	CTRL+HOME
End of file	Endfile	CTRL+END
To specific line number	Arg *number* Mark	ALT+A *number* CTRL+M
Position before last scroll	Arg Mark	ALT+A CTRL+M
Saved position	Restcur	Unassigned
Named mark	Arg *name* Mark	ALT+A *name* CTRL+M
Scroll window down one line	Mlines	CTRL+UP
Scroll window up one line	Plines	CTRL+DOWN
Scroll window so cursor at top	Arg Plines	ALT+A CTRL+DOWN

Table 7.11 Cursor-Movement Commands (*continued*)

Cursor Movement	Command	Keys
Scroll window so cursor at bottom	Arg Mlines	ALT+A CTRL+UP
Scroll window so cursor at home	Arg Setwindow	ALT+A CTRL+]

Arg

Key ALT+A

Arg
Begin an argument to a function or begin a selection.

After you execute **Arg**, PWB displays `Arg[1]` on the status bar. Each time you execute **Arg**, PWB increments the Arg count.

PWB functions perform variations of their action depending on the Arg count and the "Meta state." You can use the **Meta** and **Arg** function prefixes in any order. See: **Meta**.

To select text or create a function argument:

1. Execute **Arg** (ALT+A).
2. Execute a cursor-movement function.

   Or hold down the SHIFT key and click the left mouse button.

PWB creates a stream, box, or line selection based on the current selection mode. A selection in each of these modes creates a function argument called "streamarg," "boxarg," or "linearg," respectively.

To create a text argument:

1. Execute **Arg** (ALT+A).
2. Type the text of the argument.

When you type the first character of the argument, PWB displays the Text Argument dialog box where you can enter the textarg without modifying your file. The Text Argument dialog box does not have an OK button; instead, you execute the function to which you are passing the text argument. Choose Cancel to save the text and do nothing.

To "pick up" text from a window:

1. Select the text that you want to use in the Text Argument dialog box.
2. Execute **Lasttext** (CTRL+O).

   PWB copies the selected text into the text argument dialog box.

To cancel an argument or selection:

- Execute **Cancel** (ESC).

Returns The return value of **Arg** cannot be tested.

See **Cancel**, **Lastselect**, **Lasttext**, **Meta**, **Prompt**

Arrangewindow

Key Unassigned

Arrangewindow
Cascades all unminimized windows on the desktop. Does not affect minimized windows. See: **_pwbcascade**.

Arg Arrangewindow (ALT+A Unassigned)
Arranges all unminimized windows on the desktop. Does not affect minimized windows. See: **_pwbarrange**.

Meta Arrangewindow (F9 Unassigned)
Tiles up to 16 unminimized windows. Does not affect minimized windows. See: **_pwbtile**.

Meta Arg Arrangewindow (F9 ALT+A Unassigned)
Arranges all icons (minimized windows) on the desktop.

Returns

True	Windows or icons arranged.
False	Nothing to arrange, or more than 16 windows open.

Assign

Key ALT+=

The **Assign** function assigns a function to a keystroke, defines a macro, or sets a PWB switch. You can also assign keys and set switches by using the commands in the Options menu. To see the current assignment for a key or the

definition of a macro, use Options Keys Assignments or the **Tell** function (CTRL+T). See: **Tell**.

Assign
Performs the assignment using the text on the current line. If the line ends with a line continuation, PWB uses the next line, and so on for all continued lines.

Arg Assign (ALT+A ALT+=)
Same as **Assign**, except uses text starting from the cursor.

Arg *textarg* **Assign** (ALT+A *textarg* ALT+=)
Performs the assignment using the specified *textarg*.

Arg *mark* **Assign** (ALT+A *mark* ALT+=)
Performs the assignment using the text from the line at the cursor to the specified mark. The *mark* argument can be either a line number or a previously defined mark name. See: **Mark**.

Arg *boxarg* | *linearg* | *streamarg* **Assign** (ALT+A *boxarg* | *linearg* | *streamarg* ALT+=)
Performs the assignment using the selected text. Ignores blank and comment lines.

Returns

True	Assignment successful.
False	Assignment invalid.

Example

To set the Tabstops switch to 8:

1. Execute **Arg** (ALT+A).
2. Type the following switch assignment:

```
tabstops:8
```

3. Execute **Assign** (ALT+=).

Update

Assign
Arg Assign
With PWB 1.*x*, **Assign** and **Arg Assign** do not recognize line continuations. With PWB 2.00, they use all continued lines for the assignment.

Arg *streamarg* **Assign**
With PWB 1.*x*, a streamarg is not allowed. With PWB 2.00, **Assign** accepts a streamarg.

Arg ? **Assign**
With PWB 1.*x*, this form of the **Assign** function displays the current assignments for all functions, switches, and macros in the "<ASSIGN>Current Assignments and Switch Settings" pseudofile.

With PWB 2.00, the <ASSIGN> pseudofile does not exist; therefore, this form of the **Assign** function is obsolete. If you use this command or execute a macro that executes this command, PWB issues the error:

```
Missing ':' in '?'
```

PWB is expecting an assignment or definition using the name `?`, which is a legal macro name.

Backtab

Key SHIFT+TAB

Backtab
Moves the cursor to the previous tab stop on the line.

Returns

True	Cursor moved.
False	Cursor is at left margin.

Update PWB 2.0 supports variable tab stops. PWB 1.*x* supports only fixed-width tab stops.

See **Tab**, **Tabstops**

Begfile

Key CTRL+HOME

Begfile
Moves the cursor to the beginning of the file.

Returns

True	Cursor moved.
False	Cursor not moved; the cursor is already at the beginning of the file.

See **Endfile**

Begline

Key HOME

Begline
Places the cursor on the first nonblank character in the line.

Meta Begline (F9 HOME)
Places the cursor in the first character position of the line (column one).

Returns

True	Cursor moved.
False	Cursor not moved; the cursor is already at the destination.

Example The following macro moves the cursor to column one, then toggles between column one and the first nonblank character of the line.

```
toggle_begline := Left ->x Meta :>x Begline
```

The result of the **Left** function is tested to determine if the cursor is already in column one. If the cursor is in column one, PWB skips the **Meta** and executes **Begline** to move to the first nonblank character. If the cursor is not in column one, PWB executes **Meta Begline** to move there.

Example This macro mimics the behavior of the BRIEF HOME key:

```
bhome:= Meta Begline +> Home +> Begfile
```

The result of **Meta Begline** (go to column 1 on the line) is tested to determine if the cursor moved. If the cursor moved, the test (+>) succeeds and the macro exits. If the cursor did not move, the cursor is already in column 1, so the macro advances to the home position with **Home**. If the cursor did not move going to the home position, the macro advances to the beginning of the file with **Begfile**.

See **Left**, **Meta**

Cancel

Key ESC

Cancel
Cancels the current selection, argument, or operation. If a message appears on the status bar, the **Cancel** function restores the original contents of the status bar.

If a dialog box or menu is open, **Cancel** closes the dialog box or menu and takes no further action. If Help on a dialog box, menu, or message box is being displayed, **Cancel** closes the Help dialog box.

Returns **Cancel** always returns true.

See **Arg**

Cancelsearch

Key Unassigned

Cancelsearch

Cancels a background search.

The Search Results window contains the partial results of the aborted search and is not flushed. You can browse matches listed in the Search Results by using the Next Match, Previous Match, and Goto Match commands from the Search menu and by using the **Nextsearch** function (Unassigned).

Cancelsearch applies only to multithreaded environments.

Returns

True	Background search was canceled.
False	No background search in progress.

See **Nextsearch**, **_pwbnextlogmatch**, **_pwbpreviouslogmatch**, **_pwbgotomatch**

Cdelete

Key CTRL+G

Cdelete

Deletes the previous character, excluding line breaks. If the cursor is in column 1, **Cdelete** moves the cursor to the end of the previous line.

In insert mode, **Cdelete** deletes the previous character, reducing the line length by 1.

In overtype mode, **Cdelete** deletes the previous character and replaces it with a space character. If the cursor is beyond the end of the line, the cursor moves to the immediate right of the last character on the line.

Emacscdel is similar to **Cdelete**. However, in insert mode, **Emacscdel** deletes line breaks; in overtype mode beyond the end of the line, it does not automatically move to the end of the line.

Returns

True	Cursor moved.
False	Cursor not moved.

See **Delete**, **Emacscdel**, **Ldelete**, **Sdelete**

Clearmsg

Key Unassigned

Clearmsg
Clears the contents of the Build Results window.

Arg Clearmsg (ALT+A Unassigned)
Clears the current set of messages in the Build Results window.

Returns

True	Cleared a message set or the contents of Build Results.
False	The Build Results window is empty.

See **Nextmsg**, **_pwbnextmsg**, **_pwbprevmsg**, **_pwbsetmsg**

Clearsearch

Key Unassigned

Clearsearch
Clears the contents of the Search Results window.

Arg Clearsearch (ALT+A Unassigned)
Clears the current set of matches in the Search Results window.

Returns

True	Cleared a match set or the contents of Search Results.
False	The Search Results window is empty.

See **Clearmsg**, **Logsearch**, **_pwbnextlogmatch**, **_pwbpreviouslogmatch**, **_pwbgotomatch**

Closefile

Key Unassigned

Closefile
Closes the file in the active window. If no files remain in the window's file history, the window is also closed.

Arg Closefile (ALT+A Unassigned)
Closes the file named by the text at the cursor.

Arg *linearg* | *boxarg* | *streamarg* **Closefile**
(ALT+A *linearg* | *boxarg* | *streamarg* Unassigned)
Closes the file named by the selected text.

Arg *textarg* **Closefile** (ALT+A *textarg* Unassigned)
Closes the specified file.

Returns

True	The file was closed.
False	No file was closed.

See **Refresh**, **_pwbclosefile**

Compile

Key CTRL+F3

The **Compile** function compiles and builds targets in the project or runs external commands, capturing the result of the operation in the Build Results window. Under multithreaded environments the commands run in the background.

Arg Compile (ALT+A CTRL+F3)
Compiles the current file. This is equivalent to Project Compile File. **Arg Compile** fails if no project is open. See: **_pwbcompile**.

Arg *textarg* **Compile** (ALT+A *textarg* CTRL+F3)
Builds the target specified by *textarg*. This is equivalent to Build Target command on the Project menu. **Arg** *textarg* **Compile** fails if no project is open.

To build the current project, execute **Arg `all` Compile**.

Arg Meta *textarg* **Compile** (ALT+A *textarg* F9 CTRL+F3)
Rebuilds the specified target and its dependents. See: **_pwbrebuild**.

This command is equivalent to specifying the NMAKE /a option. Note that you can also include NMAKE command-line macro definitions in the text you pass to the **Compile** function.

Arg Meta Compile (ALT+A F9 CTRL+F3)
Aborts the background compile after prompting for confirmation. Also clears the queue of pending background operations (if any).

Arg Arg *textarg* **Compile** (ALT+A ALT+A *textarg* CTRL+F3)
Runs the program or operating-system command specified by *textarg*. The output is displayed in the Compile Results window.

Under multithreaded environments, the program runs in the background, and the Compile Results window is updated as the program executes. Several programs can be queued for background execution.

Do not use this command to execute an interactive program. The program is able to change the display but may not receive input. To run an interactive program, use the **Shell** function (SHIFT+F9).

Returns

True	Operation successfully initiated.
False	Operation not initiated.

Copy

Keys CTRL+INS, SHIFT+NUM*

Menu Edit menu, Copy command

Copy
Copies the current line to the clipboard.

Arg Copy (ALT+A CTRL+INS)
Copies text from the cursor to the end of the line. The text is copied to the clipboard, but the line break is not included.

Arg *boxarg* | *linearg* | *streamarg* **Copy**
(ALT+A *boxarg* | *linearg* | *streamarg* CTRL+INS)
Copies the selected text to the clipboard.

Arg *textarg* **Copy** (ALT+A *textarg* CTRL+INS)
Copies the specified *textarg* to the clipboard.

Arg *mark* **Copy** (ALT+A *mark* CTRL+INS)
Copies the text from the cursor to the mark. The text is copied to the clipboard. The *mark* argument can be either a line number or a previously defined mark. See: **Mark**.

The text is copied as a boxarg or linearg depending on the relative positions of the cursor and the mark. If the cursor and the mark are in the same column, the text is copied as a linearg. If the cursor and the mark are in different columns, the text is copied as a boxarg.

Arg *number* **Copy** (ALT+A *number* CTRL+INS)
Copies the specified number of lines to the clipboard, starting with the current line. For example, **Arg 5 Copy** copies five lines to the clipboard.

Returns **Copy** always returns true.

See **Delete**, **Ldelete**, **Sdelete**, **Paste**

Curdate

Key Unassigned

Curdate
Types the current date at the cursor in the format *day-month-year*, for example: `17-Apr-1999`.

Returns

True	Date typed.
False	Typing the date would make the line too long.

See **Curday**, **Curfile**, **Curfilenam**, **Curfileext**, **Curtime**

Curday

Key Unassigned

Curday
Types the three-letter abbreviation for the current day of the week, as follows: `Mon Tue Wed Thu Fri Sat Sun`.

Returns

True	Day typed.
False	Typing the day would make the line too long.

See **Curdate**, **Curfile**, **Curfilenam**, **Curfileext**, **Curtime**

Curtime

Key Unassigned

Curtime
Types the current time in the format *hours*:*minutes*:*seconds*, for example, `17:08:32`.

Returns

True	Time typed.
False	Typing the time would make the line too long.

See **Curdate**, **Curday**, **Curfile**, **Curfilenam**, **Curfileext**

Delete

Keys SHIFT+DEL, SHIFT+NUM–

Menu Edit menu, Cut command

Delete
Deletes the single character at the cursor, excluding line breaks. It does not copy the deleted character onto the clipboard. Note that the **Delete** function can delete more than one character, depending on the current selection mode.

Arg Delete (ALT+A SHIFT+DEL)
Deletes from the cursor to the end of the line. The deleted text is copied onto the clipboard. In stream selection mode, the deletion includes the line break and joins the current line to the next line.

Arg *boxarg* | *linearg* | *streamarg* **Delete**
(ALT+A *boxarg* | *linearg* | *streamarg* SHIFT+DEL)
Deletes the selected text. The text is copied on to the clipboard.

Meta … **Delete** (F9 … SHIFT+DEL)
As above but discards the deleted text. The contents of the clipboard are not changed.

Returns **Delete** always returns true.

Down

Keys DOWN, CTRL+X

Down
Moves the cursor down one line. If a selection has been started, it is extended by one line. If this movement results in the cursor moving out of the window, the window is adjusted downward as specified by the **Vscroll** switch.

Meta Down (F9 DOWN)
Moves the cursor to the bottom of the window without changing the column position.

Returns

True	Cursor moved.
False	Cursor did not move; the cursor is at the destination.

See **Up**

Emacscdel

Keys BKSP, SHIFT+BKSP

Emacscdel
Deletes the previous character. If the cursor is in column 1, **Emacscdel** moves the cursor to the end of the previous line.

In insert mode, **Emacscdel** deletes the previous character, reducing the length of the line by 1. If the cursor is in column one, **Emacscdel** deletes the line break, joining the current line to the previous line.

In overtype mode, **Emacscdel** deletes the previous character and replaces it with a space character. If the cursor is in column 1, **Emacscdel** moves the cursor to the end of the previous line and does not delete the line break.

Emacscdel is similar to **Cdelete**, but **Cdelete** never deletes line breaks; in overtype mode beyond the end of the line, **Cdelete** automatically moves to the end of the line.

Returns

True	Cursor moved.
False	Cursor not moved.

See **Cdelete**, **Delete**, **Ldelete**, **Sdelete**

Emacsnewl

Keys ENTER, NUMENTER

Emacsnewl
In insert mode, starts a new line. In overtype mode, moves the cursor to the beginning of the next line. PWB automatically positions the cursor on the new line, depending on the setting of the **Softcr** switch.

Returns **Emacsnewl** always returns True.

Update In PWB 1.*x*, PWB performs special automatic indentation for C files. In PWB 2.00, language-specific automatic indentation is handled by language extensions if the feature is enabled. Otherwise, PWB uses its default indentation rules.

See **Newline**, **Softcr**, **C_Softcr**

Endfile

Key CTRL+END

Endfile
Places the cursor at the end of the file.

Returns

True	Cursor moved.
False	Cursor did not move; the cursor is at the end of the file.

See **Begfile**

Endline

Key END

Endline
Moves the cursor to the immediate right of the last character on the line.

Meta Endline (F9 END)
Moves the cursor to the column that is one column past the active window width.

Returns

True	Cursor moved.
False	Cursor did not move; the cursor is at the destination.

See **Begline**, **Traildisp**, **Trailspace**

Environment

Key Unassigned

Environment
Executes the current line as an environment-variable setting.

For example, if the current line contains the following text when you execute **Environment**:

```
PATH=C:\UTIL;C:\DOS
```

PWB adds this setting to the current environment table. The effect is the same as the operating-system SET command. PWB uses the new environment variable for the rest of the session (including shells).

Depending on the settings of the **Envcursave** and **Envprojsave** switches, PWB saves the environment table for PWB sessions and/or projects. See: **Envcursave**, **Envprojsave**.

Arg *textarg* **Environment** (ALT+A *textarg* Unassigned)
Executes the argument as an environment-variable setting.

Arg *linearg* | *boxarg* **Environment** (ALT+A *linearg* | *boxarg* Unassigned)
Executes each selected line or line fragment as an environment-variable setting.

Meta Environment (F9 Unassigned)
Performs environment-variable substitutions for all variables on the current line, replacing each variable with its value.

The syntax for an environment variable isINDEX: Environment variable, specfying in PWB

$(*ENV***)** | **$***ENV***:**

where *ENV* is the uppercase name of the environment variable.

Arg Meta Environment (ALT+A F9 Unassigned)
Performs environment-variable substitutions (described above) for the text from the cursor to the end of the line.

Arg *boxarg* | *linearg* | *streamarg* **Meta Environment**
(ALT+A *boxarg* | *linearg* | *streamarg* F9 Unassigned)
Performs environment-variable substitutions for the selected text.

Returns

True	Environment variable successfully set or substituted.
False	Syntax error or line too long.

Update

Because the <ENVIRONMENT> pseudofile no longer exists, this form of the **Environment** function is obsolete; it is replaced by the Environment command on the Options menu.

Execute

Key

F7

The **Execute** function executes PWB functions and macros by name. It allows you to execute commands that are not assigned to a key or execute a sequence of commands in one step.

The **Execute** function executes the commands by the same rules as macros. Function prompts are suppressed, and you can use the macro flow-control and macro prompt directives. You do not need to define a macro to use these features.

Arg Execute (ALT+A F7)
Executes the text from the cursor to the end of the line as a PWB macro.

Arg *linearg* | *textarg* **Execute** (ALT+A *linearg* | *textarg* F7)
Executes the specified text as a PWB macro.

Returns

True	Last executed function returned true.
False	Last executed function returned false.

Exit

Key

F8

Exit
If you specified multiple files on the PWB command line, PWB advances to the next file. Otherwise, PWB quits and returns control to the operating system.

If the **Autosave** switch is set to yes, the file is saved if it has been modified. If **Autosave** is no and the file is modified, PWB prompts for confirmation to save the file.

Meta Exit (F9 F8)
Performs like **Exit** with the **Autosave** switch set to no, independent of the current setting of **Autosave**. If you have changed any files, PWB asks for confirmation to save before exiting.

Arg Exit (ALT+A F8)
Like **Exit**, except PWB quits immediately without advancing to the next file (if any).

Arg Meta Exit (ALT+A F9 F8)
Like **Meta Exit**, except PWB quits immediately without advancing to the next file.

Returns No return value.

See **_pwbquit**

Graphic

Keys Assigned to most alphanumeric and punctuation keys.

Graphic
Types the character corresponding to the key that you pressed.

Returns

True	The character is typed.
False	Typing the character would make the line too long.

See **Assign**, **Quote**

Home

Key GOTO (Numeric-keypad 5)

Home
Places the cursor in the upper-left corner of the window.

Meta Home (F9 GOTO)
Places the cursor in the lower-right corner of the window.

Returns

True	Cursor moved.
False	Cursor not moved; it is already at the destination.

See **Begline**, **Endline**, **Left**, **Right**

Initialize

Key SHIFT+F8

Initialize
Discards all current settings, including extension settings, then reads the statements from the [PWB] section of TOOLS.INI.

Arg Initialize (ALT+A SHIFT+F8)
Reads the statements from a tagged section of TOOLS.INI. The tag name is specified by the continuous string of nonblank characters starting at the cursor.

Arg *textarg* **Initialize** (ALT+A *textarg* SHIFT+F8)
Reads the statements from the TOOLS.INI tagged section specified by *textarg*.

Example

The section tagged with

```
[PWB-name]
```

is initialized by the command

```
Arg name Initialize
```

Example

To reload the main section of TOOLS.INI without clearing other settings that you want to remain in effect, label the main section of TOOLS.INI with the tag:

```
[PWB PWB-main]
```

then use **Arg** `main` **Initialize** to recover your main settings instead of using **Initialize** with no arguments.

Returns

True	Initialized tagged section in TOOLS.INI.
False	Did not find tagged section in TOOLS.INI.

Information

Update (obsolete)

The PWB 1.*x* **Information** function and its associated pseudofile <INFORMATION-FILE> are obsolete; they do not exist in PWB 2.00.

Insert

Key Unassigned

Insert
Inserts a single-space character at the cursor, independent of the insert/overtype mode.

Arg Insert (ALT+A Unassigned)
Breaks the line at the cursor.

Arg *boxarg* | *linearg* | *streamarg* **Insert**
(ALT+A *boxarg* | *linearg* | *streamarg* Unassigned)
Inserts space characters into the selected area.

Returns

True	Spaces or line break inserted.
False	Insertion would make a line too long.

Example If paragraphs in your file consist of a sequence of lines beginning in the same column and are separated from other paragraphs by at least one blank line, the following macro indents a paragraph to the next tab stop:

```
para_indent:=_pwbboxmode Meta Mpara Down Begline Arg   \
    Meta Ppara Up Begline Tab Insert
```

This macro starts with the predefined PWB macro **_pwbboxmode** to set box selection mode, then creates a box selection from the beginning of the paragraph to the end, one tab stop wide. The **Insert** function inserts spaces in the selection.

See **Sinsert**, **Linsert**

Insertmode

Keys INS, CTRL+V

Insertmode
Toggles between insert mode and overtype mode. If overtype mode is on, the letter O appears on the status bar. The cursor can also change shape, depending on the **Cursormode** switch. See: **Cursormode**.

In insert mode, each character you type is inserted at the cursor. This insertion shifts the remainder of the line one position to the right.

In overtype mode, the character you type replaces the character at the cursor.

Returns

True	PWB is in insert mode.
False	PWB is in overtype mode.

Lastselect

Key CTRL+U

Lastselect

Duplicates the last selection.

The Arg count and Meta state that were previously in effect are not duplicated—only the selection. The new Arg count is one, and the Meta state is the current Meta state. To use a higher Arg count, execute **Arg** (ALT+A). To toggle the Meta state, execute **Meta** (F9).

The re-created selection uses the same pair of *line*:*column* coordinates as the previous selection. Thus, different text can be selected if you have made additions or deletions to the file since the last selection.

See **Arg**, **Lasttext**, **Meta**

Lasttext

Key CTRL+O

Lasttext

Displays the last text argument in the Text Argument dialog box. You can edit the text and then execute any PWB function that accepts a text argument, or you can cancel the dialog box.

If you edit the text and then cancel the dialog box, PWB retains the modified text. Thus, when you execute **Lasttext** again, the new text appears in the dialog box.

Arg [[**Arg**]... [[**Meta**]] **Lasttext** (ALT+A [[ALT+A]]... [[F9]] CTRL+O)

Displays the last text argument in the Text Argument dialog box with the specified Arg count and Meta state.

Arg [[**Arg**]... *linearg* | *boxarg* | *streamarg* [[**Meta**]] **Lasttext**
(ALT+A [[ALT+A]]... *linearg* | *boxarg* | *streamarg* [[F9]] CTRL+O)

Displays the first line of the selection in the Text Argument dialog box with the specified Arg count and Meta state.

Returns The return value of **Lasttext** cannot be tested.

Example The `OpenInclude` macro that follows opens an include file named in the next **#include** directive. The macro demonstrates a technique using the **Lasttext** function to pick up text from the file and modify it without modifying the file or the clipboard.

```
OpenInclude :=  \
    Up Meta Begline Arg Arg "^[ \t]*#[ \t]*include" Psearch ->  \
    Arg Arg "[<>\"]" Psearch -> Right Savecur Psearch ->        \
    Selcur Lasttext Begline "$INCLUDE:" Openfile <n +>          \
    Lastselect Openfile <
```

In the fourth line, **Lasttext** pulls the selected filename into the Text Argument dialog box. The text argument is modified to prepend `$INCLUDE:` before passing
it to the **Openfile** function.

Example In some macro-programming situations, you don't want to use the text immediately. Instead, you need to pick up some text, do some other processing, then use the text. In this situation, use the phrase:

(make selection) **Lasttext Cancel** ...

This picks up the text, then cancels the Text Argument dialog box. The selected text remains in the **Lasttext** buffer for later use. To reuse the text, call **Lasttext** again.

See **Arg**, **Lastselect**, **Meta**, **Prompt**

Ldelete

Key CTRL+Y

Ldelete
Deletes the current line and copies it to the clipboard.

Arg Ldelete (ALT+A CTRL+Y)
Deletes text from the cursor to the end of the line and copies it to the clipboard.

Arg *mark* **Ldelete** (ALT+A *mark* CTRL+Y)
Deletes the text from the line at the cursor to the line specified by *mark* and copies it to the clipboard. The mark cannot be a line number.

Arg *number* **Ldelete** (ALT+A *number* CTRL+Y)

Deletes the specified number of lines starting from the line at the cursor and copies them to the clipboard.

Arg *boxarg* | *linearg* **Ldelete** (ALT+A *boxarg* | *linearg* CTRL+Y)
Deletes the specified text and copies it to the clipboard. The argument is a linearg or boxarg regardless of the current selection mode. The argument is a linearg if the starting and ending points are in the same column.

Meta ... **Ldelete** (F9 ... CTRL+Y)
As above but discards the deleted text. The clipboard is not changed.

Returns **Ldelete** always returns true.

See **Cdelete**, **Delete**, **Emacscdel**, **Sdelete**

Left

Keys LEFT, CTRL+S

Left
Moves the cursor one character to the left. If this movement results in the cursor moving out of the window, the window is adjusted to the left as specified by the **Hscroll** switch.

Meta Left (F9 LEFT)
Moves the cursor to the first column in the window.

Returns

True	Cursor moved.
False	Cursor not moved; the cursor is in column one.

See **Begline**, **Down**, **Endline**, **Home**, **Right**, **Up**

Linsert

Key CTRL+N

Linsert
Inserts one blank line above the current line.

Arg Linsert (ALT+A CTRL+N)
Inserts or deletes blanks at the beginning of a line to move the first nonblank character to the cursor.

Arg *boxarg* | *linearg* **Linsert** (ALT+A *boxarg* | *linearg* CTRL+N)
Inserts blanks within the specified area.

The argument is a linearg or boxarg regardless of the current selection mode. The argument is a linearg if the starting and ending points are in the same column.

Arg *mark* Linsert (ALT+A *mark* CTRL+N)
Like *boxarg* | *linearg* except the specified area is given by the cursor position and the position of the specified mark. The *mark* argument must be a named mark: it cannot be a line number. See: **Mark**.

Returns **Linsert** always returns true.

See **Insert**, **Sinsert**

Logsearch

Key Unassigned

Logsearch
Toggles the search-logging state.

The default search-logging mode when PWB starts up is determined by the **Enterlogmode** switch.

Returns

True	Search logging turned on.
False	Search logging turned off.

Mark

Key CTRL+M

The **Mark** function moves the cursor to a mark or specific location, defines marks, and deletes marks. Note that you cannot set a mark at specific text in a PWB window such as Help; PWB marks only the window position.

If you want to save marks between sessions, assign a filename to the **Markfile** switch or use the Set Mark File command on the Search menu.

Mark (CTRL+M)
Moves the cursor to the beginning of the file.

Arg Mark (ALT+A CTRL+M)
Restores the cursor to its location prior to the last window scroll. Use **Arg Mark** to return to your previous location after a search or other large jump.

Arg *number* **Mark** (ALT+A *number* CTRL+M)
Moves the cursor to the beginning of the line specified by *number* in the current file. Line numbering starts at 1.

Arg *textarg* **Mark** (ALT+A *textarg* CTRL+M)
Moves the cursor to the specified mark.

Arg Arg *textarg* **Mark** (ALT+A ALT+A *textarg* CTRL+M)
Defines a mark at the cursor position. The name of the mark is specified by *textarg*.

Arg Arg *textarg* **Meta Mark** (ALT+A ALT+A *textarg* F9 CTRL+M)
Deletes the specified mark. This form of the **Mark** function always returns true.

Returns

True	Move, definition, or deletion successful.
False	Invalid argument or mark not found.

See **Markfile**, **Restcur**, **Savecur**, **Selcur**

Maximize

Key Unassigned

Maximize
Expands the window to its maximum size. If the window is already maximized, the window is restored.

When the window is maximized and scroll bars are turned off by using the **Winstyle** function, PWB turns off the window borders. This is the "clean screen" look.

Meta Maximize (F9 Unassigned)
Restores the window to its original size.

Returns

True	Window is maximized.
False	Window is restored.

See **Minimize**, **Winstyle**

Menukey

Key ALT

Menukey
Activates the menu bar. Unlike other PWB functions, **Menukey** can be assigned to only one key. It cannot be assigned to a combination of keys.

Returns You cannot test the return value of **Menukey**.

Message

Key Unassigned

Message
Clears the status bar.

Arg Message (ALT+A Unassigned)
Displays the text from the cursor to the end of the line on the status bar.

Arg *textarg* **Message** (ALT+A *textarg* Unassigned)
Displays *textarg* on the status bar.

Meta ... **Message** (F9 ... Unassigned)
As above and also repaints the screen.

Returns **Message** always returns true.

Example The following macro is useful when writing new macros (the `!` is the macro name):

```
! := Meta Message
```

With this definition you can place an exclamation point in your macros wherever you want a screen update. If you also want to display a status-bar message at the time of the update, use the phrase:

... Arg "*text of message*" ! ...

See **Prompt**

Meta

Key F9

Meta
Modifies the action of the function it prefixes.

When the Meta state is turned on, the letter A (for "Alternate") appears in the status bar. You can use the **Meta** and **Arg** function prefixes in any order.

Returns

True	Meta state turned on.
False	Meta state turned off.

See **Arg**, **Lasttext**, **Lastselect**

Mgrep

Key Unassigned

The **Mgrep** function searches all the files listed in the **Mgreplist** macro. PWB places all matches in the Search Results window. Under multithreaded environments, PWB performs the search in the background.

To browse the list of matches, use **_pwbnextlogmatch** (CTRL+SHIFT+F3), **_pwbpreviouslogmatch** (CTRL+SHIFT+F4), and the **Nextsearch** function (Unassigned).

Mgrep (Unassigned)
Searches for the previously searched string or pattern.

Arg Mgrep (ALT+A Unassigned)
Searches for the string specified by the characters from the cursor to the first blank character.

Arg *textarg* **Mgrep** (ALT+A *textarg* Unassigned)
Searches for *textarg*.

Arg Arg Mgrep (ALT+A ALT+A Unassigned)
Searches for the regular expression specified by the characters from the cursor to the first blank character.

Arg Arg *textarg* **Mgrep** (ALT+A ALT+A *textarg* Unassigned)
Searches for the regular expression specified by *textarg*.

Meta ... **Mgrep** (F9 ... Unassigned)
As above except that the value of the **Case** switch is reversed for the search.

Returns

True
With MS-DOS, indicates that a match was found. With multithreaded environments, indicates that a background search was successfully initiated.

False
No matches, no search pattern specified, search pattern invalid, or search terminated by CTRL+BREAK.

Update

In PWB 2.00, search and build results and their browsing functions are separate. A background build operation and a background search can be performed simultaneously.

In PWB 1.*x*, search and build results appear in the same window, and are browsed with the same commands. A background build operation and a multifile search cannot be performed at the same time in PWB 1.*x*.

Minimize

Key

Unassigned

Minimize
Shrinks the active window to an icon (a minimized window). If the window is already minimized, restores the window.

Arg Minimize (ALT+A Unassigned)
Minimizes all open windows.

Meta Minimize (F9 Unassigned)
Restores the window to its unminimized state.

Returns

True	Window minimized: the window is an icon.
False	Window restored: the window is not an icon.

See **Maximize**

Mlines

Keys CTRL+UP, CTRL+W

Mlines
Scrolls the window down as specified by the **Vscroll** switch.

Arg Mlines (ALT+A CTRL+UP)
Scrolls the window so the line at the cursor moves to the bottom of the window.

Arg *number* **Mlines** (ALT+A *number* CTRL+UP)
Scrolls the window down by *number* lines.

Returns

True	Window scrolled.
False	Invalid argument.

See **Plines**

Movewindow

Key Unassigned

Movewindow
Enters window-moving mode. In window-moving mode, only the following actions are available:

Action	Key
Move up one row	UP
Move down one row	DOWN
Move left one column	LEFT
Move right one column	RIGHT
Accept the new position	ENTER
Cancel the move	ESC

Arg *number* **Movewindow** (ALT+A *number* Unassigned)
Moves the upper-left corner of the window to the screen row specified by *number*.

Meta Arg *number* **Movewindow** (F9 ALT+A *number* Unassigned)
Moves the upper-left corner of the window to the screen column specified by *number*.

Returns

True	Window moved.
False	Window not moved.

Mpage

Keys PGUP, CTRL+R

Mpage
Moves the cursor backward in the file by one window.

Returns

True	Cursor moved.
False	Cursor not moved.

See **Ppage**

Mpara

Key Unassigned

Mpara
Moves the cursor to the beginning of the first line of the current paragraph. If the cursor is already on the first line of the paragraph, it is moved to the begining of the first line of the preceding paragraph.

Meta Mpara (F9 Unassigned)
Moves the cursor to the first blank line preceding the current paragraph.

Returns

True	Cursor moved.
False	Cursor not moved; no more paragraphs in the file.

See **Ppara**

Mreplace

Key Unassigned

Mreplace
Performs a find-and-replace operation across multiple files, prompting for the find-and-replacement strings and for confirmation at each occurrence. **Mreplace** searches all the files listed in the special macro **Mgreplist**.

Arg Arg Mreplace (ALT+A ALT+A Unassigned)
Performs the same action as **Mreplace** but uses regular expressions.

Meta ... **Mreplace** (F9 ... Unassigned)
As above except reverses the sense of the **Case** switch for the operation.

Returns

True	At least one replacement made.
False	No replacements made or operation aborted.

See **Mgrep**, **Mreplaceall**, **Qreplace**, **Replace**

Mreplaceall

Key Unassigned

Mreplaceall
Performs a find-and-replace operation across multiple files, prompting for the find-and-replacement strings. **Mreplaceall** searches all the files listed in the special macro **Mgreplist**.

Arg Arg Mreplaceall (ALT+A ALT+A Unassigned)
Performs the same action as **Mreplaceall** but uses regular expressions.

Meta ... **Mreplaceall** (F9 ... Unassigned)
As above except reverses the sense of the **Case** switch for the operation.

Returns

True	At least one replacement made.
False	No replacements made or operation aborted.

See **Mgrep**, **Mreplace**, **Qreplace**, **Replace**

Msearch

Key F4

Msearch
Searches backward for the previously searched string or pattern.

Arg Msearch (ALT+A F4)
Searches backward for the string specified by the text from the cursor to the first blank character.

Arg *textarg* **Msearch** (ALT+A *textarg* F4)
Searches backward for the specified text.

Arg Arg Msearch (ALT+A ALT+A F4)
Searches backward for the regular expression specified by the text from the cursor to the first blank character.

Arg Arg *textarg* **Msearch** (ALT+A ALT+A *textarg* F4)
Searches backward for the regular expression defined by *textarg*.

Meta ... **Msearch** (F9 ... F4)
As above except reverses the sense of the **Case** switch for the search.

Returns

True	String found.
False	Invalid argument, or string not found.

See **Mgrep**, **Psearch**

Mword

Keys CTRL+LEFT, CTRL+A

Mword
Moves the cursor to the beginning of the current word, or if the cursor is not in a word or at the beginning of the word, moves the cursor to the beginning of the previous word. A word is defined by the **Word** switch.

Meta Pword (F9 CTRL+RIGHT)
Moves the cursor to the immediate right of the previous word.

Returns

True	Cursor moved.
False	Cursor not moved; there are no more words in the file.

See **Pword**

Newfile

Key Unassigned

The **Newfile** function creates a new pseudofile. If the **Newwindow** switch is set to yes, it opens a new window for the file.

Newfile (Unassigned)
Creates a new untitled pseudofile. The new pseudofile is given a unique name of the form:

<Untitled.*nnn*>Untitled.*nnn*

where *nnn* is a three-digit number starting with 001 at the beginning of each PWB session. The window title shows `Untitled.001`. Use the pseudofile name `<Untitled.001>` to refer to the file in a text argument or dialog box.

Arg Newfile (ALT+A Unassigned)
Creates a new pseudofile with the name specified by the text from the cursor to the end of the line. The resulting full pseudofile name is:

"<Text on the line>Text on the line"

Arg *textarg* **Newfile** (ALT+A *textarg* Unassigned)
Creates a new pseudofile with the name specified by *textarg*. The resulting full pseudofile name is:

"<*textarg*>*textarg*"

If you want to use a different short name and window title, use the full name as an argument to the **Setfile** or **Openfile** functions. For example, **Arg** `"<temp>Temporary File"` **Openfile** opens a pseudofile in a new window that has the title `Temporary File`.

Returns

True	Successfully created the pseudofile.
False	Unable to create the pseudofile.

Newline

Keys SHIFT+ENTER, SHIFT+NUMENTER

Newline
Moves the cursor to a new line.

If the **Softcr** switch is set to yes, PWB automatically indents to an appropriate position based on the type of file you are editing.

Meta Newline (F9 SHIFT+ENTER)
Moves the cursor to column 1 of the next line.

Returns Newline always returns true.

Update In PWB 1.*x*, PWB performs special automatic indentation for C files. In PWB 2.00, language-specific automatic indentation is handled by language extensions if the feature is enabled. Otherwise, PWB uses its default indentation rules.

See **Emacsnewl**

Nextmsg

Key Unassigned

Nextmsg
Advances to next message in the Build Results window.

Arg *number* **Nextmsg** (ALT+A *number* Unassigned)
Moves to the *n*th message in the current set of messages, where *n* is specified by *number*.

To move relative to the current message, use a signed number. For example, when *number* is +1, PWB moves to the next message, and when it is –1, PWB moves to the previous message.

Arg Nextmsg (ALT+A Unassigned)
Moves to the next message in the current set of messages that does not refer to the current file.

Meta Nextmsg (F9 Unassigned)
Advances to the next set of messages.

Arg Arg Nextmsg (ALT+A ALT+A Unassigned)
Sets the message at the cursor as the current message. This works only when the cursor is on a message in the Build Results window.

Returns

True	Message found.
False	No more messages found.

Update In PWB 1.*x*, **Nextmsg** also browses the results of searches. In PWB 2.00, search results are browsed with the **Nextsearch** function.

Meta Nextmsg
In PWB 1.*x*, deletes the current set of messages and advances to the next set. In PWB 2.00, **Meta Nextmsg** does not delete the set. To delete sets of messages in PWB 2.00, use the **Clearmsg** function.

Meta Arg Arg Nextmsg
In PWB 1.*x*, closes the Compile Results window. In PWB 2.00, it behaves like **Arg Arg Nextmsg**.

See Clearmsg

Nextsearch

Key Unassigned

Nextsearch
Advances to the next match in the Search Results window.

Arg *number* **Nextsearch** (ALT+A *number* Unassigned)
Moves to the *n*th match in the current set of matches, where *n* is specified by *number*.

To move relative to the current match, use a signed number. For example, when *number* is +1, PWB moves to the next match, and when it is 1, PWB moves to the previous match.

Arg Nextsearch (ALT+A Unassigned)
Moves to the next match in the current set of matches that does not refer to the current file.

Meta Nextsearch (F9 Unassigned)
Advances to the next set of matches.

Arg Arg Nextsearch (ALT+A ALT+A Unassigned)
Sets the match at the cursor as the current match. This works only when the cursor is on a match in the Search Results window.

Update In PWB 1.*x*, the results of searches are browsed using the **Nextmsg** function.

See **Clearsearch**

Noedit

Key Unassigned

The **Noedit** function toggles the no-edit state of PWB or the current file. When the no-edit state is turned on, PWB displays the letter R on the status bar and disallows modification of the file.

Noedit
Toggles the no-edit state. If you started PWB with the /R (read-only) option, **Noedit** removes the no-edit limitation.

Meta Noedit (F9 Unassigned)
Toggles the no-edit state for the current file. This form of the **Noedit** command works only for disk files and has no effect on pseudofiles.

If you have the **Editreadonly** switch set to no, PWB turns on the no-edit state for files that are marked read-only on disk. This function toggles the no-edit state for the file so that you can modify it.

Returns

True	File or PWB in no-edit state; modification disallowed.
False	File or PWB not in no-edit state; modification allowed.

Openfile

Key F10

The **Openfile** function opens a file in a new window, ignoring the **Newwindow** switch.

Arg Openfile (ALT+A F10)
Opens the file at the cursor in a new window. The name of the file is specified by the text from the cursor to the first blank character.

Arg *textarg* **Openfile** (ALT+A *textarg* F10)
Opens the specified file in a new window.

If the argument is a wildcard, PWB creates a pseudofile containing a list of files that match the pattern. To open a file from this list, position the cursor at the beginning of the name and use **Arg Openfile** or **Arg Setfile**.

Returns

True	File and window successfully opened.
False	No argument specified, or file did not exist and you did not create it.

See **Newfile**, **Setfile**

Paste

Keys SHIFT+INS, SHIFT+NUM+

Menu Edit menu, Paste command

Paste (SHIFT+INS)
Copies the contents of the clipboard to the file at the cursor. The text is always inserted independent of the insert/overtype mode.

If the clipboard contents were copied to the clipboard as a *linearg*, PWB inserts the contents of the clipboard above the current line. Otherwise, the contents of the clipboard are inserted at the cursor.

Arg *boxarg* | *linearg* | *streamarg* **Paste**
(ALT+A boxarg | linearg | streamarg SHIFT+INS)
Replaces the selected text with the contents of the clipboard.

Arg Paste (ALT+A SHIFT+INS)
Copies the text from the cursor to the end of the line. The text is copied to the clipboard and inserted at the cursor.

Arg *textarg* **Paste** (ALT+A *textarg* SHIFT+INS)
Copies *textarg* to the clipboard and inserts it at the cursor.

Arg Arg *filename* **Paste** (ALT+A ALT+A *filename* SHIFT+INS)
Copies the contents of the file specified by *textarg* to the current file above the current line.

Arg Arg !*textarg* **Paste** (ALT+A ALT+A **!***filename* SHIFT+INS)
Runs *textarg* as an operating-system command, capturing the command's output to standard output. The output is copied to the clipboard and inserted above the current line.

You must enter the exclamation mark as shown.

Returns

True	**Paste** always returns true except for the following cases.
False	Tried **Arg Arg** *filename* **Paste** and file did not exist, or the pasted text would make a line too long.

Example

The following command copies a sorted copy of the file SAMPLE.TXT to the current file: **Arg Arg `!SORT <SAMPLE.TXT` Paste** (ALT+A ALT+A `!SORT <SAMPLE.TXT` SHIFT+INS).

Pbal

Key

CTRL+[

Pbal
Scans backward through the file, balancing parentheses (**( )**) and brackets (**[ ]**). The first unmatched parenthesis or bracket is highlighted when found.

If an unbalanced parenthesis or bracket is found, it is highlighted and the corresponding character is inserted at the cursor. If no unbalanced characters are found, PWB displays a message box.

The search does not include the cursor position and looks for more opening brackets or parentheses than closing ones.

Arg Pbal (ALT+A CTRL+[)
Like **Pbal** except that it scans forward through the file and searches for right brackets or parentheses lacking opening partners.

Meta Pbal (F9 CTRL+[)
Like **Pbal** but does not insert the unbalanced character. If no unbalanced characters are found, moves to the matching character.

Arg Meta Pbal (ALT+A F9 CTRL+[)
Like **Arg Pbal** but does not insert the character. If no unbalanced characters are found, moves to the matching character.

Update
In PWB 1.*x*, the messages appear on the status bar. In PWB 2.00, they appear in a message box.

Returns

True	Balance successful.
False	Invalid argument, or no unbalanced characters found.

See **Infodialog**

Plines

Keys CTRL+DOWN, CTRL+Z

Plines
Scrolls the text up as specified by the **Vscroll** switch.

Arg Plines (ALT+A CTRL+DOWN)
Scrolls the text such that the line at the cursor is moved to the top of the window.

Arg *number* **Plines** (ALT+A *number* CTRL+DOWN)
Scrolls the text up by *number* lines.

Returns

True	Text scrolled.
False	Invalid argument.

See **Mlines**

Ppage

Keys PGDN, CTRL+C

Ppage
Moves the cursor forward in the file by one window.

Returns

True	Cursor moved.
False	Cursor not moved.

See **Mpage**

Ppara

Key Unassigned

Ppara
Moves the cursor to the beginning of the first line of the next paragraph.

Meta Ppara (F9 Unassigned)
Moves cursor to the beginning of the first blank line after the current paragraph. If the cursor is not on a paragraph, moves the cursor to the first blank line after the next paragraph.

Returns

True	Cursor moved.
False	Cursor not moved; no more paragraphs in the file.

See **Mpara**

Print

Key Unassigned

The **Print** function prints files or selections. If the **Printcmd** switch is set, PWB uses the command line given in the switch. Otherwise, PWB copies the file or selection to PRN. Under multithreaded environments, PWB runs the print command in the background.

Print (Unassigned)
Prints the current file.

Arg *textarg* **Print** (ALT+A *textarg* Unassigned)
Prints all the files listed in *textarg*. Use a space to separate each name from the preceding name. You can use environment variables to specify paths for the files.

Arg *boxarg* | *linearg* | *streamarg* **Print**
(ALT+A *boxarg* | *linearg* | *streamarg* Unassigned)
Prints the selected text.

Arg Meta Print (ALT+A F9 Unassigned)
Cancels the current background print.

Returns

True	Print successfully submitted.
False	Could not start print job.

Update In PWB 1.*x* there is no way to cancel a background print.

Project

Key Unassigned

Project
Open the last project.

Arg Project (ALT+A Unassigned)
Open the project makefile at the cursor as a PWB project. The name of the project is specified by the text from the cursor to the first blank character.

Arg *textarg* **Project** (ALT+A *textarg* Unassigned)
Open the project makefile specified by *textarg* as a PWB project.

Arg Arg Project (ALT+A ALT+A Unassigned)
Close the current project.

Arg Meta Project (ALT+A F9 Unassigned)
Open the project makefile at the cursor as a non-PWB project (foreign makefile).

Arg *textarg* **Meta Project** (ALT+A *textarg* F9 Unassigned)
Open the project makefile specified by *textarg* as a non-PWB project.

Returns

True	A project is open.
False	A project is not open.

See **Lastproject**

Prompt

Key Unassigned

The **Prompt** function displays the Text Argument dialog box where you can enter a text argument. You can use this function interactively, but because it is mainly useful in macros, it is not assigned to a key by default. You usually use **Lasttext** or **Arg** to directly enter a text argument.

Prompt
Displays the Text Argument dialog box without a title. See: **Lasttext**

Arg Prompt (ALT+A Unassigned)
Uses the text of the current line from the cursor to the end of the line as the title.

Arg *textarg* **Prompt** (ALT+A *textarg* Unassigned)
Uses *textarg* as the title.

Arg *boxarg* | *linearg* | *streamarg* **Prompt**
(ALT+A *boxarg* | *linearg* | *streamarg* Unassigned)
Uses the selected text as the title. If the selection spans more than one line, the title is the first line of the selected text.

Returns

True	Textarg entered; the user chose the OK button.
False	The dialog box was canceled.

Example

With the following macro, PWB prompts for a Help topic:

```
QueryHelp := Arg "Help Topic to Find:" Prompt -> Pwbhelp
QueryHelp : Ctrl+Q
```

When you press CTRL+Q, PWB displays a dialog box with the string `Help Topic to Find:` as the title and waits for a response. PWB passes your response to the **Pwbhelp** function as if the command **Arg** *textarg* **Pwbhelp** had been executed.
If you cancel the dialog box, **Prompt** returns false and the macro conditional `->` terminates the macro without executing **Pwbhelp**.

See

Assign

Psearch

Key

F3

Psearch
Searches forward for the previously searched string or pattern.

Arg Psearch (ALT+A F3)
Searches forward in the file for the string specified by the text from the cursor to the first blank character.

Arg *textarg* **Psearch** (ALT+A *textarg* F3)
Searches forward for the specified text.

Arg Arg Psearch (ALT+A ALT+A F3)
Searches forward in the file for the regular expression specified by the text from the cursor to the first blank character.

Arg Arg *textarg* **Psearch** (ALT+A ALT+A *textarg* F3)
Searches forward for the regular expression defined by *textarg*.

Meta … **Psearch** (F9 … F3)
As above but reverses the value of the **Case** switch for one search.

Returns

True	String found.
False	Invalid argument, or string not found.

Pwbhelp

Key Unassigned

Pwbhelp
Displays the default Help topic.

Arg Pwbhelp (ALT+A Unassigned)
Displays Help on the topic at the cursor. Equivalent to the macro **_pwbhelp_context** (F1).

Arg *textarg* **Pwbhelp** (ALT+A *textarg* Unassigned)
Displays Help on the specified text argument.

Arg *streamarg* **Pwbhelp** (ALT+A *streamarg* Unassigned)
Displays Help on the selected text. The selection cannot include more than one line.

Meta Pwbhelp (F9 Unassigned)
Prompts for a key, then displays Help on the function or macro assigned to the key you press.

If you press a key that is not assigned to a function or macro, PWB displays help on the **Unassigned** function. If you press a key that PWB does not recognize, the prompt remains displayed until you press a key that PWB recognizes.

Returns

True	Help topic found.
False	Help topic not found.

Pwbhelpnext

Key CTRL+F1

Pwbhelpnext
Displays the next physical topic in the current Help database.

Meta Pwbhelpnext (F9 CTRL+F1)
Displays the previous Help topic on the backtrace list. This is the Help topic that you previously viewed. Up to 20 Help topics are retained in the backtrace list.

Equivalent to the Back button on the Help screens and the macro **_pwbhelp_back** (ALT+F1).

Arg Pwbhelpnext (ALT+A CTRL+F1)
Displays the next occurrence of the current Help topic within the Help system.

Equivalent to the macro **_pwbhelp_again** (Unassigned).

Use this command when the Help topic appears several times in the set of open Help databases.

Returns

True	Help topic found.
False	Help topic not found.

Pwbhelpsearch

Key Unassigned

The **Pwbhelpsearch** function performs a global search of the Help system. The search is case insensitive unless you use the **Meta** form of **Pwbhelpsearch**, which uses the setting of the **Case** switch to determine case sensitivity.

Pwbhelpsearch (Unassigned)
Displays the results of the last global Help search.

Equivalent to the predefined macro **_pwbhelp_searchres** (Unassigned).

Arg Pwbhelpsearch (ALT+A Unassigned)
Searches Help for the word at the cursor.

Arg *textarg* **Pwbhelpsearch** (ALT+A *textarg* Unassigned)
Searches Help for the selected text.

Arg Arg Pwbhelpsearch (ALT+A ALT+A Unassigned)
Searches Help using the regular expression at the cursor.

Arg Arg *textarg* **Pwbhelpsearch** (ALT+A ALT+A *textarg* Unassigned)
Searches Help for the selected regular expression.

Meta ... **Pwbhelpsearch** (F9 ... Unassigned)
As above except the search is case sensitive if the **Case** switch is set to yes.

Returns

True	At least one match found.
False	No matches found, or search canceled.

Pwbrowse Functions

Most of the **Pwbrowse**... functions provided by the PWBROWSE Source Browser extension display one of the Source Browser's dialog boxes. The Source Browser functions attached to Browse menu commands are listed in the following table.

Function	Browse Menu Command	Key
Pwbrowsecalltree	Call Tree (Fwd/Rev)	Unassigned
Pwbrowseclhier	Class Hierarchy	Unassigned
Pwbrowsecltree	Class Tree (Fwd/Rev)	Unassigned
Pwbrowsefuhier	Function Hierarchy	Unassigned
Pwbrowsegotodef	Goto Definition	Unassigned
Pwbrowsegotoref	Goto Reference	Unassigned
Pwbrowselistref	List References	Unassigned
Pwbrowsenext	Next	CTRL+NUM+
Pwbrowseoutline	Module Outline	Unassigned
Pwbrowseprev	Previous	CTRL+NUM-
Pwbrowseviewrel	View Relationship	Unassigned
Pwbrowsewhref	Which Reference	Unassigned

The browser functions in the following table do not correspond to a Browse menu command.

Function	Description	Key
Pwbrowse1stdef	Go to 1st definition	Unassigned
Pwbrowse1stref	Go to 1st reference	Unassigned
Pwbrowsepop	Go to previously browsed location	Unassigned

Pwbwindow

Key Unassigned

The **Pwbwindow** function opens PWB windows. If the specified window is already open, PWB switches to that window.

Arg Pwbwindow (ALT+A Unassigned)
Opens the PWB window with the name at the cursor. The name is specified by the text from the cursor to the first blank character.

Arg *textarg* **Pwbwindow** (ALT+A *textarg* Unassigned)
Opens the specified PWB window.

Arg Meta Pwbwindow (ALT+A F9 Unassigned)
Closes the PWB window specified by the name at the cursor.

Arg *textarg* **Meta Pwbwindow** (ALT+A *textarg* F9 Unassigned)
Closes the specified PWB window.

Returns

True	The specified window was opened.
False	The window could not be opened.

Pword

Keys CTRL+RIGHT, CTRL+F

Pword
Moves the cursor to the beginning of the next word. A word is defined by the **Word** switch.

Meta Pword (F9 CTRL+RIGHT)
Moves the cursor to the immediate right of the current word, or if the cursor is not in a word, moves it to the right of the next word.

Returns

True	Cursor moved.
False	Cursor not moved; there are no more words in the file.

See **Mword**

Qreplace

Key CTRL+\

The **Qreplace** function performs a find-and-replace operation on the current file, prompting for find-and-replacement strings and confirmation at each occurrence.

Qreplace (CTRL+\)
Performs the replacement from the cursor to the end of the file, wrapping around the end of the file if the **Searchwrap** switch is set to yes.

Arg *boxarg* | *linearg* | *streamarg* **Qreplace**
(ALT+A *boxarg* | *linearg* | *streamarg* CTRL+\)
Performs the replacement over the selected area.

Note that PWB does not adjust the selection at each replacement for changes in the length of the text. For *boxarg* and *streamarg*, PWB may replace text that was not included in the original selection or miss text included in the original selection.

Arg *mark* **Qreplace** (ALT+A *mark* CTRL+\)
Performs the replacement on text from the cursor to the specified mark. Replaces over text as if it were selected, according to the current selection mode. The *mark* argument cannot be a line number. See: **Mark**.

Arg *number* **Qreplace** (ALT+A *number* CTRL+\)
Performs the replacement for the specified number of lines, starting with the line at the cursor.

Arg Arg … **Qreplace** (ALT+A ALT+A … CTRL+\)
As above except using regular expressions.

Meta … **Qreplace** (F9 … CTRL+\)
As above except the sense of the **Case** switch is reversed for the operation.

Returns

True	At least one replacement was performed.
False	String not found, or invalid pattern.

See **Mreplace**, **Replace**, **Searchwrap**

Quote

Key CTRL+P

Quote
Reads one key from the keyboard and types it into the file or dialog box. In a dialog box, the key is always CTRL+P, no matter what function or macro you may have assigned to CTRL+P for the editor.

This is useful for typing a character (such as TAB or CTRL+L) whose keystroke is assigned to a PWB function.

Returns

True	**Quote** always returns true except in the following case.
False	Character would make line too long.

Record

Key SHIFT+CTRL+R

The **Record** function toggles macro recording. While a macro is being recorded, PWB displays the letter X on the status bar, and a bullet appears next to the Record On command from the Edit menu. If a menu command cannot be recorded, it is disabled while recording.

When macro recording is stopped, PWB assigns the recorded commands to the default macro name **Playback**. During the recording, PWB writes the name of each command to the definition of **Playback** in the Record window, which can be viewed as it is updated.

Macro recording in PWB does not record changes in cursor position accomplished by clicking the mouse. Use the keyboard if you want to include cursor movements in a macro.

Record (SHIFT+CTRL+R)
Toggles macro recording on and off.

Arg *textarg* **Record** (ALT+A *textarg* SHIFT+CTRL+R)
Turns on recording if it is off and assigns the name specified in the text argument to the recorded macro. Turns off recording if it is turned on.

Meta Record (F9 SHIFT+CTRL+R)
Toggles macro recording. While recording, no editing commands are executed until recording is turned off. Use this form of the function to record a macro without modifying your file.

Arg Record (ALT+A SHIFT+CTRL+R)
Arg Arg *textarg* **Record** (ALT+A ALT+A *textarg* SHIFT+CTRL+R)
Arg Arg Meta Record (ALT+A ALT+A F9 SHIFT+CTRL+R)
As above but if the target macro already exists, the commands are appended to the end of the macro.

Returns

True	Recording turned on.
False	Recording turned off.

Update

In PWB 2.00, more menu commands can be recorded than with PWB 1.*x*.

Refresh

Key

SHIFT+F7

Refresh
Prompts for confirmation and then rereads the file from disk, discarding its Undo history and all modifications to the file since the file was last saved.

Returns	Condition
True	File reread.
False	Prompt canceled

Arg Refresh (ALT+A SHIFT+F7)
Prompts for confirmation and then removes the file from the active window and the window's file history. If the active window is the last window that has the file in its history, the file is discarded from memory without saving changes, and the file is closed.

Returns	Condition
True	File removed from the window.
False	Prompt canceled, or bad argument. The file is not removed from the window.

Repeat

Key Unassigned

Repeat
Repeats the last editing action relative to the current cursor position. The **Repeat** function considers the following types of operations to be editing actions:

- Typing a contiguous stream of characters without entering a command or moving the cursor
- Deleting text
- Pasting from the clipboard

Repeat does not repeat macros or cursor movements.

Arg *number* **Repeat** (ALT+A *number* Unassigned)
Performs the last action the number of times specified by *number*.

Returns

True	Action repeated and returned true.
False	Action repeated and returned false, or no action to repeat.

Replace

Key CTRL+L

The **Replace** function performs a find-and-replace operation on the current file, prompting for find and replacement strings. **Replace** substitutes all matches of the search pattern without prompting for confirmation.

Replace (CTRL+L)
Performs the replacement from the cursor to the end of the file, wrapping around the end of the file if the **Searchwrap** switch is on.

Arg *boxarg* | *linearg* | *streamarg* **Replace**
(ALT+A *boxarg* | *linearg* | *streamarg* CTRL+L)
Performs the replacement over the selected area.

Note that PWB does not adjust the selection at each replacement for changes in the length of the text. For *boxarg* and *streamarg*, PWB may replace text that was not included in the original selection or miss text included in the original selection.

Arg *mark* **Replace** (ALT+A *mark* CTRL+L)
Performs the replacement on text from the cursor to the specified mark. It searches the range of text as if it were selected, according to the current selection mode. The *mark* argument cannot be a line number.

Arg *number* **Replace** (ALT+A *number* CTRL+L)
Performs the replacement over the specified number of lines, starting with the current line.

Arg Arg ... **Replace** (ALT+A ALT+A ... CTRL+L)
As above except using regular expressions.

Meta ... **Replace** (F9 ... CTRL+L)
As above except the sense of the **Case** switch is reversed for the operation.

Returns

True	At least one replacement was performed.
False	String not found, or invalid pattern.

See **Qreplace**, **Searchwrap**

Example

To use the replace function in a macro, use the phrase:

```
...Replace "pattern" Newline "replacement" Newline +>found...
```

Enter the replies to the prompts as you would when executing **Replace** interactively. This example also shows where to place the conditional to test the result of **Replace**.

You can specify special characters in the find-and-replacement strings by using escape sequences similar to those in the C language. Note that backslashes in the macro string must be doubled.

To restore the usual prompts, use the phrase:

```
...Replace <
```

To use an empty replacement text (replace with nothing), use the following phrase:

```
...Replace "pattern" Newline " " Cdelete Newline...
```

If you find that you write many macros with empty replacements, the common phrase can be placed in a macro, as follows:

```
nothing := " " Cdelete Newline
```

In addition, macro definitions can be more readable with the following definition:

```
with := Newline
```

With these definitions, you can write:

```
... Replace "pattern" with nothing ...
```

Resize

Key Unassigned

Resize
Enters window-resizing mode. When in window-resizing mode, only the following actions are available:

Action	Key
Shrink one row	UP
Expand one row	DOWN
Shrink one column	LEFT
Expand one column	RIGHT
Accept the new size	ENTER
Cancel the resize	ESC

Arg *number* **Resize** (ALT+A *number* Unassigned)
Resizes the window to *number* rows high.

Arg *number* **Meta Resize** (ALT+A *number* F9 Unassigned)
Resizes the window to *number* columns wide.

See **Movewindow**

Restcur

Key Unassigned

Restcur
Moves the cursor to the last position saved with the **Savecur** function (Unassigned, Set To Anchor command, Edit menu). **Restcur** always clears the saved position.

Returns

True	Position restored.
False	No saved position to restore.

See **Selcur**

Right

Keys RIGHT, CTRL+D

Right
Moves the cursor one character to the right. If this action causes the cursor to move out of the window, PWB adjusts the window to the right according to the **Hscroll** switch.

Meta Right (F9 RIGHT)
Moves the cursor to the rightmost position in the window.

Returns

True	Cursor on text in the line.
False	Cursor past text on the line.

Example In a macro, the return value of the **Right** function can be used to test if the cursor is on text in the line or past the end of the line.

The following macro tests the return value to simulate the **Endline** function:

```
MyEndline := Begline :>loop Right +>loop
```

See **Begline**, **Endfile**, **Endline**, **Home**, **Left**

Saveall

Key Unassigned

Saveall
Saves all modified disk files. Pseudofiles are not saved.

Returns **Saveall** always returns true.

Savecur

Key Unassigned

Menu Edit menu, Set Anchor command

Savecur
Saves the cursor position (sets an anchor).

To restore the cursor to the saved position, use the **Restcur** function (Unassigned). To select text from the current position to the saved position, use the Select To Anchor command from the Edit menu or the **Selcur** function (Unassigned).

Returns **Savecur** always returns true.

Sdelete

Key Unassigned

Sdelete
Deletes the character at the cursor. Does not copy the character to the clipboard.

Arg Sdelete (ALT+A Unassigned)
Deletes text from the cursor to the end of the line, including the line break. The deleted text is copied to the clipboard.

Arg *streamarg* | *boxarg* | *linearg* **Sdelete**
(ALT+A *streamarg* | *boxarg* | *linearg* Unassigned)
Deletes the selected stream of text from the starting point of the selection to the cursor and copies it to the clipboard. Always deletes a stream, regardless of the current selection mode.

Meta ... **Sdelete** (F9 ... Unassigned)
As above but discards the deleted text. The contents of the clipboard are unchanged.

Returns **Sdelete** always returns true.

Searchall

Key Unassigned

Searchall
Highlights all occurrences of the previously searched string or pattern. Moves the cursor to the first occurrence in the file.

Arg Searchall (ALT+A Unassigned)
Highlights all occurrences of the string specified by the text from the cursor to the first blank character.

Arg *textarg* **Searchall** (ALT+A *textarg* Unassigned)
Highlights all occurrences of *textarg*.

Arg Arg Searchall (ALT+A ALT+A Unassigned)
Highlights all occurrences of the regular expression defined by the characters from the cursor to the first blank character.

Arg *streamarg* **Searchall** (ALT+A *streamarg* Unassigned)
Highlights all occurrences of *streamarg*.

Arg Arg *textarg* **Searchall** (ALT+A ALT+A *textarg* Unassigned)
Highlights all occurrences of a regular expression defined by *textarg*.

Meta ... **Searchall** (F9 ... Unassigned)
As above but reverses the value of the **Case** switch for one search.

Returns

True	String or pattern found.
False	No matches found.

Selcur

Key Unassigned

Menu Edit menu, Select To Anchor command

Selcur
Selects text from the cursor to the position saved using the Set Anchor command from the Edit menu or the **Savecur** function (Unassigned). If no position has been saved, **Selcur** selects text from the cursor to the beginning of the file.

Returns **Selcur** always returns true.

Select

Keys SHIFT+PGUP, SHIFT+CTRL+PGUP, SHIFT+PGDN, SHIFT+CTRL+PGDN, SHIFT+END, SHIFT+CTRL+END, SHIFT+HOME, SHIFT+CTRL+HOME, SHIFT+LEFT, SHIFT+CTRL+LEFT, SHIFT+UP, SHIFT+RIGHT, SHIFT+CTRL+RIGHT, SHIFT+DOWN

Select
Causes a shifted key to take on the cursor-movement function associated with the unshifted key and begins or extends a selection.

To see the key combinations currently assigned to this function, use the Key Assignments command from the Options menu.

Selmode

Key Unassigned

Selmode
Advances the selection mode between stream, line, and box modes, starting with the current mode.

Returns

True	New mode is stream mode.
False	New mode is box mode or line mode.

See **_pwbstreammode**, **_pwbboxmode**, **_pwblinemode**

Selwindow

Key F6

Selwindow
Moves the focus to the next window.

Arg Selwindow (ALT+A F6)
Moves the focus to the next unminimized window. Minimized windows (icons) are skipped.

Arg *number* **Selwindow** (ALT+A *number* F6)
Moves the focus to the specified window.

Meta Selwindow (F9 F6)
Moves the focus to the previous window.

Arg Meta Selwindow (ALT+A F9 F6)
Moves the focus to the previous unminimized window.

Returns

True	Focus moved to another window.
False	No other windows are open.

Setfile

Key F2

Setfile
Switches to the first file in the active window's file history. If there are no files in the file history, PWB displays the message `No alternate file`. When the **Autosave** switch is set to yes, PWB saves the current file if it has been modified.

Setfile does not honor the **Newwindow** switch. To open a new window when you open a file, use **Openfile**.

Arg Setfile (ALT+A F2)
Switches to the filename that begins at the cursor and ends with the first blank character.

Arg *textarg* **Setfile** (ALT+A *textarg* F2)
Switches to the file specified by *textarg*. If the file is not already open, PWB opens it. You can use environment-variable specifiers in the argument.

If the argument is a drive or directory name, PWB changes the current drive or directory to the specified one and displays a message to confirm the change. See: **Infodialog**.

Arg !*number* **Setfile** (ALT+A **!***number* F2)
If the argument has the form **!***number*, PWB switches to the file with that number in the file history. The number can be from 1 to 9, inclusive. See: **_pwbfile***n*.

Arg *wildcard* **Setfile** (ALT+A *wildcard* F2)
If the argument is a wildcard, PWB creates a pseudofile containing a list of files that match the pattern. To open a file from this list, position the cursor at the beginning of the name and execute **Arg Openfile** (ALT+A F10) or **Arg Setfile** (ALT+A F2).

Meta … **Setfile** (F9 … F2)
As above but does not save the changes to the current file.

Arg Arg Setfile (ALT+A ALT+A F2)
Saves the current file.

Arg Arg *textarg* **Setfile** (ALT+A ALT+A *textarg* F2)
Saves the current file under the name specified by *textarg*.

Returns

True File opened successfully.
False No alternate file, the specified file does not exist, and you did not wish
to create it; or the current file needs to be saved and cannot be saved.

See **Newfile**

Sethelp

Key SHIFT+CTRL+S

The **Sethelp** function opens and closes single Help files. The **Sethelp** function can also display the current list of open Help files. **Sethelp** affects only the current PWB session.

Arg Sethelp (ALT+A SHIFT+CTRL+S)
Opens the Help file specified by the filename at the cursor.

Arg *streamarg* | *textarg* **Sethelp** (ALT+A *streamarg* | *textarg* SHIFT+CTRL+S)
Opens the Help file specified by the selected filename.

Meta ... **Sethelp** (F9 ALT+A SHIFT+CTRL+S)
As above except the specified Help file is closed.

Arg ? Sethelp (ALT+A ? SHIFT+CTRL+S)
Lists all currently open Help files.

Returns

True	Help file opened or closed, or list of Help files displayed.
False	The specified file could not be opened or closed, or the list of files could not be displayed.

See **Helpfiles**

Setwindow

Key CTRL+]

Setwindow
Redisplays the contents of the active window.

Meta Setwindow (F9 CTRL+])
Redisplays the current line.

Arg Setwindow (ALT+A CTRL+])
Adjusts the window so that the cursor position becomes the home position (upper-left corner).

Returns **Setwindow** always returns true.

Shell

Key SHIFT+F9

Shell
Runs an operating-system command shell. To return to PWB, type `exit` at the operating-system prompt.

> **Warning** Do not start terminate-and-stay-resident (TSR) programs in a shell. This causes unpredictable results.

Arg Shell (ALT+A SHIFT+F9)
Runs the text from the cursor to the end of the line as a command to the shell, and returns to PWB.

Arg *boxarg* | *linearg* **Shell** (ALT+A *boxarg* | *linearg* SHIFT+F9)
Runs each selected line as a separate command to the shell, and returns to PWB.

Arg *textarg* **Shell** (ALT+A *textarg* SHIFT+F9)
Runs *textarg* as a command to the shell, and returns to PWB.

Meta ... **Shell** (F9 ... SHIFT+F9)
Runs a shell, ignoring the **Autosave** switch. Modified files are not saved to disk, but they are retained in PWB's virtual memory.

Returns

True	Shell ran successfully.
False	Invalid argument, or error starting the operating-system command processor.

See **Askrtn**, **Restart**, **Savescreen**

Sinsert

Key CTRL+J

Sinsert
Inserts a space at the cursor.

Arg Sinsert (ALT+A CTRL+J)
Inserts a line break at the cursor, splitting the line.

Arg *streamarg* | *linearg* | *boxarg* **Sinsert**
(ALT+A *streamarg* | *linearg* | *boxarg* CTRL+J)
Inserts a stream of blanks between the starting point of the selection and the cursor. The insertion is always a stream, regardless of the current selection mode.

Returns

True	Spaces or line break inserted.
False	Insertion would make a line too long.

Example

The following macro inserts a stream of spaces up to the next tab stop, regardless of the current selection mode:

```
InsertTab := Arg Tab Sinsert
```

See **Insert**, **Linsert**

Tab

Key TAB

Tab
Moves the cursor to the next tab stop. If there are no tab stops to the right of the cursor, the cursor does not move. Tab stops are defined by the **Tabstops** switch.

Returns

True	Cursor moved.
False	Cursor not moved.

Update In PWB 1.*x*, tab stops appear at fixed intervals. In PWB 2.00, tab stops can be at variable or fixed intervals.

See **Backtab**

Tell

Key CTRL+T

Tell
Displays the message `Press a key to tell about` and waits for a keystroke. After you press a key or combination of keys, **Tell** brings up the Tell dialog box showing the name of the key and its assigned function in TOOLS.INI key-assignment format.

The key-assignment format is:

function:*key*

If the key is not assigned a function, **Tell** displays `unassigned` for the function name. See: **Unassigned**.

If you press a combination of keys, but **Tell** still shows the `Press a key` prompt (when you press SCROLL LOCK, for example), PWB is unable to recognize that combination of keys and you cannot use it as a key assignment.

Arg Tell (ALT+A CTRL+T)
Prompts for a key, then displays the name of the function or macro assigned to the key in one of these formats:

function:*key*
macroname:=*definition*

Arg *textarg* **Tell** (ALT+A *textarg* CTRL+T)
Displays the definition of the macro named by *textarg*. If you specify a PWB function, **Tell** displays:

function:*function*

Meta ... **Tell** (F9 ... CTRL+T)
As above except **Tell** types the result into the current file rather than displaying it in a dialog box. This is how to discover the definition of any macro, including PWB macros.

Returns

True	Assignment displayed or typed.
False	No assignment for the key or the specified name.

Update

In PWB 1.*x*, the prompt and results appear on the status bar; in PWB 2.00, the prompt and results appear in dialog boxes.

Remarks

Meta Tell is a convenient and reliable way of writing a key assignment when you are configuring PWB.

For example, if you want to execute the **Curdate** function (type today's date) when you press the CTRL, SHIFT, and D keys simultaneously, perform the following steps:

1. Go to an empty line in the [PWB] section of TOOLS.INI.
2. Execute **Meta Tell** (F9 CTRL+T).

   Tell displays the message: `Press a key to tell about`.
3. Press the D, SHIFT, and CTRL keys simultaneously.

   If you have not already assigned a function to this combination, **Tell** types:

   ```
   unassigned:Shift+Ctrl+D
   ```

4. Select the word `unassigned` and type `curdate`.
5. If you want the assignment to take effect immediately, move the cursor to the line you've just entered and execute the **Assign** function (ALT+=).

You can use **Meta Arg** *textarg* **Tell** to recover the definition of a predefined PWB macro or a macro that you have not saved or entered into a file.

See **_pwbuser***n*, **Assign**, **Record**

Unassigned

Keys Assigned to all available keys.

Unassigned
Displays a message for keys that do not have a function assignment.

All unassigned keys are actually assigned the **Unassigned** function. Thus, to remove a function assignment for a key, assign the **Unassigned** function to the key. The **Unassigned** function is not useful in macros.

Returns The **Unassigned** function always returns false.

See **Assign**, **Tell**

Undo

Keys **ALT**+BKSP, SHIFT+CTRL+BKSP

Undo
Reverses the last editing operation. The maximum number of times this can be performed for each file is set by the **Undocount** switch.

Meta Undo (F9 ALT+BKSP)
Performs the operation previously reversed with **Undo**. This action is often called "redo."

Returns

True	Operation undone or redone.
False	Nothing to undo or redo.

See **_pwbundo**, **_pwbredo**, **Repeat**

Up

Keys UP, CTRL+E

Up
Moves the cursor up one line. If a selection has been started, it is extended by one line. If this movement results in the cursor moving out of the window, the window is adjusted upward as specified by the **Vscroll** switch.

Meta Up (F9 UP)
Moves the cursor to the top of the window without changing the column position.

Returns

True	Cursor moved.
False	Cursor not moved; the cursor is already at the destination.

See **Down**

Usercmd

Key Unassigned

The **Usercmd** function executes a custom command added to the Run menu by using Customize command from the Run menu or setting the **User** switch.

Arg *number* **Usercmd** (ALT+A *number* Unassigned)
Executes the given custom Run menu command. The *number* can be in the range 1–9.

Returns

True	Command exists.
False	Command does not exist, or invalid argument.

See **_pwbuser***n*, **Assign**, **Record**

Window

Key Unassigned

Window
Switch to the next window.

Returns	Condition
True	Switched to next window.
False	No next window to switch to: zero or one window open.

Arg [[**Arg**]] **Window** (ALT+A [[ALT+A]] Unassigned)
Open a new window.

Returns	Condition
True	Opened a new window.
False	Window not opened.

Meta Window (F9 Unassigned)
Close the active window.

Returns	Condition
True	Window closed.
False	No open window to close.

Meta Arg Window (ALT+A F9 Unassigned)
Switch to the previous window.

Returns	Condition
True	Switched to previous window.
False	No previous window to switch to: zero or one window open.

Update In PWB 1.*x*, **Arg Window** and **Arg Arg Window** split the window at the cursor. In PWB 2.00, these forms of **Window** open a new window.

See **Selwindow**, **Setwindow**

Winstyle

Key CTRL+F6

Winstyle

Advances through the following series of window styles, starting from the current style:

Horizontal Scroll Bar	Vertical Scroll Bar
No	No
No	Yes
Yes	No
Yes	Yes

When the horizontal scroll bar is not shown, a maximized window does not show its bottom border. Similarly, when the vertical scroll bar is not shown, a maximized window does not show its left and right borders. PWB always displays the title bar.

To get the "clean-screen" look, maximize the window and advance the window style until the borders disappear.

Default Set the default window style with the **Defwinstyle** switch.

Returns

True	Changed window style.
False	No windows open.

Update The no-border state in PWB 1.*x* is not available in PWB 2.00. In PWB 2.00, when a window is maximized and no scroll bars are present, PWB displays the window without borders.

See **Maximize**

Predefined PWB Macros

PWB predefines a number of macros, most of which correspond to a command in the PWB menus. You can define a shortcut key for a menu command by assigning the key to the corresponding macro. Note that some menu commands such as the Open command from the File menu do not correspond to a macro, and some macros do not correspond to a menu command.

Table 7.12 PWB Macros

Macro	Description	Key
Curfile	Current file's full path	Unassigned
Curfileext	Current file's extension	Unassigned
Curfilenam	Current file's name	Unassigned
_pwbarrange	Arrange command, Window menu	ALT+F5
_pwbboxmode	Box Mode command, Edit menu	Unassigned
_pwbbuild	Build command, Project menu	Unassigned
_pwbcancelbuild	Cancel Build command, Project menu	Unassigned
_pwbcancelprint	Cancel Print command, File menu	Unassigned
_pwbcancelsearch	Cancel Search command, Search menu	Unassigned
_pwbcascade	Cascade command, Window menu	F5
_pwbclear	Delete command, Edit menu	DEL
_pwbclose	Close command, Window menu	CTRL+F4
_pwbcloseall	Close All command, Window menu	Unassigned
_pwbclosefile	Close command, File menu	Unassigned
_pwbcloseproject	Close command, Project menu	Unassigned
_pwbcompile	Compile command, Project menu	Unassigned
_pwbfile*n*	*n file*, File menu	Unassigned
_pwbgotomatch	Goto Match command, Search menu	Unassigned
_pwbhelp_again	Next command, Help menu	Unassigned
_pwbhelp_back	Previous Help topic	ALT+F1
_pwbhelp_contents	Contents command, Help menu	SHIFT+F1
_pwbhelp_context	Topic command, Help menu	F1
_pwbhelp_general	Help on Help command, Help menu	Unassigned
_pwbhelp_index	Index command, Help menu	Unassigned
_pwbhelpnl	Display the message: `Online Help Not Loaded`	F1 when Help extension not loaded
_pwbhelp_searchres	Search Results command, Help menu	Unassigned
_pwblinemode	Line Mode command, Edit menu	Unassigned
_pwblogsearch	Log command, Search menu	Unassigned
_pwbmaximize	Maximize command, Window menu	CTRL+F10
_pwbminimize	Minimize command, Window menu	CTRL+F9
_pwbmove	Move command, Window menu	CTRL+F7
_pwbnewfile	New command, File menu	Unassigned

Table 7.12 PWB Macros (*continued*)

Macro	Description	Key
_pwbnewwindow	New command, Window menu	Unassigned
_pwbnextfile	Next command, File menu	Unassigned
_pwbnextlogmatch	Next Match command, Search menu	SHIFT+CTRL+F3
_pwbnextmatch	Next Match command, Search menu	Unassigned
_pwbnextmsg	Next Error command, Project menu	SHIFT+F3
_pwbpreviouslogmatch	Previous Match command, Search menu	SHIFT+CTRL+F4
_pwbpreviousmatch	Previous Match command, Search menu	Unassigned
_pwbprevmsg	Previous Error command, Project menu	SHIFT+F4
_pwbprevwindow	Move to previous window	SHIFT+F6
_pwbquit	Exit command, File menu	ALT+F4
_pwbrebuild	Rebuild All command, Project menu	Unassigned
_pwbrecord	Record command, Edit menu	Unassigned
_pwbredo	Redo command, Edit menu	Unassigned
_pwbrepeat	Repeat command, Edit menu	Unassigned
_pwbresize	Resize command, Window menu	CTRL+F8
_pwbrestore	Restore command, Window menu	CTRL+F5
_pwbsaveall	Save All command, File menu	Unassigned
_pwbsavefile	Save command, File menu	SHIFT+F2
_pwbsetmsg	Goto Error command, Project menu	Unassigned
_pwbshell	DOS Shell command, File menu	Unassigned
_pwbstreammode	Stream Mode command, Edit menu	Unassigned
_pwbtile	Tile command, Window menu	SHIFT+F5
_pwbundo	Undo command, Edit menu	Unassigned
_pwbuser*n*	*command n*, Run menu	ALT+F*n*
_pwbviewbuildresults	View build results button	Unassigned
_pwbviewsearchresults	View search results button	Unassigned
_pwbwindow*n*	*n file*, Window menu	ALT+*n*

PWB continually redefines the following macros to reflect the current file's name:

Macro	Description
Curfile	Full path
Curfileext	File extension
Curfilenam	File base name

PWB uses the following special-purpose macros:

Macro	Description
Autostart	Executed on startup while reading TOOLS.INI
Mgreplist	List of files for logged searches, multifile replace, **Mgrep**, and **Mreplace**
Playback	Default name of recorded macros
Restart	(Obsolete)

By default, these macros are undefined.

Autostart

Key Unassigned

The special PWB macro **Autostart** is executed after PWB finishes all initialization at startup. If used, it must be defined in the [PWB] section of TOOLS.INI.

Definition By default, **Autostart** is not defined.

Curfile

Key Unassigned

The **Curfile** macro types the full path of the current file. This macro is redefined each time you switch to a new file.

Definition curfile := "*pathname*"

Example The following macro copies the full path of the current file to the clipboard for later use:

```
Path2clip := Arg Curfile Copy
```

See **Arg**, **Copy**, **Curdate**, **Curday**, **Curfilenam**, **Curfileext**, **Curtime**

Curfileext

Key Unassigned

The **Curfileext** macro types the filename extension of the current file. This macro is redefined each time you switch to a new file.

Definition curfileext := "*extension*"

Example The following macro copies the base name plus the extension of the current file to the clipboard for later use:

```
Filename2clip := Arg Curfilenam Curfileext Copy
```

See **Arg**, **Copy**, **Curdate**, **Curday**, **Curfile**, **Curfilenam**, **Curtime**

Curfilenam

Key Unassigned

The **Curfilenam** macro types the base name of the current file. This macro is redefined each time you switch to a new file.

Definition curfilenam := "*basename*"

Example The following macro copies the base name of the current file to the clipboard for later use:

```
Name2clip := Arg Curfilenam Copy
```

See **Arg**, **Copy**, **Curdate**, **Curday**, **Curfile**, **Curfileext**, **Curtime**

Mgreplist

Key Unassigned

The special PWB macro **Mgreplist** is used by the Find and Replace commands on the Search menu, **Mgrep**, **Mreplace**, and **Mreplaceall** to specify the list of files to search.

When you create a list of files to search using the Files button in either the Find or Replace dialog box, PWB redefines the **Mgreplist** macro with the specified list of files.

To see the current list of files, choose the Files button in the Replace dialog box. You can change the list in this dialog box, and either choose OK to perform the find-and-replace operation, or choose Cancel to cancel the replace and accept the changes to **Mgreplist**.

You can also insert the definition of **Mgreplist** into the current file by using the phrase: **Arg Meta** `Mgreplist` **Tell** (ALT+A F9 `Mgreplist` CTRL+T).

You can edit the macro, then redefine it by using the **Assign** function (ALT+=).

Definition Mgreplist:= "*list*"

list

Space-separated list of filenames

The filenames can use the operating-system wildcards (* and ?), and can use environment-variable specifiers. Note that backslashes (\) must be doubled in the macro string.

See **Assign**, **Tell**, **Mgrep**, **Mreplace**, **Mreplaceall**

Restart

Key Unassigned

Update In PWB 1.*x*, the special PWB macro Restart is executed whenever PWB returns from a shell, build, or other external operation.

In PWB 2.00, the **Restart** macro is never executed automatically and has no special meaning; it is an ordinary macro.

_pwbarrange

Key ALT+F5

Menu Window menu, Arrange command

The **_pwbarrange** macro arranges all unminimized windows on the desktop. The following illustration shows a typical desktop after execution of **_pwbarrange**:

Figure 7.1 Arranged Windows

Definition

_pwbarrange:=cancel arg arrangewindow <

Cancel
Establishes a uniform "ground state" by canceling any selection or argument.

Arg Arrangewindow
Arranges all unminimized windows on the desktop.

<
Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Arrangewindow**

_pwbboxmode

Key Unassigned

Menu Edit menu, Box Mode command

The **_pwbboxmode** macro sets the selection mode to box selection mode.

Definition _pwbboxmode := :>more selmode ->more selmode

:>more
Defines the label `more`.

Selmode
Advances to the next selection mode.

->more
Branches to the label `more` if **Selmode** returns false.

The **Selmode** function advances the selection mode from box, to stream, to line. **Selmode** returns true when the mode is stream mode. The macro executes the **Selmode** function until it returns true (sets stream mode), then advances the selection mode once to set box selection mode.

See **Enterselmode**, **Selmode**

_pwbbuild

Key Unassigned

Menu Project menu, Build command

The **_pwbbuild** macro builds the “all” target of the current PWB project. The “all” pseudotarget in a PWB project lists all the targets in the project.

For non-PWB projects, **_pwbbuild** builds the targets that were last specified by using the Build Target command from the Project menu. PWB redefines **_pwbbuild** each time you use Build Target. If no target has been specified, NMAKE builds the first target listed in the project makefile.

Definition _pwbbuild := cancel arg "all" compile <

Cancel
Establishes a uniform “ground state” by cancelling any selection or argument.

Arg “all” **Compile**
Builds the `all` pseudotarget in the current project.

<
Restores the function’s prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Arg**, **Cancel**, **Compile**

_pwbcancelbuild

Key Unassigned

Menu Project menu, Cancel Build command

The **_pwbcancelbuild** macro terminates the current background build or compile and flushes any queued build operations.

Definition _pwbcancelbuild := cancel arg meta compile

Cancel
Establishes a uniform "ground state" by canceling any selection or argument.

Arg Meta Compile
Terminates the background build process.

See **Arg**, **Cancel**, **Compile**, **Meta**

_pwbcancelprint

Key Unassigned

The **_pwbcancelprint** macro terminates all background print operations.

Definition _pwbcancelprint := cancel arg meta print

Cancel
Establishes a uniform "ground state" by canceling any selection or argument.

Arg Meta Print
Terminate background print operations.

See **Arg**, **Cancel**, **Meta**, **Print**

_pwbcancelsearch

Key Unassigned

Menu Search menu, Cancel Search command

The **_pwbcancelsearch** macro cancels the current background search. PWB performs logged searches in the background under multithreaded environments.

Definition

_pwbcancelsearch := cancel cancelsearch <

Cancel
Establishes a uniform “ground state” by canceling any selection or argument.

Cancelsearch
Cancels the current background search.

<
Restores the function’s prompt (if any). By default, function prompts are suppressed while a macro is running.

See

Cancel, **Cancelsearch**, **Logsearch**

_pwbcascade

Key

F5

Menu

Window menu, Cascade command

The **_pwbcascade** macro arranges all unminimized windows in cascaded fashion so that all of their titles are visible. Up to 16 unminimized windows can be cascaded.

Definition

_pwbcascade := cancel arrangewindow <

Cancel
Establishes a uniform “ground state” by canceling any selection or argument.

Arrangewindow
Cascades all unminimized windows.

<
Restores the function’s prompt (if any). By default, function prompts are suppressed while a macro is running.

See

Arrangewindow, **Cancel**

_pwbclear

Key

DEL

Menu Edit menu, Delete command

The **_pwbclear** macro removes the selected text from the file. If there is no selection, PWB removes the current line.

The selection or line is not copied to the clipboard. It can be recovered only by using the Undo command from the Edit menu or **Undo** (ALT+BKSP).

Definition _pwbclear := meta delete

Meta Delete
Removes the selection or the current line from the file without modifying the clipboard.

See **Delete**, **Meta**

_pwbcloseall

Key Unassigned

Menu Window menu, Close All command

The **_pwbcloseall** macro closes all open windows.

Definition _pwbcloseall := cancel arg arg meta window <

Cancel
Establishes a uniform "ground state" by canceling any selection or argument.

Arg Arg Meta Window
Closes all windows.

<
Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Arg**, **Cancel**, **Meta**, **Window**

_pwbclosefile

Key Unassigned

Menu File menu, Close command

The **_pwbclosefile** macro closes the current file. If no files remain in the window's file history, the window is closed.

Definition

_pwbclosefile := cancel closefile <

Cancel
Establishes a uniform "ground state" by canceling any selection or argument.

Closefile
Closes the current file.

<
Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Cancel**, **Closefile**

_pwbcloseproject

Key Unassigned

Menu Project menu, Close Project command

The **_pwbcloseproject** macro closes the current project.

Definition

_pwbcloseproject := cancel arg arg project <

Cancel
Establishes a uniform "ground state" by canceling any selection or argument.

Arg Arg Project
Closes the current project.

<
Restores the function's prompt (if any). By default, function prompts are suppressed within a macro.

See **Arg**, **Cancel**, **Project**

_pwbcompile

Key Unassigned

Menu Project menu, Compile File command

The **_pwbcompile** macro compiles the current file.

Definition

_pwbcompile := cancel arg compile <

Cancel
Establishes a uniform "ground state" by canceling any selection or argument.

Arg Compile
Compiles the current file.

<
Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See

Arg, **Cancel**, **Compile**

_pwbgotomatch

Key

Unassigned

Menu

Search menu, Goto Match command

The **_pwbgotomatch** macro sets the match listed at the current location in the Search Results pseudofile as the current match and moves the cursor to the location specified by that match.

Definition

_pwbgotomatch := cancel arg arg nextsearch <

Cancel
Establishes a uniform "ground state" by canceling any selection or argument.

Arg Arg Nextsearch
Goes to the current match.

<
Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See

Arg, **Cancel**, **Nextsearch**

_pwbhelpnl

The **_pwbhelpnl** macro displays a message indicating the Help extension is not loaded. The Help keys are assigned this macro until the Help extension is loaded.

Definition

_pwbhelpnl := cancel arg "OnLine Help Not Loaded" message

Cancel
Establishes a uniform “ground state” by canceling any selection or argument.

Arg “OnLine Help Not Loaded” **Message**
Displays the message on the status bar.

See

Arg, **Cancel**, **Load**, **Message**

_pwbhelp_again

Key

Unassigned

Menu

Help menu, Next command

The **_pwbhelp_again** displays the next occurrence of the last topic for which you requested Help. If no other occurrences of the topic are defined in the open files, PWB redisplays the current topic.

The topic that PWB looks up when you use this command is displayed after the Next command on the Help menu, as follows:

Next: *topic key*

topic Topic string used for the command.

key Current key assignment for **_pwbhelp_again** (if any).

Definition

_pwbhelp_again:=cancel arg pwbhelp.pwbhelpnext <

Cancel
Establishes a uniform “ground state” by canceling any selection or argument.

Arg
Sets the **Arg** prefix for the **Pwbhelpnext** function.

Pwbhelp.
Specifies that the function is the PWBHELP extension function.

Pwbhelpnext

Displays the next occurrence of the previously requested topic.

<

Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Pwbhelpnext**

_pwbhelp_back

Key ALT+F1

The **_pwbhelp_back** macro displays the previously viewed Help topic. Up to 20 topics are kept in the Help backtrace list.

Definition _pwbhelp_back:=cancel meta pwbhelp.pwbhelpnext <

Cancel
Establishes a uniform “ground state” by canceling any selection or argument.

Meta
Sets the meta prefix for the function.

Pwbhelp.
Specifies that the function is the PWBHELP extension function.

Pwbhelpnext
Displays the previously viewed Help topic.

<
Restores the function’s prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Pwbhelpnext**

_pwbhelp_contents

Key SHIFT+F1

Menu Help menu, Contents command

The **_pwbhelp_contents** macro opens the Help window and displays the top-level contents of the Help system.

Within the Help system, most Help topics have a Contents button at the top of the window. This button also takes you to the top-level contents.

Definition _pwbhelp_contents:=cancel arg "advisor.hlp!h.contents" pwbhelp.pwbhelp <

Cancel
Establishes a uniform “ground state” by canceling any selection or argument.

Arg "advisor.hlp!h.contents"
Defines a text argument with the topic name for the general table of contents.

Pwbhelp.
Specifies that the function is the PWBHELP extension function.

Pwbhelp
Looks up the topic `h.contents` in the ADVISOR.HLP Help file.

<
Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Pwbhelp**

_pwbhelp_context

Key F1

Menu Help menu, Topic command

The **_pwbhelp_context** macro looks up Help on the topic at the cursor, the current selection, or the specified text argument.

Definition _pwbhelp_context:=arg pwbhelp.pwbhelp <

Arg
Sets the Arg prefix for the **Pwbhelp** function.

Pwbhelp.
Specifies that the function is the PWBHELP extension function.

Pwbhelp
Displays Help on the topic at the cursor. When text is selected, displays Help on the selected text. When you have entered an argument in the Text Argument dialog box, displays Help on the topic specified by the text argument.

<
Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Pwbhelp**

_pwbhelp_general

Key Unassigned

Menu Help menu, Help on Help command

The **_pwbhelp_general** macro opens the Help window and displays information about using the Help system.

Definition _pwbhelp_general:=cancel arg "advisor.hlp!h.default" pwbhelp.pwbhelp <

Cancel
Establishes a uniform “ground state” by canceling any selection or argument.

Arg "advisor.hlp!h.default"
Defines a text argument with the topic name for default Help.

Pwbhelp.
Specifies that the function is the PWBHELP extension function.

Pwbhelp
Looks up the topic “h.default” in the ADVISOR.HLP Help file.

<
Restores the function’s prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Pwbhelp**

_pwbhelp_index

Key Unassigned

Menu Help menu, Index command

The **_pwbhelp_index** macro opens the Help window and displays the top-level table of indexes in the Help system.

Within the Help system, most Help topics have an Index button at the top of the window. This button also takes you to the top-level table of indexes.

Definition _pwbhelp_index:=cancel arg "advisor.hlp!h.index" pwbhelp.pwbhelp <

Cancel
Establishes a uniform “ground state” by canceling any selection or argument.

Arg "advisor.hlp!h.index"
Defines a text argument with the topic name for the Help index.

Pwbhelp.
Specifies that the function is the PWBHELP extension function.

Pwbhelp
Looks up the topic "h.index" in the ADVISOR.HLP Help file.

<
Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Pwbhelp**

_pwbhelp_searchres

Key Unassigned

Menu Help menu, Search Results command

The **_pwbhelp_searchres** macro opens the Help window and displays the list of matches found during the last global Help search.

Definition _pwbhelp_searchres:=cancel pwbhelp.pwbhelpsearch <

Cancel
Establishes a uniform "ground state" by canceling any selection or argument.

Pwbhelp.
Specifies that the function is the PWBHELP extension function.

Pwbhelpsearch
Opens the Help window and displays the results of the last global Help search.

<
Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Pwbhelpsearch**

_pwblinemode

Key Unassigned

Menu Edit menu, Line Mode command

The **_pwblinemode** macro sets the selection mode to line selection mode.

Definition

_pwblinemode := :>more selmode ->more selmode selmode

:>more
Defines the label `more`.

Selmode
Advances to the next selection mode.

->more
Branches to the label `more` if **Selmode** returns false.

The **Selmode** function advances the selection mode from box, to stream, to line. **Selmode** returns true when the mode is stream mode. The macro executes the **Selmode** function until it returns true (sets stream mode), then advances the selection mode twice to set line selection mode.

See

Enterselmode, **Selmode**

_pwblogsearch

Key

Unassigned

Menu

Search menu, Log command

The **_pwblogsearch** macro toggles search logging on and off.

When search logging is turned on, PWB displays a bullet next to the Log command on the Search menu. The Next Match command executes the **_pwbnextlogmatch** macro, and the Previous Match command executes the **_pwbpreviouslogmatch** macro. When search logging is turned off, no bullet appears and the Next Match and Previous Match commands execute **_pwbnextmatch** and **_pwbpreviousmatch**.

Definition

_pwblogsearch := cancel logsearch <

Cancel
Establishes a uniform “ground state” by canceling any selection or argument.

Logsearch
Toggles the logging of search results on and off.

<
Restores the function’s prompt (if any). By default, function prompts are suppressed while a macro is running.

See

Cancel, **Logsearch**

_pwbmaximize

Key CTRL+F10

Menu Window menu, Maximize command

The **_pwbmaximize** macro enlarges the active window to its largest possible size, showing only the window, the menu bar, and the status bar on the PWB screen.

Definition _pwbmaximize := cancel maximize <

Cancel
Establishes a uniform "ground state" by canceling any selection or argument.

Maximize
Enlarges the active window to full size.

<
Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Cancel**, **Minimize**

_pwbminimize

Key CTRL+F9

Menu Window menu, Minimize command

The **_pwbminimize** macro minimizes the active window, reducing the window to an icon. To restore a window to its original size, double-click in the box or use the Restore command (CTRL+F5) on the Window menu.

Definition _pwbminimize := cancel minimize <

Cancel
Establishes a uniform "ground state" by canceling any selection or argument.

Minimize
Shrinks the window to an icon.

<
Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Cancel**, **Maximize**, **Minimize**

_pwbmove

Key CTRL+F7

Menu Window menu, Move command

The **_pwbmove** macro starts window-moving mode for the active window. In window-moving mode, you can only do the following:

Action	Key
Move up one row	UP
Move down one row	DOWN
Move left one column	LEFT
Move right one column	RIGHT
Accept the new position	ENTER
Cancel the move	ESC

To move the window in larger increments, you can use a numeric argument with the **Movewindow** function.

Definition _pwbmove := cancel movewindow <

Cancel
Establishes a uniform “ground state” by canceling any selection or argument.

Movewindow
Starts window-moving mode.

<
Restores the function’s prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Arrangewindow**, **Cancel**, **Maximize**, **Minimize**, **Resize**

_pwbnewfile

Key Unassigned

Menu File menu, New command

The **_pwbnewfile** macro creates a new pseudofile.

New pseudofiles are given a unique name of the form:

<Untitled.*nnn*>Untitled.*nnn*

where <*nnn*> is a three-digit number starting with 001 at the beginning of each PWB session. The window title shows Untitled.*nnn*. The file may be referred to by the name <Untitled.*nnn*>.

When the **Newwindow** switch is set to yes, or the active window is a PWB window, a new window is opened for the file. Otherwise, the file is opened in the active window, and the previous file is placed in the window's file history.

Definition

_pwbnewfile := cancel newfile <

Cancel
Establishes a uniform "ground state" by canceling any selection or argument.

Newfile
Creates a new untitled pseudofile.

<
Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See

Cancel, **Setfile**

_pwbnewwindow

Key

Unassigned

Menu

Window menu, New command

The **_pwbnewwindow** macro opens a new window, which shows the current file. The new window has the complete file history as the original window.

Definition

_pwbnewwindow := cancel arg window

Cancel
Establishes a uniform "ground state" by canceling any selection or argument.

Arg Window
Opens a new window on the current file

See

Arg, **Cancel**, **Window**

_pwbnextfile

Key Unassigned

Menu File menu, Next command

The **_pwbnextfile** macro moves to the next file in the list of files specified on the PWB command line. If no more files remain in the list, this macro ends the PWB session.

When the **Newwindow** switch is set to yes, or the active window is a PWB window, a new window is opened for the file. Otherwise, the file is opened in the active window, and the previous file is placed in the window's file history.

Definition _pwbnextfile := cancel exit <

Cancel
Establishes a uniform "ground state" by canceling any selection or argument.

Exit
Moves to the next file specified on the command line.

<
Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Exit**, **Askexit**, **Cancel**, **PWB Command Line**

_pwbnextlogmatch

Key SHIFT+CTRL+F3

Menu Search menu, Next Match command

The **_pwbnextlogmatch** macro advances the cursor to the next match listed in the Search Results pseudofile.

The Next Match command on the Search menu executes this macro when search logging is turned on. When search logging is turned off, Next Match executes the **_pwbnextmatch** macro.

Definition _pwbnextlogmatch := cancel nextsearch <

Cancel
Establishes a uniform "ground state" by canceling any selection or argument.

Nextsearch
Advances to the next match in Search Results.

<
Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Cancel**, **Nextsearch**

_pwbnextmatch

Key Unassigned

Menu Search menu, Next Match command

The **_pwbnextmatch** macro searches forward in the file using the last search pattern and options. The search options are Match Case, Wrap Around, and Regular Expression.

The Next Match command on the Search menu executes this macro when search logging is turned off. When search logging is turned on, Next Match executes the **_pwbnextlogmatch** macro.

Definition _pwbnextmatch := cancel psearch <

Cancel
Establishes a uniform "ground state" by canceling any selection or argument.

Psearch
Searches forward in the file for the next occurrence of the last search string or pattern.

<
Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Cancel**, **Psearch**

_pwbnextmsg

Key SHIFT+F3

Menu Project menu, Next Error command

The **_pwbnextmsg** macro moves the cursor to the next message in Build Results.

Definition

_pwbnextmsg := cancel nextmsg <

Cancel
Establishes a uniform “ground state” by canceling any selection or argument.

Nextmsg
Goes to the next message in Build Results.

<
Restores the function’s prompt (if any). By default, function prompts are suppressed while a macro is running.

See

Cancel, **Nextmsg**

_pwbpreviouslogmatch

Key

SHIFT+CTRL+F4

Menu

Search menu, Previous Match command

The **_pwbpreviouslogmatch** macro moves the cursor to the previous match listed in the Search Results pseudofile.

The Previous Match command on the Search menu executes this macro when search logging is turned on. When search logging is turned off, Previous Match executes the **_pwbpreviousmatch** macro.

Definition

_pwbpreviouslogmatch := cancel arg "-1" nextsearch <

Cancel
Establishes a uniform “ground state” by canceling any selection or argument.

Arg "-1" **Nextsearch**
Moves to the previous match listed in Search Results.

<
Restores the function’s prompt (if any). By default, function prompts are suppressed while a macro is running.

See

Arg, **Cancel**, **Nextsearch**

_pwbpreviousmatch

Key Unassigned

Menu Search menu, Previous Match command

The **_pwbpreviousmatch** macro searches backward in the file, using the last search pattern and options. The search options are Match Case, Wrap Around, and Regular Expression.

The Previous Match command on the Search menu executes this macro when search logging is turned off. When search logging is turned on, Previous Match executes the **_pwbpreviouslogmatch** macro.

Definition _pwbpreviousmatch := cancel msearch <

Cancel
Establishes a uniform "ground state" by canceling any selection or argument.

Msearch
Searches backward in the file for the last search string or pattern.

<
Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Cancel**, **Msearch**

_pwbprevmsg

Key SHIFT+F4

Menu Project menu, Previous Error command

The **_pwbprevmsg** macro moves the cursor to the previous message in the Build Results pseudofile.

Definition _pwbprevmsg := cancel arg "-1" nextmsg <

Cancel
Establishes a uniform "ground state" by canceling any selection or argument.

Arg "-1" **Nextmsg**
Goes to the previous message in Build Results.

<

Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Arg**, **Cancel**, **Nextmsg**

_pwbprevwindow

Key SHIFT+F6

The **_pwbprevwindow** macro moves the focus to the previous window. That is, PWB sets the previously active window as the active window. This action moves among the open windows in the reverse order of **Selwindow** (F6).

Definition _pwbprevwindow:=cancel meta selwindow <

Cancel
Establishes a uniform "ground state" by canceling any selection or argument.

Meta Selwindow
Moves the focus to the previous window.

<

Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Cancel**, **Meta**, **Selwindow**

_pwbquit

Key ALT+F4

Menu File menu, Exit command

The **_pwbquit** macro leaves PWB immediately. Any specified files on the PWB command line that have not been opened are ignored.

Definition _pwbquit := cancel arg exit <

Cancel
Establishes a uniform "ground state" by canceling any selection or argument.

Arg Exit
Leaves PWB.

<
Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Arg**, **Askexit**, **Cancel**, **Exit**, **Savescreen**

_pwbrebuild

Key Unassigned

Menu Project menu, Rebuild All command

The **_pwbrebuild** macro forces a rebuild of everything in the current project.

For non-PWB projects, **_pwbrebuild** rebuilds the targets that were last specified by using the Build Target command on the Project menu. PWB redefines **_pwbrebuild** each time you use Build Target. If no target has been specified, NMAKE rebuilds the first target listed in the project makefile.

Definition _pwbrebuild := cancel arg meta "all" compile <

Cancel
Establishes a uniform "ground state" by canceling any selection or argument.

Arg Meta "all" **Compile**
Rebuilds the `all` pseudotarget.

<
Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Arg**, **Cancel**, **Compile**, **Meta**

_pwbrecord

Key Unassigned

Menu Edit menu, Record On command

The **_pwbrecord** macro toggles macro recording on and off. If you have not set the recorded macro name and key, PWB displays the Set Macro Record dialog box so you can set them. Execute **_pwbrecord** again to start recording.

Definition _pwbrecord := cancel record <

Cancel
Establishes a uniform “ground state” by canceling any selection or argument.

Record
Toggles macro recording on and off.

<
Restores the function’s prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Cancel**, **Record**

_pwbredo

Key Unassigned

Menu Edit menu, Redo command

The **_pwbredo** macro restores the last modification that was reversed using Edit Undo or **Undo** (ALT+BKSP).

Definition _pwbredo := cancel meta undo <

Cancel
Establishes a uniform “ground state” by canceling any selection or argument.

Meta Undo
Restores the last “undone” modification.

<
Restores the function’s prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Cancel**, **Meta**, **Undo**

_pwbrepeat

Key Unassigned

Menu Edit menu, Repeat command

The **_pwbrepeat** macro repeats the last editing operation once.

Definition _pwbrepeat := cancel repeat <

Cancel
Establishes a uniform "ground state" by canceling any selection or argument.

Repeat
Repeats the last operation one time.

<
Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Cancel**, **Repeat**

_pwbresize

Key CTRL+F8

Menu Window menu, Size command

The **_pwbresize** macro starts window-sizing mode for the active window. When in window-resizing mode, only the following actions are available:

Action	Key
Shrink one row	UP
Expand one row	DOWN
Shrink one column	LEFT
Expand one column	RIGHT
Accept the new size	ENTER
Cancel the resize	ESC

To size the window in larger increments, you can use the numeric forms of the **Resize** function.

Definition _pwbresize := cancel resize <

Cancel
Establishes a uniform "ground state" by canceling any selection or argument.

Resize
Starts window-sizing mode.

<

Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Arrangewindow**, **Cancel**, **Maximize**, **Minimize**, **Movewindow**

_pwbrestore

Key CTRL+F5

Menu Window menu, Restore command

The **_pwbrestore** macro restores the active window to its size before it was maximized or minimized.

Definition _pwbrestore := cancel meta maximize

Cancel
Establishes a uniform "ground state" by canceling any selection or argument.

Meta Maximize
Restores the window to its previous size.

See **Cancel**, **Maximize**, **Meta**, **Minimize**

_pwbsaveall

Key Unassigned

Menu File menu, Save All command

The **_pwbsaveall** macro saves all modified disk files. Modified pseudofiles are not saved.

Definition _pwbsaveall := cancel saveall <

Cancel
Establishes a uniform "ground state" by canceling any selection or argument.

Saveall
Writes all modified files to disk.

<

Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Cancel**, **Saveall**

_pwbsavefile

Key SHIFT+F2

Menu File menu, Save command

The **_pwbsavefile** macro saves the current file to disk.

If the current file is a pseudofile (an untitled file), PWB displays the Save As dialog box so you can give the file a more meaningful name.

Definition _pwbsavefile := cancel arg arg setfile <

Cancel
Establishes a uniform "ground state" by canceling any selection or argument.

Arg Arg Setfile
Writes the current file to disk.

<

Restores the function's prompt (if any). By default, function prompts are suppressed when a macro is running.

See **Arg**, **Cancel**, **Setfile**

_pwbsetmsg

Key Unassigned

Menu Project menu, Goto Error command

The **_pwbsetmsg** macro sets the message listed at the current location in the Build Results pseudofile as the current message and moves the cursor to the location specified by that message.

See **Nextmsg**

Definition _pwbsetmsg := cancel arg arg nextmsg <

Cancel
Establishes a uniform “ground state” by canceling any selection or argument.

Arg Arg Nextmsg
Goes to the current message.

<
Restores the function’s prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Arg**, **Cancel**, **Nextmsg**

_pwbshell

Key Unassigned

Menu File menu, DOS Shell command

The **_pwbshell** macro temporarily leaves PWB, starting a new operating-system shell. To return to PWB, type `exit` at the operating-system prompt.

Definition _pwbshell := cancel shell <

Cancel
Establishes a uniform “ground state” by canceling any selection or argument.

Shell
Starts an operating-system shell.

<
Restores the function’s prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Askrtn**, **Cancel**, **Savescreen**, **Shell**

_pwbstreammode

Key Unassigned

Menu Edit menu, Stream Mode command

The **_pwbstreammode** macro sets the selection mode to stream selection mode.

Definition _pwbstreammode := :>more selmode ->more

:>more
Defines the label `more`.

Selmode
Advances to the next selection mode.

->more
Branches to the label `more` if **Selmode** returns false.

The **Selmode** function advances the selection mode from box, to stream, to line. **Selmode** returns true when the mode is stream mode. The macro executes **Selmode** until it returns true (sets stream selection mode).

See **Enterselmode**, **Selmode**

_pwbtile

Key SHIFT+F5

Menu Window menu, Tile command

The **_pwbtile** macro tiles all unminimized windows on the desktop so that no windows overlap and the desktop is completely covered. Up to 16 unminimized windows can be tiled.

Definition _pwbtile := cancel meta arrangewindow <

Cancel
Establishes a uniform "ground state" by canceling any selection or argument.

Meta Arrangewindow
Tiles all unminimized windows.

<
Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

See **Arrangewindow**, **Cancel**, **Meta**

_pwbundo

Key Unassigned

Menu Edit menu, Undo command

The **_pwbundo** macro reverses the last modification made to the current file. The maximum number of modifications that can be undone for each file is determined by the **Undocount** switch.

Definition

_pwbundo := cancel undo <

Cancel
Establishes a uniform “ground state” by canceling any selection or argument.

Undo
Reverses the last modification.

<
Restores the function’s prompt (if any). By default, function prompts are suppressed while a macro is running.

See

Cancel, **_pwbredo**

_pwbusern

Macro	Description	Key
_pwbuser1	Run custom Run menu command 1	[ALT+F*n*]
_pwbuser2	.	[ALT+F*n*]
_pwbuser3	.	[ALT+F*n*]
_pwbuser4	.	[ALT+F*n*]
_pwbuser5	.	[ALT+F*n*]
_pwbuser6	.	[ALT+F*n*]
_pwbuser7	.	[ALT+F*n*]
_pwbuser8	.	[ALT+F*n*]
_pwbuser9	Run custom Run menu command 9	[ALT+F*n*]

Menu

Run *command*

command Title of custom Run menu item.

The **_pwbuser***n* macros execute custom commands in the Run menu.

To add a new command to the Run menu, use the Customize Run Menu command or assign a value to the **User** switch.

Definition

_pwbuser*n* := cancel arg "*n*" usercmd <

Cancel
Establishes a uniform "ground state" canceling any selection or argument.

Arg "*n*" **Usercmd**
Executes custom run menu item number *n*.

<
Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

Example

_pwbuser1 := cancel arg "1" usercmd <

This macro executes custom Run menu command number 1.

See

Arg, **Cancel**, **Usercmd**

_pwbviewbuildresults

Key

Unassigned

Button

The View Results button in the Build Operation Complete dialog box.

The **_pwbviewbuildresults** macro opens the Build Results window.

PWB executes this macro when you choose the View Results button in the Build Operation Complete dialog box.

You can redefine this macro to change the behavior of the View Results button. For example, if you want to move to the first message in the log and arrange windows, add `_pwbnextmsg _pwbarrangewindow` to the end of the macro definition.

Definition

_pwbviewbuildresults:=cancel arg "<COMPILE>" pwbwindow

Cancel
Establishes a uniform "ground state" by canceling any selection or argument.

Arg "<COMPILE>" **Pwbwindow**
Opens the Build Results window.

See

Pwbwindow

_pwbviewsearchresults

Key Unassigned

Button The View Results button in the Search Operation Complete dialog box.

The **_pwbviewSearchresults** macro opens the Search Results window.

PWB executes this macro when you choose the View Results button in the Search Operation Complete dialog box.

You can redefine this macro to change the behavior of the View Results button. For example, if you want to move to the first location in the log and arrange windows, add `_pwbnextsearch _pwbarrangewindow` to the end of the macro definition.

Definition _pwbviewsearchresults:=cancel arg “<SEARCH>” pwbwindow

Cancel
Establishes a uniform “ground state” by canceling any selection or argument.

Arg "<SEARCH>" **Pwbwindow**
Opens the Search Results window.

See **Pwbwindow**

_pwbwindown

Macro	Description	Key
_pwbwindow1	Switch to window 1	ALT+1
_pwbwindow2	.	ALT+2
_pwbwindow3	.	ALT+3
_pwbwindow4	.	ALT+4
_pwbwindow5	.	ALT+5
_pwbwindow6	.	ALT+6
_pwbwindow7	.	ALT+7
_pwbwindow8	.	ALT+8
_pwbwindow9	Switch to window 9	ALT+9

Menu

Window *n file*

n Window number

file Current file in the window

The **_pwbwindow***n* macros each set a specific numbered window as the active window.

Definition

_pwbwindow*n* := cancel arg "*n*" selwindow <

Cancel
Establishes a uniform "ground state" by canceling any selection or argument.

Arg "*n*" **Selwindow**
Moves to window number *n*.

<
Restores the function's prompt (if any). By default, function prompts are suppressed while a macro is running.

Example

_pwbwindow1 := cancel arg "1" selwindow <

This macro sets window number 1 as the active window.

See

Arg, **Cancel**, **Selwindow**

PWB Switches

PWB provides the following switches to customize its behavior. You set switches by adding entries to the TOOLS.INI file or by using the Editor Settings, Key
Assignments, and Colors commands on the Options menu.

Switch	Description
Askexit	Prompt before leaving PWB
Askrtn	Prompt before returning from a shell
Autoload	Load PWB extensions automatically
Autosave	Save files when switching
Backup	File backup mode
Beep	Issue audible or visible alerts
Build	Rules and definitions for the build process
Case	Make letter case significant in searches

Color	Color of interface elements

Switch	Description
Cursormode	Block or underline cursor state
Dblclick	Double-click threshold
Deflang	Default language
Defwinstyle	Default window style
Editreadonly	Allow editing of files marked read-only on disk
Enablealtgr	Enable the ALTGR key on non-US keyboards
Entab	Tab translation mode while editing
Enterinsmode	Enter PWB in insert mode
Enterlogmode	Enter PWB with search logging turned on
Enterselmode	Enter PWB in specified selection mode
Envcursave	Save environment variables for PWB sessions
Envprojsave	Save environment variables for projects
Factor	Auto-repeat factor
Fastfunc	Functions for fast auto-repeat
Filetab	Width of tab characters in the file
Friction	Delay between repetitions of fast functions
Height	Height of the display
Hike	Window adjustment factor
Hscroll	Horizontal scrolling factor
Infodialog	Set of information dialogs displayed
Keepmem	XMS/EMS memory kept during shell, build, and compile
Lastproject	Set the last project on startup
Load	PWB extension to load
Markfile	Name of the current mark file
Mousemode	Mouse configuration; disabled or swapped buttons
Msgdialog	Display a dialog box for build results
Msgflush	Keep only one set of build results
Newwindow	Create a new window when opening a file
Noise	Line counting interval
Printcmd	Command for printing files
Readonly	Command for saving disk read-only files
Realtabs	Preserve tab characters in the file
Restorelayout	Restore the window layout when a project is set
Rmargin	Right margin for word wrap
Savescreen	Preserve the operating-system screen

Switch	Description
Searchdialog	Display a dialog box for search results
Searchflush	Keep only one set of search results
Searchwrap	Make searches wrap around the end of the file
Shortnames	Allow access to loaded files by base name
Softcr	Perform automatic indenting
Tabalign	Align the cursor in tab fields
Tabdisp	Character for displaying tab characters
Tabstops	Variable tab stops
Tilemode	Window tiling style
Timersave	Timer interval for saving files
Tmpsav	Number of files kept in file history
Traildisp	Character for displaying trailing spaces
Traillines	Preserve trailing lines
Traillinesdisp	Character for displaying trailing lines
Trailspace	Preserve trailing spaces
Undelcount	Maximum number of file backups
Undocount	Maximum number of edits per file to undo
Unixre	Use UNIX regular-expression syntax
User	Custom Run menu item
Vscroll	Vertical scrolling factor
Width	Width of the display
Word	Definition of a word
Wordwrap	Wrap words as they are entered

Extension Switches

The standard PWB extensions define additional switches to control their behavior. You set these switches in tagged sections of TOOLS.INI specific to that extension.

PWB Extension	Description	TOOLS.INI Section Tag
PWBROWSE.MXT	Source Browser	[PWB-PWBROWSE]
PWBMASM.MXT	Assembly Language	[PWB-PWBMASM]
PWBHELP.MXT	Microsoft Advisor Help	[PWB-PWBHELP]

The PWBROWSE switches are described in “Browser Switches” on page 286. The PWBHELP switches are described in “Help Switches” on page 287.

Filename-Parts Syntax

The filename-parts syntax is used by PWB to pass the name of the current file to external programs or operating-system commands. You use this syntax in the **Printcmd**, **Readonly**, and **User** switches.

Syntax **%%**

A literal percent sign (**%**).

Syntax **%s**

The fully qualified path of the current file. If the current file is a pseudofile, **%s** specifies the name of a temporary disk file created for the external command to operate on. The temporary file is destroyed before returning to PWB and is never accessible to the editor.

Syntax **%| [[d]][[p]][[f]][[e]]F**

Parts of the current filename. The parts of the name are drive, path, filename, and extension. If the current file is a disk file named:

```
C:\SCRATCH\TEST.TXT
```

or the pseudofile:

```
"<COMPILE>Build Results"
```

the given syntax yields:

Syntax	Disk File	Pseudofile
%|F	C:\SCRATCH\TEST.TXT	<COMPILE>
%|dF	C:	
%|pF	\SCRATCH	
%|fF	TEST	<COMPILE>
%|eF	.TXT	
%|pfF	\SCRATCH\TEST	<COMPILE>
%s	C:\SCRATCH\TEST.TXT	C:\TMP\PWB00031.R00
%%	%	%

The title of a pseudofile cannot be specified with the filename-parts syntax, but it is accessible to macros by using the **Curfile** predefined macro.

Warning The **%|F** syntax always specifies the name of the current file in the active window. For some commands, such as the command specified in the **Readonly** switch, this may not be the desired file. Use **%s** for the **Readonly** switch.

See **Printcmd**, **Readonly**, **User**

Boolean Switch Syntax

You can use either one of the following syntaxes to set Boolean switches in PWB:

Syntax 1 *switch* : ⟦ **yes** | **no** | **on** | **off** | **1** | **0** ⟧

switch
The name of a PWB switch.

yes, on, 1
Enable the feature controlled by *switch*.

no, off, 0
Disable the feature controlled by *switch*.

Syntax 2 ⟦**no**⟧*switch* **:**

switch
Enable the feature controlled by *switch*.

no*switch*
Disable the feature controlled by *switch*.

Askexit

Type Boolean

The **Askexit** switch determines if PWB prompts for confirmation before returning to the operating system.

Syntax **Askexit**:{ **yes** | **no** }

yes Prompt for confirmation before leaving PWB.

no Do not prompt before leaving PWB.

Default Askexit:no

See **Exit**

Askrtn

Type Boolean

The **Askrtn** switch determines if PWB prompts before returning to PWB after running a shell or external command.

Syntax **Askrtn**:{ **yes** | **no** }

yes
Prompt for confirmation before returning to PWB. This setting allows you to review the contents of the operating-system screen before returning to the editor.

no
Do not prompt before returning to PWB.

Default Askrtn:yes

See **Shell**

Autoload

Type Boolean

The **Autoload** switch determines if PWB automatically loads its extensions on startup.

When the **Autoload** switch is `yes`, PWB automatically loads extensions whose names begin with "PWB" and are found in the same directory as PWB.EXE. PWB always loads extensions named in a **Load** switch.

If you disable automatic extension loading, you can load extensions as you need them by assigning a value to the Load switch as follows:
Arg `load:` pwbextension Assign (ALT+A `load:`*pwbextension* ALT+=).

The *pwbextension* is the path of the extension's executable file. PWB automatically assumes the filename extension .MXT. You can specify an environment-variable path by using an environment-variable specifier.

Syntax **Autoload**:{ **yes** | **no** }

yes
Automatically load PWB extensions on startup.

no
Do not automatically load PWB extensions on startup. Load only those extensions named in **Load** switches in TOOLS.INI.

Default Autoload:yes

Update PWB 1.*x* extensions are not compatible with PWB 2.0. They are refused when you request that they be loaded. Old extensions must be recompiled with the new extension-support libraries and header files. In some cases, old extensions must also be modified for use with PWB 2.00.

Updated Microsoft PWB 1.*x* extensions not included with this product are available by contacting Microsoft Product Support Services in the United States or your local Microsoft subsidiary.

Autosave

Type Boolean

The **Autosave** switch determines if PWB automatically saves the current file without prompting whenever you move to another file, exit PWB, or execute an external operation such as a shell, build, or compile.

When the **Autosave** switch is set to no, PWB maintains the contents of files in memory for internal operations, and prompts to save modified files on exit or for external operations such as a build. With this setting, PWB never saves a file unless you explicitly save it.

Syntax **Autosave**:{ **yes** | **no** }

yes Automatically save files.

no Do not automatically save files.

Default Autosave:no

Update In PWB 1.*x*, the default value of **Autosave** is yes.

See **Shell**, **Timersave**

Backup

Type Text

The **Backup** switch determines what happens to the old copy of a file before the new version is saved to disk.

Syntax **Backup**:[[*undel* | *bak*]]

(none)
No backup: PWB overtypes the file.

undel
PWB moves the old file to a hidden directory so you can retrieve it with the UNDEL utility. The number of copies saved is specified by the **Undelcount** switch.

bak
The extension of the previous version of the file is changed to .BAK.

Default Backup:bak

Beep

Type Boolean

The **Beep** switch determines PWB's alerting method. When set to yes, PWB issues an audible sound. When no, PWB flashes the menu bar—a visual "beep."

Syntax **Beep**:{ **yes** | **no** }

yes Generate an audible beep.

no Flash the menu bar.

Default Beep:yes

Case

Type Boolean

The **Case** switch determines if letter case is distinguished in searches.

The search functions that use the **Case** switch have “meta” forms that temporarily reverse the sense of the **Case** switch.

The **Unixre** and **Case** switches have no effect on the syntax of regular expressions used by the **Build** or **Word** switches. These switches always use case-sensitive UNIX regular expressions.

Syntax

Case:{ **yes** | **no** }

yes
Case is significant in searches. Uppercase letters in search patterns do not match lowercase letters in text.

no
Case is not significant in searches. Uppercase letters match lowercase letters.

Default

Case:no

See

Meta, **Mgrep**, **Mreplace**, **Msearch**, **Psearch**, **Replace**.

Color

Type

Text

The **Color** switch specifies color of various parts of the PWB display.

Syntax

Color:*name value*

name
Identifies the part of PWB affected by the color value.

value
Two hexadecimal digits specifying the foreground and background color of the indicated item.

Color Names

PWB uses the following color names and default color values for the various parts of the PWB display:

Table 7.13 PWB Color Names

Name	Default Value	Description
Alert	70	Message box
Background	07	(Not visible)
Border	07	Window borders

Table 7.13 PWB Color Names (*continued*)

Name	Default Value	Description
Builderr	40	Build message line in active window
Buttondown	07	Button while it is down
Desktop	80	Desktop
Dialogaccel	7F	Dialog box accelerator
Dialogaccelbor	7F	Dialog box accelerator border
Dialogbox	70	Dialog box
Disabled	78	Disabled items in menus and dialogs
Elevator	07	Scroll box
Enabled	70	Available items in menus and dialogs
Greyed	78	(Not visible)
Helpbold*	8F	Bold Help text
Helpitalic*	8A	Italic Help text and the characters
Helpnorm*	87	Plain Help text
Helpunderline*	8C	Emphasized Help text
Helpwarning*	70	Current hyperlink
Highlight	1F	Highlighted text; text found by searches
Hilitectrl	07	Highlighted control item
Info	3F	Special information
Itemhilitesel	0F	Highlighted character in selected item
Listbox	70	List box within a dialog box
Location	70	Location indicator in status bar
Menu	70	Menu bar
Menubox	70	Menu
Menuhilite	7F	Highlighted character in menu
Menuhilitesel	0F	Highlighted character in selected menu
Menuselected	07	Selected menu
Message	70	Message area of status bar
Pushbutton	70	Button that is not pressed
Pwbwindowborder	07	PWB window borders
Pwbwindowtext	87	PWB window text
Scratch	07	(Not visible)
Scrollbar	70	Gray area and arrows in scroll bar
Selection	71	Current selection
Shadow	08	Shadows

Table 7.13 PWB Color Names (*continued*)

Name	Default Value	Description
Status	7F	Indicator area of status bar
Text	17	Text in a window

* Defined by the Help extension. Define these settings in the [PWB-PWBHELP] section of TOOLS.INI.

Color Values

Color values for the **Color** switch are two hexadecimal digits that specify the color of the item. The first digit specifies the background color and the second digit specifies the foreground color, according to the following table:

Table 7.14 PWB Color Values

Color	Digit	Color	Digit
Black	0	Dark Gray	8
Blue	1	Bright Blue	9
Green	2	Bright Green	A
Cyan	3	Bright Cyan	B
Red	4	Bright Red	C
Magenta	5	Bright Magenta	D
Brown	6	Yellow	E
White	7	Bright White	F

For example, a setting of 3E displays a cyan background (3) and a yellow foreground (E).

Note that only color displays support all the colors listed. If you have a monochrome adapter or monochrome monitor, the only available colors are black (0), white (7), and bright white (F). All other colors are displayed as white.

Cursormode

Type Numeric

The **Cursormode** switch determines the shape of the cursor when PWB is in insert and overtype mode, according to the following table:

Cursormode Value	Insert Mode Cursor	Overtype Mode Cursor
0	Underscore	Underscore
1	Block	Block
2	Block	Underscore
3	Underscore	Block

Default Cursormode:2

See **Status Bar**

Dblclick

Type Numeric

The **Dblclick** switch sets the double-click threshold for the mouse (the maximum time between successive clicks of the mouse button). The units for the **Dblclick** switch are 1/18 of a second.

Default Dblclick:10

See **Mousemode**

Deflang

Type Text

The **Deflang** switch determines the default file extension for file lists in PWB dialog boxes.

Syntax Deflang:*language*

language

One of the following settings:

Setting	Extension
NONE	.*
Asm	.ASM
Basic	.BAS
C	.C

Setting	Extension
C++	.CPP
CXX	.CXX
COBOL	.CBL
FORTRAN	.FOR
LISP	.LSP
Pascal	.PAS

Default Deflang:NONE

Defwinstyle

Type Numeric

The **Defwinstyle** switch sets the default window style. The possible values for **Defwinstyle** are:

Value	Style
1	No scroll bars
3	Vertical scroll bar
5	Horizontal scroll bar
7	Both scroll bars

You can change the active window style by using the **Winstyle** function (CTRL+F6).

Default Defwinstyle:7

See **Maximize**

Editreadonly

Type Boolean

The **Editreadonly** switch determines if PWB allows you to edit a file marked read-only on disk.

Syntax **Editreadonly**:{ **yes** | **no** }

yes
Allow modification of files that are marked read-only on disk. When PWB attempts to save the modified file, PWB informs you that the file is marked read-only. It also prompts you to confirm that the command specified by the **Readonly** switch is to be run. If you decline to run the command, PWB gives you the opportunity to save the file with a different name.

no
Disallow modification of read-only files. For files that cannot be modified, PWB displays the letter R on the status bar. You can reenable modification of a read-only file by using the Read Only command on the Edit menu or the **Noedit** function.

Default Editreadonly:yes

Enablealtgr

Type Boolean

The **Enablealtgr** switch determines if PWB recognizes the ALTGR key (the right ALT key) on international keyboards as ALTGR (Graphic Alt) or ALT.

When ALTGR is enabled, pressing ALTGR+*key* produces the corresponding graphic character. ALTGR is never recognized as a key name for use in PWB key assignments.

Syntax **Enablealtgr**:{ **yes** | **no** }

yes Recognize the right ALT key as ALTGR.

no Recognize the right ALT key as ALT.

Default Enablealtgr:no

Entab

Type Numeric

The **Entab** switch controls how PWB converts white space on modified lines. PWB converts white space only on the lines that you modify.

When the **Realtabs** switch is set to yes, tab characters are converted. When set to no, tab characters are not converted.

The **Entab** switch can have the following values:

Value	Meaning
0	Convert all white space to space (ASCII 32) characters.
1	Convert white space outside quoted strings to tabs. A quoted string is any span of characters enclosed by a pair of single quotation marks or a pair of double quotation marks. PWB does not recognize escape sequences because they are language-specific. For well-behaved conversions with this setting, make sure that you use a numeric escape sequence to encode quotation marks in strings or character literals.
2	Convert white space to tabs.

With settings 1 and 2, if the white space being considered for conversion to a tab character occupies an entire tab field or ends at the boundary of a tab field, it is converted to a tab (ASCII 9) character. The width of a tab field is specified by the **Filetab** switch.

In all conversions, PWB maintains the text alignment as it is displayed on screen.

Default Entab:1

See **Filetab**, **Realtabs, Tabalign**

Enterinsmode

Type Boolean

The **Enterinsmode** switch determines if PWB is to start in insert mode or overtype mode. You can toggle the current mode by using the **Insertmode** function (INS).

When the current mode is overtype mode, the letter O appears on the status bar. Depending on the setting of the **Cursormode** switch, the shape of the cursor reflects the current mode.

Syntax **Enterinsmode**:{ **yes** | **no** }

yes Start PWB in insert mode.

no Start PWB in overtype mode.

Default Enterinsmode:yes

Enterlogmode

Type Boolean

The **Enterlogmode** switch determines if search logging is turned on or off when PWB starts up. The current search-logging mode can be changed at any time using the Log command on the Search menu or the **Logsearch** function (Unassigned).

Syntax **Enterlogmode**:{ **yes** | **no** }

yes Start PWB with search logging on.

no Start PWB with search logging off.

Default Enterlogmode:no

Enterselmode

Type Text

The **Enterselmode** switch determines the selection mode when PWB starts up.

Syntax **Enterselmode**:{ **stream** | **box** | **line** }

stream
Starts PWB in stream selection mode.

box
Starts PWB in box selection mode.

line
Starts PWB in line selection mode.

Default Enterselmode:stream

See **Selmode**

Envcursave

Type Boolean

The **Envcursave** switch determines if PWB saves and restores the current environment table for PWB sessions.

You can change environment variables by using the Environment command on the Options menu or the **Environment** function (Unassigned).

If you always want to use the operating-system environment, set both **Envcursave** and **Envprojsave** to no.

Syntax

Envcursave:{ **yes** | **no** }

yes
Save and restore environment variables for PWB sessions. Use this setting if you want to use an environment that is specific to PWB. The PWB environment overrides the operating-system environment.

no
Do not save environment variables between PWB sessions.

Default

Envcursave:no

Update

In PWB 1.*x*, the INCLUDE, LIB, and HELPFILES environment variables were always saved for PWB sessions and projects.

Envprojsave

Type

Boolean

The **Envprojsave** switch determines if PWB saves and restores the environment table for each project. A project's environment overrides both the PWB environment and the external (operating-system) environment.

If you always want to use the operating-system environment table, set both **Envcursave** and **Envprojsave** to no. You can change environment variables by using the Environment command on the Options menu or the **Environment** function (Unassigned).

Syntax

Envprojsave:{ **yes** | **no** }

yes
Save environment variables for the project. Use this setting if you want to set project-specific environments.

no
Do not save environment variables for the project.

Default

Envprojsave:yes

Update

In PWB 1.*x*, the INCLUDE, LIB, and HELPFILES environment variables were always saved for PWB sessions and projects.

Factor

Type

Text

The **Factor** switch, together with the **Friction** switch, controls how quickly PWB executes a fast function. A fast function is a PWB function whose action repeats as rapidly as possible while you hold down the associated keystroke.

Syntax

Factor:{ **%***percent* | **-***constant* } [[*count*]]

percent
Percentage between 0 and 100 to reduce friction.

constant
Constant value between 0 and 65,535 to reduce friction.

count
Interval between reductions of friction.

PWB reduces friction by *percent* percent or *constant* every *count* repetition of a keystroke, until friction is zero.

Default

Factor:%50 10

Example

If you hold down the RIGHT ARROW key with the settings:

```
Right   :RIGHT
Fastfunc:Right
Friction:1000
Factor  :%75 7
```

PWB moves the cursor at the current speed until it has moved seven characters to the right. Then PWB changes the friction to 250 (75 percent reduction of the initial friction of 1000). When the cursor has moved 14 characters, the friction changes to 188 (75 percent reduction of the friction of 250). The cursor moves faster the longer you hold down the RIGHT ARROW key.

See

Fastfunc

Fastfunc

Type Text

The **Fastfunc** switch specifies functions whose action is rapidly repeated by PWB as you hold down the associated key combination.

The **Friction** and **Factor** switches control the repeat speed and acceleration of fast functions.

Syntax **Fastfunc**:*function* {**on** | **off**}

function PWB function to repeat.

on Enable fast repeat for *function*.

off Disable fast repeat for *function*.

Default
Fastfunc:Down on
Fastfunc:Left on
Fastfunc:Mlines on
Fastfunc:Mpage on
Fastfunc:Mpara on
Fastfunc:Mword on
Fastfunc:Plines on
Fastfunc:Ppage on
Fastfunc:Ppara on
Fastfunc:Pword on
Fastfunc:Right on
Fastfunc:Up on

Filetab

Type Numeric

The **Filetab** switch determines the width of a tab field for displaying tab (ASCII 9) characters in the file. The width of a tab field determines how white space is translated when the **Realtabs** switch is set to no. The **Filetab** switch does not affect the cursor-movement functions **Tab** (TAB) and **Backtab** (SHIFT+TAB).

Default Filetab:8

See **Entab**, **Realtabs**, **Tabdisp**

Friction

Type Numeric

The **Friction** switch, together with the **Factor** switch, controls how quickly PWB executes a fast function. A fast function is a PWB function whose action repeats rapidly when you hold down the associated key.

The value of the **Friction** switch is a decimal number between 0 and 65,535 and specifies the delay between repetitions of a fast function. As the function is repeated, the delay is reduced according to the setting of the **Factor** switch.

Default Friction:40

See **Factor**, **Fastfunc**

Height

Type Numeric

The **Height** switch determines the number of lines on the PWB screen. The **Height** switch can have one of these values: 25, 43, 50, or 60. The last setting of this switch is saved and restored across PWB sessions and for each project.

Default Height: *first screen height*

When you start PWB for the first time, PWB uses the current screen height. Thereafter, PWB restores the previous setting until you explicitly assign a new value to the **Height** switch.

Note that when you change the setting for **Height** in the Editor Settings dialog box, the change does not take effect until you choose OK. Other switches takes effect immediately when you choose Set Switch.

See **Assign**

Hike

Type Numeric

The **Hike** switch determines the number of lines from the cursor to the top of the window after you move the cursor out of the window by more than the number of lines specified by the **Vscroll** switch.

The minimum value is 1. When the window occupies less than the full screen, the value is reduced in proportion to the window size.

Default Hike:4

See **Hscroll**

Hscroll

Type Numeric

The **Hscroll** switch controls the number of columns that PWB scrolls the text left or right when you move the cursor out of the window. When the window does not occupy the full screen, the amount scrolled is in proportion to the window size.

Text is never scrolled in increments greater than the size of the window.

Default Hscroll:10

See **Vscroll**

Infodialog

Type Numeric

The **Infodialog** switch determines which information dialog boxes are displayed.

Syntax

Infodialog:*hh*

hh
Two hexadecimal digits specifying a set of flags to indicate which information dialog boxes should be displayed. When a bit is on (1), the corresponding dialog box is displayed. When a bit is off (0), the corresponding dialog box is not displayed.

To set the value of **Infodialog**, add up the hexadecimal numbers listed in the table below for the dialog boxes you want to display.

Value	Information Dialog
01	*n* occurrences found *n* occurrences replaced
02	End of Build Results End of Search Results
04	`'`pattern`'` not found
08	No unbalanced characters found
10	Changed directory to *directory* Changed drive to *drive*

Default

Infodialog:0F

The default value of **Infodialog** tells PWB to display all information dialog boxes except for the "Changed..." dialog boxes.

Keepmem

Type

Numeric

The **Keepmem** switch specifies the amount of extended (XMS) memory or expanded (EMS) memory kept by PWB during a shell, compile, build, or other external command. Specify the value in units of kilobytes (1024 bytes).

A larger number means that shelling is faster and leaves less memory for tools that use extended or expanded memory. A smaller number means that shelling is slower and leaves more memory for tools. If the number you specify is not large enough, PWB uses no extended or expanded memory.

Default

Keepmem:2048

Lastproject

Type Boolean

The **Lastproject** switch determines if PWB automatically opens the last project on startup. The /PN, /PP, /PL, and /PF command-line options override the setting of the **Lastproject** switch.

Syntax **Lastproject**:{ **yes** | **no** }

yes On startup, open the last project that was open.

no Do not open the last project on startup.

Default Lastproject:no

See **Project**

Load

Type Text

The **Load** switch specifies the filename of a PWB extension to load.

When this switch is assigned a value, PWB loads the specified extension. The initialization specified in the extension is performed, and the functions and switches defined by the extension become available in PWB.

The extension can be loaded during initialization of a TOOLS.INI section. You can also interactively load an extension by using the Editor Settings command on the Options menu or by using the **Assign** function to assign a value to the **Load** switch.

Syntax **Load**:[[*path*]]*basename*[[**.***ext*]]

path
Can be a path or an environment-variable specifier.

basename
Base name of the extension executable file.

ext
Normally you do not specify a filename extension.

See **Autoload**

Markfile

Type

Text

The **Markfile** switch specifies the name of the file PWB uses to save marks.

When no mark file is open, marks are kept in memory, and they are lost when you exit PWB. When you open a mark file, marks in memory are saved in the mark file, unless a mark file is already open. When a mark file is already open, the marks in memory are saved in the open file.

To open a mark file, use the Set Mark File command on the Search menu or assign a value to the **Markfile** switch by using the Editor Settings command on the Options menu or the **Assign** function. To close a mark file without opening a new one, assign an empty value to the **Markfile** switch. That is, use the setting:

```
Markfile:
```

To set a permanent mark file that is used for every PWB session, place a **Markfile** definition in the [PWB] section of TOOLS.INI.

Syntax

Markfile: *filename*

filename The name of the file containing mark definitions.

Default

Markfile:

The Markfile switch has no default value and is initially undefined.

See

Assign, **Mark**

Mark File Format

A mark file is a text file containing mark definitions of the form:

markname filename line column

The mark *markname* is defined as the location given by *line* and *column* in the file *filename*. The *markname* cannot contain spaces and cannot be a number.

Update

With PWB 1.*x*, when you open a mark file and no mark file is currently open, the marks in memory are lost. With PWB 2.00, the marks are saved in the new mark file.

Mousemode

Type Numeric

The **Mousemode** switch enables or disables the mouse and sets the actions of the left and right mouse buttons.

Value	Description
0	The mouse is disabled and the mouse pointer is not visible.
1	Normal mouse control.
2	Exchanges the actions of the left and right mouse buttons.

Default Mousemode:1

See **Dblclick**

Msgdialog

Type Boolean

The **Msgdialog** switch determines if PWB brings up a dialog box summarizing build results or only beeps when a build is complete.

Syntax **Msgdialog**:{ **yes** | **no** }

yes Display a dialog box summarizing build results when a build is complete.

no Beep when a build is complete.

Default Msgdialog:yes

See **Beep**, **Compile**, **Searchdialog**

Msgflush

Type Boolean

The **Msgflush** switch determines if previous build results are retained in the Build Results window or flushed when a new build is started.

Syntax **Msgflush**:{ **yes** | **no** }

yes Flush previous build results when a new build is started.

no Save previous build results.

Default Msgflush:yes

See **Nextmsg**, **Searchflush**

Newwindow

Type Boolean

The **Newwindow** switch determines if certain PWB functions open a file in a new window or in the active window. The **Newwindow** switch provides the default state of the New Window check box in the Open File dialog box. This check box does not change the value of the **Newwindow** switch.

When **Newwindow** is set to yes, PWB behaves like a Multiple Document Interface (MDI) application. That is, when you open a new file, PWB opens a new window for the file, except in certain situations as noted below.

When **Newwindow** is set to no, PWB behaves like PWB 1.*x*. In this case, PWB opens files into the active window, creating a file history for that window. This mode is useful when working with large numbers of files.

Some functions use the **Newwindow** switch to determine if a new window is created when opening a file.

The following functions ignore the **Newwindow** switch, and either create a new window or open the file into the active window:

Function	Creates a New Window
Mreplace	No
Openfile	Yes
Setfile	No
Nextmsg	No
Nextsearch	No

When the active window is a PWB window, PWB always creates a new window. You cannot open a file into a PWB window.

Syntax **Newwindow**:{ **yes** | **no** }

yes
Open a new window when a new file is opened. This setting makes PWB behave like other MDI applications such as Microsoft Word 5.5 and Microsoft Works.

no
Open files into the active window, adding the previous file to the window's file history. This setting makes PWB behave like PWB 1.*x*.

Default Newwindow:yes

See **Exit**, **Mark**, **Mreplace**, **Newfile**, **Nextmsg**, **Nextsearch**, **Openfile**, **Setfile**

Noise

Type Numeric

The **Noise** switch specifies the number of lines counted at a time as PWB traverses a file while reading, writing, or searching. PWB displays the line counter on the right side of the status bar, in the area which usually shows the current line.

Set **Noise** to 0 to turn off the display of scanned lines.

Default Noise:50

Printcmd

Type Text

The **Printcmd** switch specifies a program or operating system command that PWB starts when you choose the Print command from the File menu or execute the **Print** function (Unassigned).

Syntax **Printcmd**: *command_line*

command_line An operating-system command line.

To pass the filename of the current file, specify **%s** in the command line. Specify **%%** to pass a literal percent sign. You can extract parts of the full filename using a special PWB syntax. See "Filename-Parts Syntax" on page 247.

Default Printcmd:COPY %s PRN

See **Print**

Readonly

Type Text

The **Readonly** switch specifies the operating-system command invoked when PWB attempts to write to a read-only file.

When PWB attempts to overtype a file that is marked read-only on disk, PWB informs you that the file is read-only. It also prompts you to confirm that the command specified in the **Readonly** switch is to be run. If you decline to run the **Readonly** command, PWB gives you the opportunity to save the file with a different name.

Syntax

Readonly:[[*command*]]

command Operating-system command line.

If no command is specified, PWB prompts you to enter a new filename to save the file.

To pass the filename of the current file to the command, specify **%s** in the command line. Specify **%%** to pass a literal percent sign. You can extract parts of the full path using a special PWB syntax. See "Filename-Parts Syntax" on page 247.

Note that only **%s** is guaranteed to give the name of the read-only file. The **%|F** syntax gives the current filename (the file displayed in the active window), even when PWB is saving a different file.

Default

Readonly:

The default value specifies that PWB should run no command and should prompt for a different filename.

Example

The **Readonly** switch setting

```
Readonly:Attrib -r %s
```

removes the read-only attribute from the file on disk so PWB can overtype it.

See

Editreadonly, **Noedit**

Realtabs

Type

Boolean

The **Realtabs** switch determines if PWB preserves tab (ASCII 9) characters or translates white space according to the **Entab** switch when a line is modified. **Realtabs** also determines if the **Tabalign** switch is in effect.

Syntax

Realtabs:{ **yes** | **no** }

yes Preserve tab characters when editing a line.

no Translate tab characters when editing a line.

Default Realtabs:yes

See **Entab**, **Filetab**, **Tabalign**

Restorelayout

Type Boolean

The **Restorelayout** switch determines if PWB restores the saved window layout and file history from the project status file when you open a project or retains the active window layout and file history.

This switch provides the default state of the Restore Window Layout check box in the Open Project dialog box.

Syntax **Restorelayout**:{ **yes** | **no** }

yes
Restore a project's saved window layout and file history when the project is opened.

no
Do not restore the project's windows and file history.

Default Restorelayout:yes

See **Project**

Rmargin

Type Numeric

The **Rmargin** switch sets the right margin for word wrapping. It has an effect only when word wrapping is turned on.

Default Rmargin:78

Update In PWB 1.*x*, **Rmargin** sets the beginning of a six-character "probation" zone where typing a space wraps the line. After the zone, typing any character wraps the current word. This behavior is similar to that of a typewriter. PWB 2.00 uses
a word-processor's style of wrapping.

To maintain the same margins as PWB 1.*x*, increase your **Rmargin** settings by 6.

See **Softcr**, **Wordwrap**

Savescreen

Type Boolean

The **Savescreen** switch determines if PWB preserves the operating-system screen image and video mode.

Syntax **Savescreen**:{ **yes** | **no** }

yes
Save the operating-system screen when starting PWB, and restore it when leaving PWB.

no
Do not preserve the operating-system screen. When you leave PWB, the operating-system screen is blank, and the video mode is left in PWB's last video mode.

Default Savescreen:yes

Searchdialog

Type Boolean

The **Searchdialog** switch determines if PWB brings up a dialog box that summarizes logged search results or only beeps when a logged search is complete. The **Searchdialog** switch has an effect only while logging search results.

Syntax **Searchdialog**:{ **yes** | **no** }

yes
Display a dialog box summarizing search results when a logged search is complete.

no
Beep when a logged search is complete.

Default Searchdialog:yes

See **Beep**, **Enterlogmode**, **Logsearch**, **Msgdialog**

Searchflush

Type Boolean

The **Searchflush** switch determines if previous logged search results are flushed or retained when you start a new logged search.

This switch has an effect only when PWB performs a logged search.

Syntax **Searchflush**:{ **yes** | **no** }

yes
Flush the previous search results from the Search Results window when a new search is begun.

no
Preserve previous search results in the Search Results window.

Default Searchflush:yes

See **Logsearch**, **Mgrep**

Searchwrap

Type Boolean

The **Searchwrap** switch determines if search commands and replace commands wrap around the ends of a file.

Syntax **Searchwrap**:{ **yes** | **no** }

yes Searches wrap around the beginning and end of the file.

no Searches stop at the beginning and end of the file.

Default Searchwrap:no

See **Msearch**, **Psearch**, **Replace**.

Shortnames

Type Boolean

The **Shortnames** switch determines if currently loaded files can be accessed by their short names (base name only).

Syntax **Shortnames**:{ **yes** | **no** }

yes
You can switch to a file currently loaded into PWB by specifying only the base name to the **Setfile** (F2) or **Openfile** (F10) functions.

no
You must specify the extension as well as the base name to switch to a file.

Default Shortnames:yes

See **Openfile**, **Setfile**

Softcr

Type Boolean

The **Softcr** switch controls indentation of new lines based on the format of surrounding text when you execute the **Emacsnewl** (ENTER) and **Newline** (SHIFT+ENTER) functions.

Syntax **Softcr**:{ **yes** | **no** }

yes
Indent new lines.

no
Do not indent new lines. After executing **Emacsnewl** or **Newline**, the cursor is placed in column 1.

Default Softcr:yes

Tabalign

Type Boolean

The **Tabalign** switch determines the positioning of the cursor when it enters a tab field. A tab field is the area of the screen representing a tab character (ASCII 9) in the file. The width of a tab field is specified by the **Filetab** switch.

The **Tabalign** switch takes effect only when the **Realtabs** switch is set to yes.

Syntax **Tabalign**:{ **yes** | **no** }

yes
PWB aligns the cursor to the beginning of the tab field when the cursor enters the tab field. The cursor is placed on the actual tab character in the file.

no
PWB does not align the cursor within the tab field.

You can place the cursor on any column in the tab field. When you type a character at this position, PWB inserts enough leading blanks to ensure that the character appears in the same column.

Default Tabalign:no

Tabdisp

Type Numeric

The **Tabdisp** switch specifies the decimal ASCII code of the character used to display tab (ASCII 9) characters in your file. If you specify 0 or 255, PWB uses the space (ASCII 32) character.

It is sometimes useful to set **Tabdisp** to the code for a graphic character so that tabs can be distinguished from spaces.

Default Tabdisp:32

The default value 32 specifies the ASCII space character.

See **Filetab**, **Realtabs**, **Traildisp**, **Traillinesdisp**

Tabstops

Type Text

The **Tabstops** switch specifies variable tab stops used by the **Tab** and **Backtab** functions. **Tab** moves the cursor to the next tab stop; **Backtab** moves the cursor
to the previous tab stop.

Note that the **Tabstops** switch has no effect on the handling of tab (ASCII 9) characters in a file.

Syntax **Tabstops**: [[*tabwidth*]]... *repeat*

tabwidth
The width of a tab stop. You can repeat *tabwidth* for as many tab stops as will fit on a PWB line (250 characters).

repeat
The width of every tab stop after the explicitly listed tab stops.A value of 0 for *repeat* specifies that there are no tab stops after the list of *tabwidth* settings. When the cursor is past the last tab stop, the **Tab** function does nothing.

Default Tabstops:4

Update In PWB 1.*x*, **Tabstops** is a numeric switch specifying a single value, equivalent to the repeat value in PWB 2.0. The default PWB 2.00 **Tabstops** setting mimics the default behavior of PWB 1.*x*.

Example The **Tabstops** switch setting

```
Tabstops:4
```

sets a tab stop every four columns.

Example The setting

```
Tabstops:3 4 7 8
```

sets a tab stop at columns 4, 8, 15, and every eight columns thereafter.

Example The setting

```
Tabstops:3 4 7 25 25 0
```

sets a tab stop at columns 4, 8, 15, 40, and 65. When the cursor is past column 65, the **Tab** function does nothing.

See **Backtab**, **Entab**, **Filetab**, **Realtabs**, **Tab**

Tilemode

Type Numeric

The **Tilemode** switch specifies the window tiling style. It can take one of the values below:

Value	Tiling Style
0	The first three windows are stacked one above the other.
1	The top two windows are tiled side-by-side.

When four or more windows are open, the tiling is the same in the two styles.

In stacked style (`Tilemode:0`), the top windows are placed one above the other, as shown in gray.

Figure 7.2 Vertical Tiling

In side-by-side style (`Tilemode:1`), the top two windows are tiled next to each other, as shown in Figure 7.3. This arrangement is good for comparing two files.

Figure 7.3 Horizontal Tiling

Default Tilemode:0

See **Arrangewindow**

Timersave

Type Numeric

The **Timersave** switch sets the interval in seconds between automatic file saves. The value must be in the range 0-65,535.

Set **Timersave** to 0 to turn off time-triggered autosave.

Default Timersave:0

See **Autosave**

Tmpsav

Type Numeric

The **Tmpsav** switch determines the maximum number of files kept in the file history between sessions.

When **Tmpsav** is 0, PWB lets the file history grow without limit; all files loaded into PWB appear in this list until you delete the CURRENT.STS file or change the value of the **Tmpsav** switch.

Default Tmpsav:20

Traildisp

Type Numeric

The **Traildisp** switch specifies the decimal ASCII code for the character used to display trailing spaces on a line. If you specify 0 or 255, PWB uses the space (ASCII 32) character.

Default Traildisp:0

See **Traillines**, **Trailspace**, **Traillinesdisp**

Traillines

Type Boolean

The **Traillines** switch determines if PWB preserves or removes empty trailing lines in a file when the file is written to disk.

You can make trailing lines visible by setting the **Traillinesdisp** switch to a value other than 0 or 32.

Syntax **Traillines**:{ **yes** | **no** }

yes Preserve trailing blank lines in the file.

no Remove trailing blank lines from the file.

Default Traillines:no

See **Traildisp**, **Trailspace**

Traillinesdisp

Type Numeric

The **Traillinesdisp** switch specifies the decimal ASCII code for the character displayed in the first column of blank lines at the end of the file. If you specify 0 or 255, PWB uses the space (ASCII 32) character.

Default Traillinesdisp:32

See **Traillines**, **Traildisp**, **Trailspace**

Trailspace

Type Boolean

The **Trailspace** switch determines if PWB preserves or removes trailing spaces from modified lines.

You can make trailing spaces visible by setting the **Traildisp** switch to a value other than 0 or 32.

Syntax **Trailspace**:{ **yes** | **no** }

yes Preserve trailing spaces on lines as they are changed.

no Remove trailing spaces from lines as they are changed.

Default Trailspace:no

See **Traillines**, **Traillinesdisp**

Undelcount

Type Numeric

The **Undelcount** switch determines the maximum number of backup copies of a given file saved by PWB.

This switch is used only when the **Backup** switch is set to `undel`.

Default Undelcount:32767

Undocount

Type Numeric

The **Undocount** switch sets the maximum number of edits per file that you can reverse with **Undo** (ALT+BKSP).

Default Undocount:30

Unixre

Type Boolean

The **Unixre** switch determines if PWB uses UNIX regular-expression syntax or PWB's non-UNIX regular-expression syntax for search-and-replace commands.

The **Unixre** and **Case** switches have no effect on the syntax of regular expressions used by the **Build** and **Word** switches. These switches always use case-sensitive UNIX regular-expression syntax.

Syntax **Unixre**:{ **yes** | **no** }

yes Use UNIX regular-expression syntax when searching.

no Use non-UNIX regular-expression syntax when searching.

Default Unixre:yes

User

Type Text

The **User** switch adds a custom menu item to the PWB Run menu.

Syntax **User**: *title*, *path*, [[*arg*]], [[*out*]], [[*dir*]], [[*help*]], [[*prompt*]], [[*ask*]], [[*back*]], [[*key*]]

If any argument to the **User** switch contains spaces, it must be enclosed in double quotation marks.

title
Menu title for the program to be added. No other command can have the same title. Prefix the character to be highlighted as the access key with a tilde (**~**) or ampersand (**&**). If you do not specify an access key, the first letter of the title is used.

path
Full path of the program. If the program is on the PATH environment variable, you can specify just the filename of the program.

arg
Command-line arguments for the program. To pass the name of the current file to the program, specify **%s** in the command line. Default: no arguments.

out
Name of a file to store program output. If no file is specified and the program is run in the foreground, the current file in PWB receives the output. Default: no output file.

dir
Current directory for the program. Default: PWB's current directory.

help
Text that appears on the status bar when the menu item is selected. Default: no help text.

prompt
Determines if PWB prompts for command-line arguments. The value of *arg* is the default response. Specify **Y** to prompt or any other character to run the program without prompting for arguments. Default: no prompt.

ask
Determines if PWB is to prompt for a keystroke before returning to PWB. Specify **Y** to prompt or any other character to return to PWB immediately after running the program. Default: return without prompting.

back
Determines if the program is run in the background under a multithreaded environment. Specify **Y** to run the program in the background or any other character to run it in the foreground. If you run the program in the background, you must also specify *output*. Default: run the program in the foreground.

key
A single digit from 1 to 9, specifying a key from ALT+F1 to ALT+F9 as the shortcut key for the command. Default: no shortcut key.

Default

By default, no custom menu commands are defined.

Example

The **User** switch setting

```
User : "~Print", XPRINT, "/2 %s", LPT1, ,  \
    "Print the current file with XPRINT", y, n, n, 8
```

specifies the following custom Run menu command:

Option	Description
title	The menu title is `Print` with the accelerator P.
path	The XPRINT program is expected to be on the PATH.
arg	The default command line specifies the /2 option and the current filename.
out	The program output is redirected to the LPT1 device.
dir	The XPRINT program is run in the current directory.
help	The Help line is `Print the current file with XPRINT`.
prompt	PWB prompts for additional arguments.
ask	PWB doesn't prompt before returning from XPRINT.
back	The XPRINT program is to run in the foreground.
key	ALT+F8 runs the XPRINT program after prompting.

The backslash at the end of the first line of the definition is a TOOLS.INI line continuation.

See

Printcmd, **_pwbuser***n*, **Usercmd**

Vscroll

Type Numeric

The **Vscroll** switch controls the number of lines scrolled up or down when you move the cursor out of the window. When the window is smaller than the full screen, the amount scrolled is in proportion to the window size.

The minimum value for **Vscroll** is 1. Text is never scrolled in increments greater than the size of the window.

The **Mlines** and **Plines** functions also scroll according to the value of the **Vscroll** switch.

Default Vscroll:1

See **Hscroll**

Width

Type Numeric

The **Width** switch controls the width of the display. Only an 80-column display is supported.

Default Width:80

See **Height**

Word

Type Text

Syntax **Word**: "*regular_expression*"

"*regular_expression*"
A macro string specifying a UNIX-syntax regular expression that matches a word.

The **Word** switch specifies a case-sensitive UNIX regular expression that matches a word. The **Unixre** and **Case** switches are ignored.

The **Word** switch accepts a TOOLS.INI macro string. The string can use escape sequences to represent nonprintable ASCII characters. Note that backslashes (\) must be doubled within a macro string.

The **Word** switch is used by functions that operate on words: **Mword**, **Pword**, **Pwbhelp**, right-clicking the mouse for Help, and double-clicking the mouse to select a word.

Default

Word:"[a-zA-Z0-9_$]+"

The default value mimics the behavior of PWB 1.*x*.

Examples

The **Word** switch can be used to change the definition of a word. The following examples show some useful word definitions.

The following setting works the same way as the default setting, except that **Pword** and **Mword** stop at the end of a line:

```
Word:"\\{[a-zA-Z0-9_$]+\\!$\\}"
```

The default setting of the **Word** switch matches Microsoft C/C++ identifiers and unsigned integers. To restrict the definition of a word to match the ANSI C standard for identifiers, you would use the setting:

```
Word:"[a-zA-Z_][a-zA-Z0-9_]*"
```

Another useful setting is to define a word as a contiguous stream of nonspace characters:

```
Word:"[^ \t]+"
```

The following **Word** setting defines a word as an identifier or unsigned integer, a stream of white space, a stream of other characters, or the beginning or end of the line. This causes the word-movement functions to stop at each boundary, and allows a double-click to select white space.

```
Word:"\\{[a-zA-Z0-9_$]+\\![ ]+\\![^a-zA-Z0-9_$]+\\!$\\!^\\}"
```

Wordwrap

Type Boolean

The **Wordwrap** switch determines if PWB performs automatic word wrap as you enter text.

When word wrapping is turned on and you type a nonspace character past the column specified by **Rmargin**, PWB brings the current word down to a new line. A word is defined by the **Word** switch.

Syntax **Wordwrap**:{ **yes** | **no** }

yes Wrap words as you enter text.

no Do not wrap.

Default Wordwrap:no

Update See **Rmargin**

Browser Switches

The PWBBROWSE extension provides the following switches to control the behavior of the Source Browser in PWB.

Browcase

Type Numeric

The **Browcase** switch determines the initial case sensitivity of the browser when a database is opened. The browser consults this switch only when it opens the database.

This switch must appear in the [PWB-PWBROWSE] tagged section of TOOLS.INI.

A dot appears next to the Match Case command on the Browse menu when the browser matches case. Choose Match Case to turn case-sensitive browsing on and off. Changing the current state does not affect the value of the **Browcase** switch.

Syntax **Browcase**:{ **0** | **1** | **2** }

0
Use the case sensitivity stored in the database by BSCMAKE. The default case sensitivity matches the case sensitivity of the source language.

1
Match case for browse queries.

2
Ignore case for browse queries.

Default Browcase:0

Browdbase

Type Text

The **Browdbase** switch specifies the browser database to use. When this switch is not set, or the setting is empty, the browser uses the database for the current project (if any). You set this switch by using the Save Current Database command in the Custom Database Management dialog box.

This switch must appear in the [PWB-PWBROWSE] tagged section of TOOLS.INI.

Syntax **Browdbase**: *database*

database
The full filename of the browser database (.BSC file) to use. When *database* is not specified, the browser uses the database for the open project.

Help Switches

The PWBHELP extension provides the following switches to control the behavior of the Help system in PWB.

Color (Help Colors)

The PWBHELP extension defines the following Color switches to set the colors for items displayed in the Help window. These switches must appear in the [PWB-PWBHELP] tagged section of TOOLS.INI. When you choose OK in the Save Colors dialog box, PWB automatically writes the new settings to the correctly tagged section of TOOLS.INI.

Name	Default Value	Description
Color: Helpnorm	87	Plain Help text
Color: Helpbold	8F	Bold Help text
Color: Helpitalic	8A	Italic Help text and the characters
Color: Helpunderline	8C	Emphasized Help text
Color: Helpwarning	70	Current hyperlink

For a complete description of the **Color** switch, see **Color**.

Helpautosize

Type

Boolean

The **Helpautosize** switch determines if PWB displays the Help window according to the size of the current topic or displays Help with its previous size and position.

This switch must appear in the [PWB-PWBHELP] tagged section of TOOLS.INI.

Syntax

Helpautosize: { **yes** | **no** }

yes
When displaying a new topic, automatically resize the Help window to the size of the topic.

no
Do not automatically resize the Help window. The Help window is displayed with its previous size and position.

Default

Helpautosize:no

Update

In PWB 1.*x*, the Help window is always automatically resized. In PWB 2.00, the Help window is not resized by default.

Helpfiles

Type

Text

The **Helpfiles** switch lists Help files or directories containing Help files that PWB should open in addition to the Help files listed in the HELPFILES environment variable.

This switch must appear in the [PWB-PWBHELP] tagged section of TOOLS.INI.

Syntax

Helpfiles: [[*file*]][[**;***file*]]...

file
The filename of a Help file to open or the name of a directory. If a directory name is used, all Help files in the directory are opened. Each *file* can contain wildcards or environment-variable specifiers.

Default

Helpfiles:

By default, PWB uses only the Help files in the current directory and those listed in the HELPFILES environment variable.

Helplist

Type

Boolean

The **Helplist** switch determines if PWB searches every Help file when you request Help or displays the first occurrence of the topic that it finds.

This switch must appear in the [PWB-PWBHELP] tagged section of TOOLS.INI.

Syntax

Helplist: { **yes** | **no** }

yes
Displays a list of Help files that contain the topic you requested Help on when the topic is defined more than once.

no
Does not display a list of topics. PWB displays the first Help associated with the requested topic. To see the other Help screens that define the topic, use the Next command on the Help menu.

Default

Helplist: yes

Helpwindow

(obsolete)

The PWB 1.*x* **Helpwindow** switch is obsolete and does not exist in PWB 2.00. PWB 2.00 always displays Help in the Help window.

P A R T 2

The CodeView Debugger

Chapter 8 Getting Started with CodeView 293
Chapter 9 The CodeView Environment 319
Chapter 10 Special Topics 351
Chapter 11 ..... Using Expressions in CodeView 375
Chapter 12 CodeView Reference 393

CHAPTER 8

Getting Started with CodeView

Microsoft CodeView is a window-oriented debugging tool that helps you find and correct errors in MASM and Microsoft C/C++ programs. With CodeView, you can examine source-level code and the corresponding compiled code at the same time. You can execute your code in increments and view and modify data in memory as your program runs.

Your MASM 6.10 package includes CodeView for MS-DOS (CV.EXE) and CodeView for Windows (CVW.EXE). The names "CodeView," "CodeView debugger," and "the debugger" refer to both versions unless the discussion indicates otherwise.

This chapter shows you how to:

- Write programs to make debugging easier.
- Formulate a debugging strategy.
- Compile and link your programs to include Microsoft Symbolic Debugging Information.
- Set up the files CodeView needs.
- Configure CodeView with TOOLS.INI.
- Start CodeView and load a program.
- Use the CodeView command-line options.
- Use or disable the CURRENT.STS state file.

Preparing Programs for Debugging

You can use CodeView to debug any MS-DOS or Windows-based executable file produced from MASM or Microsoft C/C++ source code. "Compiling" means producing object code from source files. All references to compiling also apply to assembling unless stated otherwise.

General Programming Considerations

This section describes programming practices that make debugging with CodeView easier and more efficient.

Multiple Statements on a Line

CodeView treats each source-code line as a unit. For this reason, you cannot trace and set a breakpoint on more than one statement per line. You can change from Source display mode to Mixed or Assembly display mode (see "The Source Windows" on page 324) and then set breakpoints at individual assembly instructions. If a single statement is broken across multiple lines, you may be able to set breakpoints on only the starting or ending line of the statement.

Macros and Inline Code

Microsoft C, C++, and MASM support macro expansion. Microsoft C and C++ also support inline code generation. These features pose a debugging problem because a macro or an inlined function is expanded where it is used, and CodeView has no information about the source code. This means that you cannot trace or set breakpoints in a macro or inlined function when debugging at the source level.

To work around this condition, you can:

- Manually expand the macro to its corresponding source code.
- Rewrite the macro as a function.
- Suppress inline code generation with the /Ob0 compiler option.

You can often rewrite macros as inline functions, then selectively disable inlining with a compiler option or pragma so that you can step and trace the routine. Rewriting macros as inlined functions can have additional benefits such as argument type checking. However, in some cases the best solution for debugging macros or inline code is to use Assembly or Mixed display mode.

Segment Ordering and Naming

For assembly-language programs, you must declare your segments according to the standard Microsoft high-level language format. MASM versions 5.10 and later provide directives to specify the standard segment order and naming.

Programs that Alter the Environment

Programs that run under CodeView can read the environment table, but they cannot permanently change it. When you exit CodeView, changes to the environment are lost.

Programs that Access the Program Segment Prefix

CodeView processes the command line in the program segment prefix (PSP) the same way as the C/C++ run-time library does. Quotation marks are removed, and exactly one space is left between command-line arguments. As a result, a program that accesses the PSP directly cannot expect the command line to appear exactly
as typed.

Compiling and Linking

After you compile and link your program into a running executable file, you can begin debugging with CodeView. To take full advantage of CodeView, however, you must compile and link with the options that generate CodeView Symbolic Debugging Information. This book refers to this information as “CodeView information,” “debugging information,” or “symbolic information.”

The CodeView information tells CodeView about:

- All program symbols, including locals, globals, and publics
- Data types
- Line numbers
- Segments
- Modules

Without this information, you cannot refer to any source-level names, and you can view the program only in Assembly display mode. When CodeView loads a module that does not contain symbolic information, CodeView starts in Assembly mode and displays the message:

```
CV0101 Warning: No symbolic information for PROGRAM.EXE
```

You get this message if you try to debug an executable file that you did not compile and link with CodeView options, if you use a compiler that does not generate CodeView information, or if you link your program with an old version of the linker. If you retain an old linker version and it is first in your path, the proper information may not be generated for CodeView.

You can specify CodeView compiler and linker options from the command line, in a makefile, or from within the Microsoft Programmer’s Workbench (PWB). To compile and link your program with CodeView options from PWB, choose Build Options from the Options menu, and turn on Use Debug Options. By default, all project templates enable the generation of CodeView information for debug builds.

Assembler/Compiler Options

You can specify CodeView options when you assemble a source file of a program you want to debug. Specify the /Zi option on the command line or in a makefile to instruct the assembler to include line-number and complete symbolic information in the object file.

Symbolic information takes up a large amount of space in the executable file and in memory while debugging. If you do not need full symbolic information in some modules, compile those modules with the /Zd option. The /Zd option specifies that only line numbers and public symbols are included in the object file. In such modules you can view the source file and examine and modify global variables, but type information and names with local scope are not available.

For modules that are assembled or compiled with the /Zd option, all names in that module are displayed and can only be referred to using their "decorated name." The decorated name is the form of the name in the object code produced by the compiler. With full debugging information, CodeView can translate between the source form of the name and the decorated name.

Name decoration encodes additional information into a symbol's name by adding prefixes and suffixes. For example, the C compiler prefixes the names of functions that use the C calling convention with an underscore. You often see decorated names for library routines in disassembly or output from the Examine Symbols (**X**) command. For more information on decorated names, see "Symbol Formats" on page 385.

All Microsoft high-level language compilers are optimizing compilers that may rearrange and remove source code. As a result, optimizations destroy the correspondence between source lines and generated machine code, which can make debugging especially difficult. While you are debugging, you should disable optimizations with the /Od compiler option. When you finish debugging, you can compile a final version of your program with full optimizations.

Note The /Od option does not pertain to MASM.

Linker Options

When you are using Microsoft C/C++ or the Microsoft Assembler, you must use the Microsoft Segmented Executable Linker (LINK) version 5.30 or later to generate an executable file with CodeView information. If you include debugging options when you compile, the compiler automatically invokes the linker with the appropriate options. In turn, LINK runs the CVPACK utility, which compresses the symbolic information.

When compiling, you can specify the compile-only (/c) option to disable running LINK. To include debugging information when you link the object modules separately, specify the LINK /CO option. LINK automatically runs CVPACK when you specify /CO.

If you link with the /EXEPACK option, you must execute the program's startup code before setting breakpoints in the program. If you set breakpoints in a packed executable file before the startup code has executed, CodeView behavior is unpredictable.

An executable file that includes debugging information can be executed from the command line like any other program. However, to minimize the size of the final version of the program, compile and link without the CodeView options.

Examples

The following command sequence assembles and links two files:

```
ML /C /Zi MOD1.ASM
ML /C /Zd MOD2.ASM
LINK /CO MOD1 MOD2
```

This example produces the object file MOD1.OBJ, which contains line-number and complete symbolic information, and the object file MOD2.OBJ, containing only line-number and public-symbol information. The object files are then linked to produce a smaller file than the file that is produced when both modules are assembled with the /Zi option.

The following commands produce a mixed-language executable file:

```
CL /Zi PROG.CPP
CL /Zi /Od /c /AL SUB1.C
ML /C /Zi /MX SUB2.ASM
LINK /CO PROG SUB1 SUB2
```

You can use CodeView to trace through C, C++, and MASM source files in the same session.

Debugging Strategies

The process of debugging a program varies from programmer to programmer and program to program. This section offers some guidelines for detecting bugs. If you are familiar with symbolic, source-level debuggers, you can skip this section.

Identifying the Bug

If your program crashes or yields incorrect output, it has a bug. There are times, however, when a program runs correctly with some input but produces incorrect

output or crashes with different input. You can assume a bug exists, but finding it may be difficult.

Locating the Bug

You may not need to use CodeView to find bugs in simple programs. For more complex programs, however, using CodeView can save you debugging time and effort.

Setting Breakpoints

When you debug with CodeView, you usually cycle between two activities:

- Running a small part of the program
- Stopping the program to check its status

You use breakpoints to switch between these tasks. CodeView runs your program until it reaches a breakpoint. At that time, CodeView gives you control. You can then enter CodeView commands in the Command window or use the menus and shortcut keys to proceed.

To find an error, try the following:

- Set breakpoints around the place you think the bug might be. Execute the program with the Go command so that it runs at full speed until it reaches the area that you suspect harbors the bug. You can then execute the program step by step with the Program Step and Trace commands to see if there is a program execution error.
- Set breakpoints when certain conditions become true. You can, for example, set a breakpoint to check a range of memory starting at DS:00, the base of your program's data. If your program writes to memory using a null pointer, the breakpoint is taken, and you can see what statement or variables within the statement are in error.

Setting Watch Expressions

Watch expressions constantly display the values of variables in the Watch window. By setting a Watch expression, you can see how a variable or an expression changes as your program executes.

Try using watch expressions as follows:

- Set a Watch expression on an important variable. Then step through a part of the program where you suspect there is a bug. When you see a variable in the Watch window take on an unexpected value, you know that there is probably a bug in the line you just executed.

- Explore Watch expressions. A bug can appear when your program builds complex data structures. Both the Watch window and the Quick Watch dialog box allow you to explore the data structure by expanding arrays and pointers. Use this feature to make sure the program creates the data structure correctly. As soon as you execute code that destroys the structure, you have probably found a bug.

Arranging Your Display

Your display can be more effective if you arrange your windows so that they display the information you need. You will need at least one Source window. You can open a second Source window to see each assembly-language instruction.

You may also need one or more Memory windows to examine ranges of memory in various formats. You may want to change values in memory. For example, a program that does its own dynamic-memory allocation may need an initialized block of memory. You can edit memory directly in the Memory window or fill the block with zeros using the Memory Fill command. If a certain value is required for a mathematical function, you can type over values displayed in the Memory window or assign the value in the Command window. If you expect a value to appear at a certain location and it does not, you can use the Memory Search command to find it.

Use the Register window to see the CPU registers and the Local and Watch windows to keep track of changing variable values. Open the Calls menu to examine your program's stack to see what routines have been called.

You can set up CodeView's windows to display the information you want to see by using keyboard commands or the commands in the Window menu. For example, when you press SHIFT+F5 or choose Tile from the Windows menu, CodeView arranges all open windows to fill the entire window area. When the windows are tiled, you can press ALT+F5 or choose Arrange from the Windows menu. This allows you to move your open windows with a mouse so that you can view several or all of them at once.

Setting up CodeView

The MASM SETUP program installs all the necessary CodeView files. Make sure that all of the CodeView executable files (.EXE and .DLL files) are in a directory listed in the PATH environment variable.

In addition, SETUP creates TOOLS.PRE in the INIT directory that you specify when you run SETUP. If you do not already have a TOOLS.INI file, rename TOOLS.PRE as TOOLS.INI.

This file contains the recommended settings to run CodeView for MS-DOS and CodeView for Windows. For more information on the entries in TOOLS.INI, see "Configuring CodeView with TOOLS.INI" on page 301.

CodeView version 4.0 introduces a new, flexible architecture for the debugger. CodeView is made up of a main executable program: CV.EXE (CodeView for MS-DOS) or CVW.EXE (CodeView for Windows) and a collection of dynamic-link libraries (DLLs). Each DLL implements an aspect of the debugging process.

The following table summarizes CodeView's component DLLs:

TOOLS.INI Entry	Component	Required	Example
Eval	Expression evaluator	Required	C or C++
Model	Additional nonnative execution model	Optional	P-code
Native	Native execution model	Required	MS-DOS or Windows
Symbolhandler	Symbol handler	Required	MS-DOS or Windows
Transport	Transport layer	Required	Local or remote

This architecture allows for the implementation of such improbable debugging configurations as a Windows operating system-hosted debugger that debugs interpreted Macintosh programs across a network. The existing CVW.EXE could be used with new transport, symbol handling, and execution model DLLs. Instead of creating completely different programs for each combination of host and target, all that is needed is the appropriate set of DLLs.

CodeView Files

CodeView for Windows and CodeView for MS-DOS use several additional files. One of these is the executable program file that you are debugging. CodeView requires one executable (.EXE) file to load for debugging.

program.EXE
: An .EXE-format program to debug. CodeView assumes the .EXE extension when you specify the program to load for debugging.

source.ext
: A program source file. Your program may consist of more than one source file. When CodeView needs to load a source file for a module at startup or when you step into a new module, it searches directories in the following order:

1. The "compiled directory." This is the source-file path specified when you invoke the compiler.
2. The directory where the program is located.

If CodeView cannot find the source file in one of these directories, it prompts you for a directory. You can enter a new directory or press ENTER to indicate that you do not want a source file to be loaded for the module. If you do not specify a source file, you can debug only in Assembly mode.

CV.HLP
ADVISOR.HLP
: Help files for CodeView and the Microsoft Advisor. These two files are the minimum set of files required to use Help during a CodeView session. They must be in a directory listed in the HELPFILES environment variable or in the **Helpfiles** entry of TOOLS.INI. Depending on what programming environment you work in, you may also want to use the various programming language and p-code help files.

TOOLS.INI
: Specifies paths for CodeView .DLL files and other files that CodeView uses. The MASM SETUP program creates the file TOOLS.PRE in the directory specified in your INIT environment variable.

  If CodeView cannot find the modules it needs in its own directory, it looks for entries in TOOLS.INI that specify paths for the modules it needs. You can include other settings for CodeView in TOOLS.INI.

TOOLHELP.DLL
: System support .DLL for CVW.

Remote debugging requires additional files and a different configuration. The files and configuration required for remote debugging are described in Chapter 10, “Special Topics.”

Configuring CodeView with TOOLS.INI

You can configure CodeView and other Microsoft tools including the Microsoft Programmer’s WorkBench (PWB) and NMAKE by specifying entries in the TOOLS.INI file. You must have separate sections in TOOLS.INI for each tool. TOOLS.INI sections begin with a “tag”—a line containing the base name of the executable file enclosed in brackets (**[ ]**). The tag must appear in column one. The CV and CVW section tags look like this:

```
[CV]
;        MS-DOS CodeView entries
. . .
[CVW]
;        Windows operating system CodeView entries
. . .
```

In the TOOLS.INI file, a line beginning with a semicolon (**;**) is a comment.

CodeView looks for certain entries following the tag. Each entry may be preceded by any number of spaces, but the entire entry must fit on one line. You may want to indent each entry for readability.

CodeView TOOLS.INI Entries

You may want to specify or change entries in TOOLS.INI to customize CodeView. Table 8.1 summarizes the TOOLS.INI entries.

Table 8.1 CodeView TOOLS.INI Entries

Entry	Description
Autostart	Commands to execute on startup
Color	Screen colors
Cvdllpath	Path to CodeView .DLL files
Eval	Expression evaluator
Helpbuffer	Size of help buffer
Helpfiles	List of help files
Model	Additional execution model (such as p-code)
Native	Native execution model
Printfile	Default name for print command or file
Statefileread	Read or ignore CURRENT.STS state file
Symbolhandler	Symbol handler
Transport	Transport layer

Autostart

The **Autostart** entry specifies a list of Command-window commands that CodeView executes on startup.

Syntax

Autostart:*command*[[**;***command*]]...

command
A command for CodeView to execute at startup. Separate multiple commands with a semicolon (**;**).

Example

The following entry automatically executes the program's run-time startup code. It specifies that CodeView always starts with the Screen Swap option off and the Trace Speed option set to fast.

```
Autostart:OF-;TF;Gmain
```

Color

The **Color** entry is retained only for compatibility with previous versions of CodeView. You should set screen colors with the Colors command on the Options menu.

Cvdllpath

The **Cvdllpath** entry specifies the default path for CodeView's dynamic-link libraries (DLLs). CodeView searches this path when it cannot find its DLLs in CodeView's directory or along the PATH environment variable. This entry is recommended.

Syntax

Cvdllpath:*path*

path
The path to the CodeView .DLL files.

Eval

The **Eval** entry specifies an expression evaluator. The expression evaluator looks up symbols, parses, and evaluates expressions that you enter as arguments to CodeView commands. If there is no **Eval** entry in TOOLS.INI, CodeView loads the C++ expression evaluator by default. CodeView uses the specified expression evaluator when you are debugging modules with source files ending in the specified extensions.

Syntax

Eval:[[*path*\]]**EE***host evaluator*.**DLL** *extension*...

path
The path to the specified expression evaluator.

host
The host environment.

Specifier	Operating Environment
D1	MS-DOS
W0	Windows

evaluator
The source language expression evaluator.

Specifier	Source Language
CAN	C or MASM
CXX	C, C++, or MASM

extension

A source-file extension. CodeView uses the specified expression evaluator when it loads a source file with the given extension. You can list any number of extensions.

Example

The following example loads both the C and C++ expression evaluators for the MS-DOS CodeView:

```
Eval:C:\C700\DLL\EED1CAN.DLL .C .ABC .ASM .H
Eval:C:\C700\DLL\EED1CXX.DLL .CPP .CXX .XYZ .HXX
```

With the entries in this example, when you trace into a module whose source file has the extension `.C`, `.ABC`, or `.ASM`, CodeView uses the C expression evaluator. When you trace into a source file with a `.CXX`, `.CPP`, or `.XYZ` extension, CodeView switches to the C++ expression evaluator.

Note The C++ expression evaluator is the only expression evaluator provided with MASM 6.10. For most MASM, C, and C++ programs the C++ expression evaluator is sufficient.

You can load expression evaluators after CodeView has started by using the Load command from the Run menu. You can override CodeView's automatic choice of expression evaluator by using the Language command on the Options menu or the **USE** command in the Command window.

For more information about choosing an appropriate expression evaluator and how to use expressions in CodeView, see Chapter 11, "Using Expressions in CodeView."

Helpbuffer

The **Helpbuffer** entry specifies the size of the buffer CodeView uses to decompress help files. You can set **Helpbuffer** to 0 to disable Help and maximize the amount of memory available for debugging. Otherwise, specify a value between 1 and 256.

Syntax

Helpbuffer:*size*

size

The number of kilobytes (K) of memory to use for decompressing help files. The default help buffer size is 24K. Specify 0 to disable help.

The following table shows values you can specify and the actual size of the buffer that is allocated:

Value Specified	Help Buffer Size
1–24	24K
25–128	128K
129–256	256K

The smallest buffer size is 24K, and the largest is 256K.

Helpfiles

The **Helpfiles** entry lists help files for CodeView to load. These files are loaded before any files listed in the HELPFILES environment variable.

Syntax

Helpfiles:*file*[[**;***file*]]...

file

A directory or help file. If you list a directory, CodeView loads all files with the .HLP extension in that directory. Separate multiple files or directories with a semicolon (**;**).

Model

The **Model** entry specifies an additional execution model that CodeView uses when you are debugging nonnative code such as p-code. The execution model handles tasks specific to the type of executable code that you are debugging.

Syntax

Model:[[*path*\]]**NM***host model*.**DLL**

path

The path to the specified file.

host

The host environment must be one of the following:

Specifier	Operating Environment
D1	MS-DOS
W0	Windows

model

A nonnative execution model. The p-code execution model (PCD) is required if you plan to debug p-code.

Example

```
Model:NMD1PCD.DLL
```

Native

The **Native** entry specifies the native execution model. This DLL handles tasks that are specific to the machine and operating system on which you are running (the host) and specific to the native code (the target).

Syntax

Native:[[*path*\]]**EM***host target*.**DLL**

path

The path to the specified native execution model.

host
The host environment must be one of the following:

Specifier	Operating Environment
D1	MS-DOS
W0	Windows

target
The target environment must be one of the following:

Specifier	Operating Environment
D1	MS-DOS
W0	Windows

Printfile

The **Printfile** entry lists the default device name or filename used by the Print command on the File menu. This can be a printer port (for example, LPT1 or COM2) or an output file. If **Printfile** is omitted, CodeView prints to a file named CODEVIEW.LST in the current directory. This entry is ignored by CVW, which does not have the Print command.

Syntax

Printfile:*path*

path
The path to the specified output file or the name of a device.

Statefileread

The **Statefileread** entry tells CodeView to read or ignore the CodeView state file (CURRENT.STS) on startup. You can toggle this setting from the command line using the /TSF (Toggle State File) option. These options have no effect on writing CURRENT.STS. CodeView always saves its state on exit.

Syntax

Statefileread:[**y** | **n**]

y (yes)
CodeView reads CURRENT.STS on startup.

n (no)
CodeView ignores CURRENT.STS on startup.

Symbolhandler

The **Symbolhandler** entry specifies a symbol handler. The symbol handler manages the CodeView symbol and type information.

Syntax **Symbolhandler**:[[*path*\]]**SH***host*.**DLL**

path
The path to the symbol handler.

host
The host environment must be one of the following:

Specifier	Operating Environment
D1	MS-DOS
W0	Windows

Transport

The **Transport** entry specifies a transport layer. A transport layer provides the data link for communication between the host and target during debugging.

Syntax **Transport**:*path***TL***host transport*.**DLL** [[COM{1|2}:[[*rate*]]]]

path
The path to the specified transport layer.

host
The host environment must be one of the following:

Specifier	Operating Environment
D1	MS-DOS
W0	Windows

transport
Specifies a transport layer.

Specifier	Transport Layer
LOC	Local transport layer
COM	Serial remote transport layer

The optional [[COM{1|2}:[[*rate*]]]] specifies a communications port and baud rate for remote debugging. No space is allowed between COM and the port number (1 or 2). The default port is COM1. The <rate> can be any number from 50 through 9600. The default rate is 9600.

You specify the local transport layer (LOC) when the debugger and the program you are debugging are running on the same machine. With the appropriate transport layer, CodeView can support remote debugging across serial lines or networks. For more information on remote debugging, see Chapter 10.

The following example specifies the transport layer for debugging a program that is running on the same machine.

Example

```
Transport:C:\C700\DLL\TLW0LOC.DLL
```

Memory Management and CodeView

CodeView for MS-DOS (CV) requires at least 2 megabytes of memory. The memory must be managed by a Virtual Control Program Interface (VCPI) server, DOS Protected-Mode Interface (DPMI) server, or extended memory (XMS) manager. These drivers manage memory at addresses above 1 megabyte on an 80286, 80386, or 80486 machine. CodeView loads itself and the debugging information for the program into high memory. In this way, CodeView uses only approximately 17K of conventional MS-DOS memory.

CodeView can use the following memory managers:

- A VCPI server such as EMM386.EXE or EMM386.SYS. With a VCPI server, your program is also able to use EMS memory. To use this memory manager you must have a command in your CONFIG.SYS file such as:

  ```
  DEVICE=C:\DOS\EMM386.EXE ram
  ```

- A DPMI server such as 386max.
- An Extended Memory Standard (XMS) driver such as HIMEM.SYS. To use this memory manager you must have a command in your CONFIG.SYS file such as:

  ```
  DEVICE=C:\DOS\HIMEM.SYS
  ```

For more information about using memory managers, see your memory manager's documentation. When you make new entries in your CONFIG.SYS file, remember to reboot your system so that your changes take effect.

The CodeView Command Line

You can specify CV or CVW options when you start them from the command line. You can also specify commands from within the CodeView environment to modify these startup arguments.

Syntax

CV[[**W**]] [[*options*]] [[*program* [[*arguments*]]]]
CV[[**W**]] *@file* [[*program* [[*arguments*]]]]

W
Indicates the Windows operating system version of CodeView.

options
One or more options. The CodeView options are described in the "Command-Line Options" section on page 310.

program
Program to be debugged. Specifies the name of an executable file to be loaded by the debugger. If you specify *program* as a filename with no extension, CodeView searches for a file with the extension .EXE. If you do not specify a program, CodeView starts up and displays the Load dialog box where you can specify a program and its command-line arguments.

arguments
The program's command-line arguments. All remaining text on the CodeView command line is passed to the program you are debugging as its command line. If the program you are debugging does not accept command-line arguments, you do not need to specify any. Once you've started debugging, you can change the program's command-line arguments.

@file
File of command-line arguments. You can also specify arguments in a text file. The file contains a list of arguments, one per line. An argument file lets you specify a large number of arguments without exceeding the operating-system limit on the length of a command line. This is especially useful when starting a session that uses many DLLs.

After CodeView loads its DLLs, processes the debugging information, and loads the source file, the CodeView display appears. If you do not specify a program to debug or CodeView cannot find all of its required DLLs, CodeView prompts for the necessary files.

After starting up, CodeView is at the beginning of the program startup code, and you are ready to start debugging. At this point, you can enter an execution command (such as Trace or Program Step) to execute through the startup code to the beginning of your program.

Leaving CodeView

To exit CodeView at any time, choose the Exit command from the File menu. You can also press ALT+F4, or type `Q` (for "Quit") in the Command window.

At this point, you may want to skip ahead to the next chapter, "The CodeView Environment" for information on CodeView's menus and windows. The rest of this chapter describes each command-line option in detail, then continues with a description of how PWB and CodeView use the CURRENT.STS file.

Command-Line Options

CV and CVW accept some of the same options for debugging. Table 8.2 summarizes the CodeView command-line options.

Table 8.2 CodeView Command-Line Options

Option	CV	CVW	Description
/2	Yes	Yes	Use two displays
/8	No	Yes	Use 8514 and VGA displays
/25, /43, /50	Yes	Yes	Set 25-line, 43-line, or 50-line mode
/B	Yes	Yes	Use black-and-white display
/C*commands*	Yes	Yes	Execute commands
/F	Yes	No	Flip video pages
/G	Yes	Yes	Control snow on CGA displays
/I[0 | 1]	Yes	Yes	Trap NMIs and 8259 interrupts
/L*dll*	No	Yes	Load DLL or application symbols
/M	Yes	Yes	Disable mouse
/N[0 | 1]	Yes	Yes	Trap nonmaskable interrupts
/S	Yes	No	Swap video buffers
/TSF	Yes	Yes	Read or ignore state file
/X	No	Yes	Set starting X coordinate (pixels)
/Y	No	Yex	Set starting Y coordinate (pixels)

The remainder of this section describes each option in detail.

Use Two Displays (CV, CVW)

Option **/2**

The /2 option permits the use of two monitors. The program display appears on the default monitor, while CodeView displays on the secondary monitor. You must have two monitors and two adapters to use the /2 option. The secondary display must be a monochrome adapter.

If you are debugging a Windows-based application and have an IBM PS/2 with an 8514 primary display and a Video Graphics Adapter (VGA) secondary display, use the /8 option.

Use 8514 and VGA Displays (CVW)

Option **/8**

If your system is an IBM PS/2, you can configure it with an 8514 as the primary display and a VGA as the secondary display. To use this configuration, specify the /8 (8514) option on the CVW command line.

If your VGA monitor is monochrome, it is recommended to use the /B (black-and-white) option. The 8514 serves as the Windows operating system screen and the VGA as the debugging screen.

By default, the debugging screen operates in 50-line mode in this configuration. If you specify the /8 option, you can specify /25 or /43 for 25-line or 43-line mode on the debugging screen.

Warning Results are unpredictable if you attempt to run non-Windows-based applications or the MS-DOS shell while you are running CVW with the /8 option.

Set Line-Display Mode (CV, CVW)

Options **/25**
/43
/50

If you have the appropriate display adapter and monitor, you can display 25, 43, or 50 lines when you are running CV, and 25 or 50 lines when you are running CVW. The mode you specify is saved in the CURRENT.STS file so that it is still in effect the next time you run CodeView.

CVW.EXE supports 25, 43 and 50 lines on VGA monitors. It does not support 50-line mode on EGA monitors. If you specify a mode that is not supported by your adapter and your monitor, CodeView displays 25 lines.

To display 43 or 50 lines on a screen, you must use the OEM fonts supplied with CodeView. There are two OEM files: OEM08.FON for 50-line mode, and OEM10.FON for 43-line mode. To use these fonts, change the OEMFONTS.FON entry in your SYSTEM.INI file. For example, to use 50-line mode, change:

```
OEMFONTS.FON=VGAOEM.FON
```

to:

```
OEMFONTS.FON=C:\MASM\BIN\OEM08.FON
```

Use Black-and-White Display (CV, CVW)

Option **/B**

When you start CodeView, it checks the kind of display adapter that is installed in your computer. If the debugger detects a monochrome adapter, it displays in black and white; if it finds a color adapter, it displays in color. The /B option tells CodeView to display in black and white even if it detects a color adapter.

If you use a monochrome display or laptop computer that simulates a color display, you many want to disable color. These displays may be difficult to read with CodeView's color display.

You can also customize CodeView's colors by choosing the Colors command from the Options menu. For more information, see "Colors" on page 345.

Execute Commands (CV, CVW)

Option **/C***commands*

You type commands in the CodeView Command window. You can also specify Command-window commands when you start CodeView. The /C option allows you to specify one or more CodeView Command-window commands to be executed upon startup. If you specify more than one command, you must separate each one with a semicolon (**;**).

If the commands contain spaces or redirection symbols (< or >), enclose the entire option in double quotation marks (**"**). Otherwise, the debugger interprets each argument as a separate CodeView command-line argument rather than as a Command-window command.

For complete information on CodeView Command-window commands, see Chapter 12, "CodeView Reference."

Examples The following example loads CV with `CALCPR` as the executable file and `/p TST.DAT` as the program's command line:

```
CV /CGmain CALCPR /p TST.DAT
```

Upon startup, CV executes the high-level language startup code with the command `Gmain`. Since no space is required between the command (`G`) and its argument (`main`), there is no need to enclose the option in double quotation marks.

The next example loads CV with `CALCPR` as the executable file and `/p TST.DAT` as the program's command line. It starts CodeView with a long list of startup commands.

```
CV "/C VS &;G signal_lpd;MDA print_buffer L 20" CALCPR /p TST.DAT
```

CodeView starts with the Source window displaying in Mixed mode (`VS &`). Then it executes up to the function `signal_lpd` with the command `G signal_lpd`. Next, it dumps 20 characters starting at the address of `print_buffer` with the command `MDA print_buffer L 20`. Since some of the commands use spaces, the entire /C option is enclosed in quotation marks.

In this example, the command directs CV to take Command-window input from the file SCRIPT.TXT rather than from the keyboard:

```
CV "/C<SCRIPT.TXT" CALCPR TST.DAT
```

Although the option does not include any spaces, you must enclose it in quotation marks so that the less-than symbol (<) is read by CodeView rather than by the operating-system command processor.

Set Screen-Exchange Method (CV)

Options

/F
/S

CodeView allows you to move between the output screen, which contains your program display output, and the CodeView screen, which contains the debugging display. In MS-DOS, CodeView can perform this screen exchange in two ways: screen flipping or screen swapping. The /F (flipping) and /S (swapping) options allow you to choose the method from the command line. These two methods are:

Flipping
: Flipping is the default for a computer with a graphics adapter. CodeView uses the graphic adapter's video-display pages to store each screen of text. Flipping is faster than swapping and uses less memory, but it cannot be used with a monochrome adapter or to debug programs that use graphic video modes or the video-display pages. CodeView ignores the /F option if you have a monochrome adapter.

Swapping
: Swapping is the default for computers with monochrome adapters. It has none of the limitations of flipping, but it is slower than flipping and requires more memory. To swap screens, CodeView creates a buffer in memory and uses it
to store the screen that is not displayed. When you request the other screen, CodeView swaps the screen in the display buffer for the one in the storage buffer. When you use screen swapping, the buffer is 16K bytes for all adapters. The amount of memory CodeView uses is increased by the size of the buffer.

Suppress Snow (CV, CVW)

Option **/G**

The /G option suppresses snow that can appear on some CGA displays. Use this option if your CodeView display is unreadable because of snow.

Specify Interrupt Trapping (CV, CVW)

Options **/I**[[0 | 1]]
/N[[0 | 1]]

The /I option tells CV whether to handle nonmaskable-interrupt (NMI) and 8259-interrupt trapping. The /N option controls only CodeView's handling of NMIs and does not affect handling of interrupts generated by the 8259 chip. The following table summarizes the options and their effects:

Option	Effect
/I0	Trap NMIs and 8259 interrupts
/I1, /I	Do not trap NMIs or 8259 interrupts
/N0	Trap NMIs
/N1, /N	Do not trap NMIs

You may need to force CodeView to trap interrupts with /I0 on computers that CodeView does not recognize as IBM compatible. Using /I0 enables the CTRL+C and CTRL+BREAK interrupts on such computers.

Load Other Files (CVW)

Option **/L***dll*
/L*exe*

To load symbolic information from a dynamic-link library (DLL) or from another application, use the /L option when you start CodeView. Specify /L for each DLL or application that you want to debug.

When you place a module in a DLL, neither code nor debugging information for that module is stored in an application executable (.EXE) file. Instead, the code and symbols are stored in the library and are not linked to the main program until run time. The same is true for symbols in another application running within Windows. Thus, CVW needs to search the DLL or other application for symbolic information. Because the debugger does not automatically know which libraries to look for, use the /L option to preload the symbolic information.

Example

The following command starts CodeView for Windows:

```
CVW /LPRIORITY.DLL /LCAPPARSE.DLL PRINTSYS
```

CVW is used to debug the program PRINTSYS.EXE. CVW loads symbolic information for the dynamic-link libraries PRIORITY.DLL and CAPPARSE.DLL, as well as the file PRINTSYS.EXE.

Disable Mouse (CV, CVW)

Option **/M**

If you have a mouse installed on your system, you can tell CodeView to ignore it by using the /M option. You may need to use this option if you are debugging a program that uses the mouse and there is a usage conflict between the program and CodeView.

Nonmaskable-Interrupt Trapping (CV, CVW)

Option **/N**

For information on the /N option, see “Specify Interrupt Trapping” on page 314.

Set Screen Swapping (CV)

Option **/S**

The /S option sets the CodeView screen-exchange method to swapping. For complete information on CodeView screen-exchange methods, see “Set Screen-Exchange Method” on page 313.

Toggle State-File Reading

Option **/TSF**

The Toggle State File (/TSF) option either reads or ignores CodeView’s state file and color files, depending on the **Statefileread** entry in the CodeView sections of TOOLS.INI. The /TSF option reverses the effect of the **Statefileread** entry. The **Statefileread** entry is set to yes by default.

These options have no effect on writing the files. CodeView always saves its state on exit.

The effect of different combinations of **Statefileread** and /TSF are summarized in the following table:

/TSF	Statefileread	CodeView Result
Specified	y (or omitted)	Do not read files
Specified	n	Read files
Not specified	y (or omitted)	Read files
Not specified	n	Do not read files

The state file is CURRENT.STS. The color files are CLRFILE.CV4 for CV and CLRFILE.CVW for CVW.

Set Startup X and Y Coordinates

Options **/X, /Y**

The window CodeView uses within Windows cannot be moved or sized while Windows is running. You can specify the position of the CodeView window with the /X and /Y options. In the Command Line field of the Program Item Properties dialog box, enter

```
C:\MASM\BIN\CVW.EXE /X:X /Y:Y
```

where X and Y are the pixel coordinates for the upper lefthand corner of the CodeView session window. (The location for your CVW.EXE file may be different.) Note that this still does not allow the CodeView window to be moved to another location on the Program Manager workspace. For more information on specifying command-line options with Windows operating system applications, see your Windows *User's Guide*.

The CURRENT.STS State File

CodeView and PWB save settings and state information in the CURRENT.STS file. The file contains information about the current state of the two environments. When you restart CodeView or PWB, they read CURRENT.STS and restore their previous state. CodeView uses additional files to save your most recent color settings. These files are CLRFILE.CV4 for CV and CLRFILE.CVW for CVW.

CodeView and PWB search for these files in the directory that the INIT environment variable specifies. If no INIT environment variable exists, CodeView and PWB search the current directory. If no state file is found, new CURRENT.STS and CLRFILE.CV4 or CLRFILE.CVW files are created in the INIT directory or the current directory if no INIT variable is set.

Information about CodeView stored in CURRENT.STS includes:

- Window layout
- Breakpoints
- Watch expressions
- Source, Local, and Memory display options
- Global CodeView options such as case sensitivity, screen exchange method, radix, and expression evaluator

You can set CodeView options in TOOLS.INI or on the command line and then modify them during a session. They are saved in CURRENT.STS when you exit CodeView. During each CodeView session, these features are set in the following order:

1. From TOOLS.INI
2. From the CodeView command line
3. From CURRENT.STS
4. During the debugging session

The following items are not saved between sessions:

- The current location (CS:IP).
- The expansion state of watch expressions.

  All watch expressions and their format specifiers are restored, but they appear in their contracted state.
- Absolute-address breakpoints.

  Breakpoints set at an absolute *segment*:*offset* address are not saved. CodeView saves breakpoints only at specific line numbers or symbols.
- Memory window addresses.

  Each memory window is restored with its display type and options, but CodeView does not save the starting address. Instead, Memory windows show the start of the data segment (address DS:00).

CHAPTER 9

The CodeView Environment

CodeView provides a powerful environment in which to debug programs and dynamic-link libraries (DLLs). Its rich set of commands helps you track program execution and changing data values.

In CodeView you can “point-and-click” your source code to start and stop execution or modify bytes in memory. You can also use more traditional keyboard commands. You can use function keys to execute common commands, such as tracing and stepping through a program. When you quit CodeView, it remembers your breakpoints, window arrangement, watch expressions, and option settings.

This chapter describes the CodeView display, shows you how to use the menu commands, and how to interact with the different types of windows.

The CodeView Display

The CodeView screen is divided into three parts:

- The menu bar across the top of the screen
- The window area between the menu bar and status bar
- The removable status bar across the bottom of the screen

Figure 9.1 shows a typical CodeView screen with several open windows. The figure shows selected elements of the display, which are described in the sections that follow.

Figure 9.1 CodeView Display

The Menu Bar

The menu bar displays the names of the CodeView menus. To open a menu, use one of the following methods:

- Click a menu title with the mouse.
- Press ALT plus the menu title's highlighted letter.
- Press and release ALT, use the arrow keys to select a menu, and then press DOWN ARROW or ENTER to open it.

Each command in a menu has a highlighted letter. To choose that command, press the highlighted letter. Many commands also list a shortcut key that you can press at any time instead of opening a menu and choosing a command.

A command that does not apply to a particular situation is dimmed on the menu. When you press the corresponding shortcut key, no action is performed.

The Window Area

Most of your debugging takes place in the window area, where you can open, close, move, size, and overlap the various types of CodeView windows.

Although each window serves a different function for debugging, the windows have a number of common features. The Close, Maximize, Restore, and Minimize boxes work in the same way as they do in PWB. The scroll bars also work the same as in PWB. For information on the window border controls, see Chapter 4, "User Interface Details."

Only one window can be active at a time. You always use the currently active window, which appears with a highlighted border and a shadow on the screen. The text cursor always appears in the active window.

The Status Bar

The status bar contains information about the active window. It usually includes a row of buttons you can click to execute commands. You can also use the shortcut keys shown on the buttons.

To remove the status bar and gain an extra line for the window area, choose Status Bar from the Options menu, or type the OA- command in the Command window. To restore the status bar, choose Status Bar from the Options menu, or type the OA+ command in the Command window. For more information on this command, see the "Options" command on page 422.

CodeView Windows

CodeView windows organize and display information about your program. This section describes each CodeView window, the information you can display, and how you can change information and enter commands in the Command window. It also explains how to move among the windows and manipulate them.

How to Use CodeView Windows

Each CodeView window has a different function and operates independently of the others. Only one window can be active at a time. Commands you choose from the menus or by using shortcut keys affect the active window. The following list briefly describes each window's function:

Source
: Displays the source or assembly code for the program you are debugging. You can open a second Source window to view an include file or any ASCII text file.

Command
: Accepts debugging commands from the keyboard. CodeView displays the results, including error messages, in the Command window. When you enter a command in a dialog box, CodeView displays any resulting errors in a pop-up window.

Watch
: Displays the values of variables and expressions you select. You can modify the value of watched variables, browse the contents of structures and arrays, and follow pointers through memory.

Local
: Lists the values of all variables local to the current scope. You can set Local window options to show other scopes. You can modify the values of variables displayed in the Local window.

Memory
: Displays the contents of memory. You can open a second Memory window to view a different section of memory. You can set Memory window options to select the format and address of displayed memory. You can directly change the displayed memory by typing in the Memory window.

Register
: Displays the contents of the machine's registers and flags. You can directly edit the values in the registers, and you can toggle flags with a single keystroke or mouse click.

8087
: Displays the registers of the hardware math coprocessor or the software emulator.

Help
: Displays the Microsoft Advisor Help system.

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. The Local window is empty until you trace into the main part of the program.

You can open or close any CodeView window. However, at least one Source window must remain open. When you exit CodeView, it records which windows are open and how they are positioned, along with their display options. These settings become the default the next time you run CodeView.

To open a window, choose a window from the Windows menu. Some operations, such as setting a watch expression or requesting help, open the appropriate window automatically.

You can change how CodeView displays information in the Source, Memory, and Local windows. Choose the appropriate window options command from the

Options menu. When the cursor is in one of these windows, you can press CTRL+O to display that window's options dialog box.

CodeView automatically updates the windows as you debug your program. To interact with a particular window (such as entering a command or modifying a variable), you must select it. The selected window is the "active" window. The active window is marked in the following ways:

- The window's frame is highlighted.
- The window casts a shadow over other windows.
- The cursor appears in the window.
- The horizontal and vertical scroll bars move to the window.

To make a window active, click anywhere in the window or in the window frame. You can also press F6 or SHIFT+F6 to cycle through the open windows, making each one active in turn. You can also choose a window from the Windows menu or press ALT plus a window number. In addition, some CodeView commands make a certain window active.

Moving Around in CodeView Windows

To move the cursor to a specific window location, click that location. You can also use the keyboard to move the cursor as shown in Table 9.1.

Table 9.1 Moving Around with the Keyboard

Action	Keyboard
Move cursor up, down, left, and right	UP ARROW, DOWN ARROW, LEFT ARROW, RIGHT ARROW
Move cursor left and right by words	CTRL+LEFT, CTRL+RIGHT
Move cursor to beginning of line	HOME
Move cursor to end of line	END
Page up and down	PAGE UP, PAGE DOWN
Page left and right	CTRL+PAGE UP, CTRL+PAGE DOWN
Move cursor to beginning of window	CTRL+HOME
Move cursor to end of window	CTRL+END
Move to next window	F6
Move to previous window	SHIFT+F6
Restore window	CTRL+F5
Move window	CTRL+F7
Size window	CTRL+F8
Minimize window	CTRL+F9

Maximize window	CTRL+F10
Close window	CTRL+F4
Tile windows	SHIFT+F5
Arrange windows	ALT+F5

The Source Windows

The Source windows display the source code. You can open a second Source window to view other source files, header files, the same source file at a different location, or any ASCII text file. To open a Source window, use one of the following methods:

- From the Windows menu, choose Source 1 or Source 2.
- In the Command window, type the View Source (**VS**) command.
- Press ALT+3 to open Source window 1.
- Press ALT+4 to open Source window 2.

You cannot edit source code in CodeView, but you can temporarily modify the machine code in memory using the Assemble (**A**) command. For more information on the Assemble command, see page 400.

Source windows can display three different views of your program code in three different modes:

- Source mode shows your source file with numbered lines.
- Assembly mode shows a disassembly of your program's machine code.
- Mixed mode shows each numbered source line followed by a disassembly of the machine code for each line.

Note When you are debugging p-code while Native mode is off, CodeView displays p-code instructions rather than disassembled machine instructions. See the "Options" command on page 422. For more information on p-code, see "Debugging P-code" on page 363.

CodeView automatically switches to Assembly mode when you trace into routines for which no source is available, such as library or system code. The debugger switches back to the original display mode when you continue tracing into code for which source code is available.

For more information on setting display modes, see the "View Source" command on page 433. For detailed information about the Source window display options, see page 343.

The Watch Window

The Watch window displays the value of program variables or the value of expressions you specify in a high-level language. For each expression or variable, you can change the format of the data that is displayed. You can expand aggregate variables, such as structures and arrays, to show all the elements of an aggregate and contract them to save space in the Watch window. You can follow chains of pointers to display and help debug more complex structures, such as linked lists or binary trees.

To open a Watch window, use one of the following methods:

- From the Windows menu, choose Watch.
- In the Command window, type the Add Watch (**W?**) command followed by the variable or expression name.
- Press ALT+2.

To add expressions to the Watch window, use the Add Watch command from the Data menu or the Quick Watch dialog box (SHIFT+F9). You can also add watch expressions using the Add Watch (**W?**) and Quick Watch (**??**) commands.

Note Do not edit a string in the Watch window.

To change the value of any variable displayed in the Watch window, move the cursor to the value, delete the old value, and type the new value. To change the format in which a variable is displayed or to specify a new format, move the cursor to the end of the variable name and type a new format specifier.

To toggle between insert and overtype modes, press the INS key.

Using the Watch Window to View Multi-Level Arrays

You can use the watch window to view the changing values of a structure or array as you step or trace through your program:

1. Open the Watch Window.
2. Add the structure whose elements you want to track to the Watch window with the Add Watch command from the Data menu, or by using the Quick Watch dialog box (SHIFT+F9). The structure name will be added to the Watch Window.
3. Using the mouse, double-click anywhere on the structure name in the Watch Window to expand it one level. Double-click again on any subsequent levels until the structure is open to the level you want to watch.

4. Step or Trace through the code using F8 or F10 keys. The structure elements will update with each step.

For information on expanding and contracting aggregate types and following pointers, see the "Quick Watch" command on page 453. For detailed information on specifying and using watch expressions, see the "Codeview Expression Reference" on page 393 and Chapter 11, "Using Expressions in CodeView."

The Command Window

You type CodeView commands in the Command window to execute code, set breakpoints, and perform other debugging tasks. You can use the menus, mouse, and keyboard for many debugging tasks, but you can use some CodeView commands only in the Command window.

When you first start the debugger, the Command window is active, and the cursor is at the CodeView prompt (>). To return to the Command window after you make another window active, click the command window, or press ALT+9.

Using the Command window is similar to using an operating-system prompt, except that you can scroll back to view previous results and edit or reuse previous commands or parts of commands.

How to Enter Commands and Arguments

You enter commands in the Command window at the CodeView prompt when the Command window is active. Type the command followed by any arguments and press ENTER. Some commands, such as the Assemble (**A**) command, prompt for an indefinite series of arguments until you enter an empty response. CodeView may display errors, warnings, or other messages in response to commands you enter in the Command window.

If a Source window is active and the Command window is open, you can still type Command-window commands. When you begin typing, the cursor moves to the Command window and remains there until you press ENTER. The cursor returns to the Source window, and CodeView executes the command. If you have begun typing but do not want to execute a command, press ESC to clear the text and place the cursor at the prompt. After you press ESC, the Command window becomes active.

Command Format

The format for CodeView commands is as follows:

command [[*arguments*]] [[;*command2*]]

The *command* is the command name, and *arguments* are control options or expressions that represent values or addresses to be used by the command. The first argument can usually be placed immediately after *command* with no intervening spaces. Arguments may be separated by spaces or commas, depending on the command. For more information, see Chapter 12, "CodeView Reference."

To specify additional commands on the same line, separate each command with a semicolon (**;**).

Commands are always one, two, or three characters long. They are not case sen-
sitive, so you can use any combination of uppercase and lowercase letters. Arguments to commands may be case sensitive, depending on the command.

Example

The following example shows three commands separated by semicolons:

```
MDB 100 L 10 ; G .178 ; MDB 100 L 10
```

The first command (`MDB 100 L 10`) dumps 10 bytes of memory starting at address 100. The second command (`G .178`) executes the program up to line 178 in the current module. The third command is the same as the first and is used to see if the executed code changed memory.

Example

This example demonstrates the Comment (*****) command:

```
U extract_velocity    ;* Unassemble at this routine
```

The first command is the Unassemble (**U**) command, given the argument `extract_velocity`. The next command is the Comment command. Comment commands are used throughout the CodeView examples in this book.

How to Copy Text for Use with Commands

Copy and paste text instead of retyping.

Text that appears in any CodeView window can be copied and used in a command. For example, an address that is displayed in a Memory window or the Register window can be copied and used in a breakpoint command.

To copy and use text:

1. Select the text with the mouse or the keyboard.

   To select text with the mouse, move the mouse pointer to the beginning of the desired text, hold down the left mouse button, and drag the mouse. When you have selected the desired text, release the button.

To select text with the keyboard, move the cursor to the desired text, hold down the SHIFT key, and move the cursor with the ARROW keys.

2. Choose the Copy command from the Edit menu or press CTRL+INS.
3. Move the cursor to the location where you want to use the text and choose the Paste command from the Edit menu, or press SHIFT+INS.
4. Edit the command if desired, and press ENTER to execute the command.

Because all input to CodeView windows is line oriented, you cannot copy more than a single line. If you select more than a single line, the Copy command in the Edit menu is unavailable, and CTRL+INS has no effect. However, you can still select more than one line for use with the Print command on the File menu. For more information about the Print command, see "Print" on page 333.

When editing a command, you can toggle between insert and overtype modes by pressing the INS key.

How to Use the Command Buffer

CodeView keeps the last several screens of commands and output in the Command window. You can scroll the Command window to view the commands you entered earlier in the session. This is particularly useful for viewing the output from commands, such as Memory Dump (**MD**) or Examine Symbols (**X**), whose output exceeds the size of the window.

The TAB key provides a convenient way to move among the previously entered commands. Press TAB to move the cursor to the beginning of the next command, and press SHIFT+TAB to move to the beginning of the previous command. If the cursor is at the beginning or the end of the command buffer, the cursor wraps around to the other end. To return to the current command prompt, you can press CTRL+END or press TAB repeatedly.

You can also reuse any command that appears in the Command window without copying and pasting. Move the cursor to the command or press TAB, edit the command if desired, and press ENTER to execute it. When you press ENTER, CodeView restores the original command, copies the new command to the current prompt, and executes the command. If you make a mistake while editing a command, press ESC to restore the line.

The Local Window

The Local window shows all local variables in the current scope. The Local window is similar to the Watch window, except that the variables that are displayed change as the local scope changes. A variable in the Local window is always shown in its default type format. When you edit in the Local window, you can toggle between insert and overtype modes by pressing the INS key.

You can expand and contract pointers, structures, and arrays the same way you do in the Watch window. You can also change the values of the variables as in the Watch window.

The keyboard shortcut to open or switch to the Local window is ALT+1.

You can see the local variables of each active routine in the stack by selecting the routine from the Calls menu. For more information on this feature, see "The Calls Menu" on page 346.

By default, the Local window shows the addresses of the local variables on the left side of the window. You can turn this address display on or off using the Options (**O**) command. For more information on the Options command, see page 422.

The Register Window

The Register window displays the names and current values of the native CPU registers and flags. When you are debugging p-code, it displays names and values of the p-code registers and flags. You can change the value of any register or flag directly in the Register window.

To open the Register window, choose Register from the Windows menu, press ALT+7, or F2. You can also view and modify registers by using the Register (**R**) command. For more information about the Register command, see page 426.

When a register value changes after a program step or trace, CodeView highlights the new value so you can see how your program uses the CPU registers. Depending on the current instruction, the Register window also displays the effective address at the bottom of the window. This display shows the location of an operand in physical memory and its value.

If you are debugging on an 80386 or 80486 machine, you can view and modify the 32-bit registers. To turn on the 32-bit Registers option, choose the 386 command from the Options menu or use the O3+ command. The 32-bit registers are available if you are debugging on an 80386 or 80486 machine.

When you are debugging p-code, CodeView displays the p-code registers: DS, SS, CS, IP, SP, BP, PQ, TH, and TL.

If your program has taken an unexpected turn, you may be able to compensate for the problem and continue debugging if you change the value of a register or a flag. You can change a flag value before a dump or looping instruction to test a different branch of code, for example. You can change the instruction pointer (CS:IP) to jump to any code in your program or to execute code you have assembled elsewhere in memory.

To change the value of any register, move the cursor to the register value you want to change and overtype the old value with the new value. The cursor automatically moves to the next register.

Although you cannot change the value of the flag register numerically in the Register window, you can conveniently toggle the values of each flag using either the mouse or the keyboard:

- To toggle a flag with the mouse, double-click the flag.
- To toggle a flag using the keyboard, move the cursor to the flag and press any key except ENTER, TAB, or ESC. After toggling a flag, CodeView moves the cursor to the next flag.

To restore the value of the last flag or register that you changed, choose Undo from the Edit menu or press ALT+BACKSPACE. If you happen to lose the cursor somewhere in the register window, press TAB. The TAB key moves the cursor to the next register or flag that can be changed.

The 8087 Window

The 8087 window displays the current status of the math coprocessor's registers and flags. If you are debugging a program that uses the software-emulated coprocessor, the emulated registers are displayed. To open the 8087 window, choose 8087 from the Windows menu or press ALT+8.

The display in the 8087 window is the same as the display produced by the 8087 (**7**) command, except that the window is continually updated to show the current status of the math coprocessor. For more information about the display, see the "8087" command on page 448.

If your program uses floating-point libraries provided by several Microsoft languages, or if your program does not use floating-point arithmetic, the 8087 window and 8087 command display the message:

```
Floating point not loaded
```

CodeView displays this message until at least one floating-point instruction has been executed.

The Memory Windows

Memory windows display memory in a number of formats. CodeView allows two Memory windows to be open at the same time. You can open a Memory window in several ways:

- From the Windows menu, choose Memory 1 or Memory 2.

- From the Options menu, choose Memory1 Window when no Memory windows are open.
- In the Command window, enter the View Memory (**VM**) command.
- At the keyboard, press ALT+5 or ALT+6.

By default, memory is displayed as bytes or as the last type specified by a Memory Enter (**ME**), Memory Dump (**MD**), or View Memory (**VM**) command. The byte display shows hexadecimal byte values followed by an ASCII representation of those byte values. For values that are outside the range of printable ASCII characters (decimal 32 to 127), CodeView displays a period (**.**).

How to Change Memory Display Format

It is not always most convenient to view memory as byte values. If an area of memory contains character strings or floating-point values, you might prefer to view them in a directly readable form.

To change the display format of a Memory window, choose Memory1 Window or Memory2 Window from the Options menu. CodeView displays a dialog box where you can choose from a variety of display options. When the cursor is in a Memory window, you can presss CTRL+O to display the corresponding Memory Window Options dialog box. You can also set memory display options using the View Memory (**VM**) command. For detailed information about the display options, see “View Memory” on page 431.

To cycle through the display formats, click the `<Sh+F3=Mem1Fmt>` or `<Sh+F3=Mem2 Fmt>` buttons on the status bar, or press SHIFT+F3. Pressing CTRL+SHIFT+F3 displays the cycle in reverse order.

When you first open the Memory window, it displays memory starting at address DS:00. To change the starting address, use one of the commands to set Memory window options. You can specify the starting address or enter an expression to use as the starting address.

You can also type over the *segment*:*offset* addresses shown in the left column of the Memory window to change the displayed addresses. Move the cursor to an address in the window, or repeatedly press TAB until the cursor is on an address, and type a new address.

How to Change Memory Directly

To change the values in memory, overtype the value you want to change. To move quickly from field to field in the Memory window, press TAB. You can change memory by entering new values for the format that is displayed or by typing over the raw bytes in the window. CodeView ignores the input if you

press a key that does not make sense for the current format (for example, if you type the letter X in anything but ASCII format).

To undo a change to memory, choose Undo from the Edit menu, or press ALT+BACKSPACE.

How to View Memory at a Dynamic Address

Live expressions make it easy for you to watch a dynamic view of an array or pointer in the Memory window. “Live” means that the starting address of memory in the window changes to reflect the current value of an address expression.

To create a live expression, choose the Memory1 Window or Memory2 Window command from the Options menu. In the Memory Window Options dialog box, type in an address expression, then turn on the Re-evaluate Expression Always (Live) option.

It is usually more convenient to view an item in the Watch window than in the Memory window. However, some items are more easily viewed using live expressions. For example, you can examine what is currently on top of the stack by entering SS:SP as the live expression.

The Help Window

In CodeView, you can request Help:

- From the Help menu.
- By pressing F1 when the cursor is on the keyword, menu, or dialog box for which you want Help.
- By clicking the right mouse button on a Help keyword.
- Using the Help (**H**) command.
- By choosing Help from the Windows menu. You can also press ALT+0 for Help on CodeView windows.

The Microsoft Advisor Help window is displayed whenever you request Help for CodeView. For information about getting the most out of the Microsoft Advisor Help system, see Chapter 21.

CodeView Menus

Many commands that you are likely to use frequently are in the CodeView menus. This section describes the menus and the commands or options in each menu. Not all commands are available in both versions of the CodeView debugger. When applicable, the menu descriptions discuss command availability.

The File Menu

The File menu contains commands to load source files and other ASCII text files into the Source window, print from the active window, start an operating-system shell, and end the debugging session.

The following table summarizes the commands on the File menu. Commands marked with an asterisk are not shown in the CVW File menu:

Command	Purpose
Open Source	Opens a source, include, or other text file
Open Module	Opens a source file for a module in the program
Print*	Prints all or part of the active window
DOS Shell*	Goes to the operating-system prompt temporarily
Exit	Exits CodeView

Open Source

The Open Source command displays the Open Source File dialog box. You can select the name of the source file, include file, or other text file to display in the active Source window.

Open Module

The Open Module command displays the Open Module dialog box. This dialog box provides an easy way to load the source file for any module in your program. The dialog box lists the source files that make up the modules in the program you are debugging. Only those modules that include line-number or full symbolic information are listed.

CodeView displays the source file you choose in the active Source window.

Print

In CodeView for MS-DOS only, the Print command displays the Print dialog box, which offers several options to write information in the active window to a device or a file. You can print text in the active window in the following ways:

- Window view, which prints text that currently appears in the active window
- Complete window contents, which prints the contents of the active window, including what has scrolled out of the window

To print to a file, specify a filename in the dialog box. To append the printed text to the end of the file, select Append. To overwrite a file that already exists, select Overwrite. If you specify a device instead of a file, you can choose either Append or Overwrite.

To print directly to a printer, specify the name of the printer port such as LPT1 or COM2. You must specify a filename or a device name. CodeView reports an error if you omit the name.

DOS Shell

In MS-DOS only, you can use the DOS Shell command to leave CodeView temporarily and go to the operating-system prompt.

When you choose the Shell command, CodeView starts a second copy of the command processor specified by the COMSPEC environment variable. If there is not enough memory to open a new shell, a message appears. Even if you do have enough memory to start a command shell, you may not have enough memory to execute large programs from the shell.

While in the shell, do not start any terminate-and-stay-resident (TSR) programs, such as MSHERC.COM, and do not delete files you are working on during your debugging session. Also, do not delete any files used by CodeView, such as the CURRENT.STS file.

To return to CodeView, type exit at the operating-system prompt to close the shell. For more information about starting a shell, see the "Shell Escape" command on page 443.

Exit

The Exit command saves the current CodeView environment and returns to the program that called CodeView, such as COMMAND.COM, PWB, or another editor. CodeView saves the window arrangement, watch expressions, option settings, and most breakpoints in the state file, CURRENT.STS. It saves current color settings in CLRFILE.CV4 if you are using CV and in CLRFILE.CVW if you are using CVW.

When you start the debugger at a later time, CodeView restores these settings. To prevent CodeView from restoring the information it stores in CURRENT.STS, start the debugger with the /TSF option or use the **Statefileread** entry in your TOOLS.INI file.

The Edit Menu

The Edit menu contains commands to undo changes to window's fields, copy selected text to the clipboard, and paste the contents of the clipboard into a window. For more details on editing in CodeView's windows, see "CodeView Windows" on page 321.

The following table summarizes the commands on the Edit menu:

Command	Purpose
Undo	Reverses the last editing change
Copy	Copies the selected text to the clipboard
Paste	Inserts the contents of the clipboard at the cursor

Undo

The Undo command (ALT+BACKSPACE) reverses the last editing action.

Copy

The Copy command (CTRL+INS) copies selected text to the clipboard. Because input to CodeView is restricted to single lines, you can copy only a single line of text. If you select more than a single line of text, the Copy command is disabled and pressing CTRL+INS has no effect.

Paste

The Paste command (SHIFT+INS) inserts text from the clipboard at the cursor in the Command window.

The Search Menu

The Search menu provides commands to find strings and regular expressions in source files and to locate the definitions of labels and routines.

The following table summarizes the commands on the Search menu:

Command	Purpose
Find	Searches for a text string or pattern in the source file
Selected Text	Searches for the selected text in the source file
Repeat Last Find	Repeats the last text search
Label/Function	Searches for a label or function definition in the program

Find

The Find command displays the Find dialog box. In the Find What text box, type the text or pattern you want to find. You can also select text in a window and then choose Find. The text you selected is shown in the dialog box.

You can select options in the dialog box to modify the way CodeView searches for text. The following options are available:

Whole Word
: CodeView matches the text only when it occurs as a word by itself. For example, when you search for the pattern `print` with the Whole Word option, CodeView finds `print("eeep")`, but not `error_print("eeep")`.

Match Case
: CodeView matches the text when each letter in the pattern has the same case as the source file. For example, the pattern `fish` matches `fish`, but not `Fish`.

Regular Expression
: CodeView treats the text as a regular expression. Regular expressions provide a powerful way to specify patterns that match several different sections of text. For more information about regular expressions, see Appendix A.

To search for a regular expression in the active Source window using the Command window, you can type the Search (/) command followed by the string. CodeView searches the file starting at the current position. CodeView places the cursor on the next occurrence of the search pattern. If the end of the file is reached without finding a match, CodeView wraps around and continues searching from the beginning of the file.

Selected Text

The Selected Text command (CTRL+\) searches for the next occurrence of the selected text in the Source window.

Repeat Last Find

The Repeat Last Find command (ALT+/) searches for the next occurrence of the search pattern, including search options, you last specified.

Label/Function

The Label/Function command lets you search the program's symbolic information for the definition of a label or routine. When you choose Label/Function, the Find Label/Function dialog box appears. The currently selected text or the word at the cursor comes up in the Label/Function Name text box. You can search for this name or type in a different label or routine name.

When you choose OK, CodeView searches the symbolic information in the program for the name. When the label or routine name is found, CodeView positions the cursor at the name in the source file.

To view the current program location after searching, choose the first item in the Calls menu or type the Current Location (**.**) command in the Command window.

The Run Menu

The Run menu consists of commands to restart the program, animate the program in slow motion, change the program's arguments, load a new program, or configure the modules CodeView is using.

The following table summarizes the commands on the Run menu:

Command	Purpose
Restart	Restarts the program
Set Runtime Arguments	Changes the program's run-time arguments and restarts the program
Animate	Executes the program in slow motion
Load	Loads a new program to debug, sets run-time arguments, and configures CodeView's modules

Restart

The Restart command resets execution to start at the beginning of the program. After you issue the command, CodeView:

- Initializes all program variables.
- Resets the pass counts for all breakpoints.
- Preserves existing breakpoints, watch expressions, and the program's command-line arguments.

You can use Restart any time after execution stops: at a breakpoint, while stepping or tracing, or when your program ends. If your program redefines interrupts, Restart may not work correctly because it does not execute any cleanup or exit-list code in the program. If your program requires this code to be executed, let the program run to the end before restarting, or use the Display Expression (**?**) command in the Command window to call the cleanup routines. For more information on calling program routines, see "Display Expression" on page 452.

Set Runtime Arguments

The Set Runtime Arguments command lets you change your program's command-line arguments. When you set new arguments, CodeView restarts the program.

Animate

The Animate command executes your program in slow motion. CodeView highlights each statement in the Source window as your program executes. This allows you to see the flow of execution. To stop animation, press any key.

You can set the animation speed with the Trace Speed command from the Options menu or with the Trace Speed (**T**) Command-window command.

Load

The Load command displays the Load dialog box, which you can use to:

- Load executable (.EXE or .DLL) files.
- Change the program's command-line arguments.
- Specify different CodeView components from those specified in TOOLS.INI, such as a different expression evaluator or the p-code execution model.

Loading Programs or DLLs

To load program or DLL symbols into the debugger, type a filename in the File to Debug text box, or use the mouse or keyboard to select a file from the File List box. Use the Drives/Dirs list box to change to a different drive or directory.

Set Command-Line Arguments

Use the Arguments text box to change the command-line arguments to the program you are debugging or to set entirely new arguments. Type the arguments to your program as you would on the command line.

Configure CodeView Modules

CodeView uses a default setting for an execution model, transport layer, and expression evaluator if any of these is not specified in TOOLS.INI. Choose the Configure button to load different CodeView DLLs. The Configure dialog box lists the DLLs that CodeView has loaded. CodeView loads several DLLs that are required to debug your programs. These DLLs include:

- Expression evaluators for various languages and environments.
- Execution models for various operating systems.
- Execution models for p-code.
- Transport layers.

To load new DLLs, choose the Change buttons on the right side of the dialog box.

The Data Menu

The Data menu provides commands to add and delete watch expressions and breakpoints. Watch expressions allow you to observe how variables change as your program executes and also to expand arrays and dereference pointers. Breakpoints allow you to stop execution of your program to check the values of your variables, determine execution flow, and change how your program executes.

For more information about watch expressions, see Chapter 11, “Using Expressions in CodeView” and the “Add Watch Expression” command on page 436.

The following table summarizes the commands on the Data menu:

Command	Purpose
Add Watch	Adds an expression to the Watch window
Delete Watch	Deletes an expression from the Watch window
Set Breakpoint	Sets a breakpoint in the program
Edit Breakpoints	Modifies or removes existing breakpoints
Quick Watch	Displays a quick view of a variable or expression

Add Watch

The Add Watch command (CTRL+W) displays the Add Watch dialog box, which shows the selected text or the word at the cursor in the Expression text box. You can enter a different expression or add a format specifier to the expression. When you choose OK, the expression is added to the end of the Watch window.

Delete Watch

The Delete Watch command (CTRL+U) displays the Delete Watch dialog box, which displays a list of the watch expressions in the Watch window. Select the expression you want to delete from the list and choose OK. Choose the Clear All button to remove all expressions from the Watch window.

You can also delete expressions directly from the Watch window. Use the mouse or the cursor keys to move the cursor to the expression you want removed, and press CTRL+Y.

Set Breakpoint

The Set Breakpoint command displays the Set Breakpoint dialog box, which allows you to select from several kinds of breakpoints and set options for each type. The following list describes the breakpoints you can set:

Break at Location
: This is the simplest type of breakpoint. You specify an address or a line number where you want execution to pause. To specify a line number, precede it with a period (.); otherwise, CodeView will interpret it as an address. When your program's execution reaches the breakpoint location, your program stops temporarily, and you can enter CodeView commands.

Break at Location if Expression is True
: You specify a location and an expression. Whenever execution reaches that location, CodeView checks the expression. If the expression is true (nonzero), the breakpoint is taken. Otherwise, execution continues.

Break at Location if Expression has Changed
: You specify a location and an expression that represents a variable or any portion of memory. To specify a range of memory, enter the length of the range in the Length text box. CodeView checks the variable or the range of memory when execution reaches the breakpoint location. If the value of any byte has changed since the last time CodeView checked, the breakpoint is taken. Otherwise, execution continues.

Break When Expression is True
: This breakpoint is taken whenever the expression becomes true. CodeView evaluates the expression after every line or every instruction, instead of only at a certain location. As a result, this type of breakpoint can greatly slow your program's execution.

Break When Expression has Changed
: CodeView checks the variable or the range of memory as each line or each instruction is executed. You can also specify a range with the Length text box. This type of breakpoint can also slow your program's execution.

Each breakpoint is numbered, beginning with 0. For each type of breakpoint, you can set several options. If you try to use an option that does not apply to a certain breakpoint, CodeView displays N/A in the edit box and ignores that option. The options are:

Location
: Specifies where CodeView should evaluate the breakpoint.

Expression
: Specifies an expression that causes a break when it becomes true or a location that is to be watched for changes.

Length
: Specifies a range of memory (starting at the address in the Expression text box) to watch for changes.

Pass Count
: Specifies the number of times to pass over the breakpoint when it otherwise would be taken. For example, a pass count of 10 tells CodeView to ignore the breakpoint ten times.

Commands
: Specifies a list of Command-window commands, separated by semicolons, that are executed when the breakpoint is taken. If several breakpoints with commands are taken, the commands are queued and executed in first-in, first-out order.

As shortcuts, you can also set simple (break at location) breakpoints with the following methods:

- Double-click the line in the Source window.
- Move the cursor to the breakpoint location in the Source window and press F9.

A line with a breakpoint is highlighted. In the Mixed and Assembly modes, an assembly-language comment that displays the breakpoint number appears. For example:

```
0047:0b30 57       push di                    ;BK0
```

In this example, breakpoint number 0 is set at the address 0047:0B30.

You can also set breakpoints with the Breakpoint Set (**BP**) command. See the "Breakpoint Set" command on page 405.

Edit Breakpoints

The Edit Breakpoints command displays the Edit Breakpoints dialog box, where you can add, remove, change, enable, and disable breakpoints. Select a breakpoint from the list of breakpoints, then choose one of the command buttons on the right side of the dialog box.

The list of breakpoints in the Edit Breakpoints dialog box shows the current state of each breakpoint in your program. For more information on the format of the breakpoint list, see the "Breakpoint List" command on page 405.

The command buttons in the Edit Breakpoints dialog box are described in the following table:

Button	Description
Add	Adds a new breakpoint
Remove	Removes the selected breakpoint
Modify	Modifies the same breakpoint
Enable	Activates a disabled breakpoint
Disable	Disables an active breakpoint
Clear All	Removes all breakpoints

If you choose the Modify button, CodeView displays the Set Breakpoint dialog box with the appropriate options set for the breakpoint you selected. You can then modify the options and set the breakpoint just as you do with the Set Breakpoint command.

When you disable a breakpoint by selecting the Disable button, CodeView does not evaluate the breakpoint. Program execution continues as if the breakpoint was never set.

You may encounter several occasions where it is useful to disable a breakpoint. Sometimes a certain breakpoint is not practical when you are debugging a routine nested deeply in your program. You can re-enable the breakpoint later when you really need it. Also, conditional breakpoints are evaluated at every program step and can slow execution. You can disable some conditional breakpoints in areas of your program where you're certain you won't need them.

Quick Watch

The Quick Watch command (SHIFT+F9) displays the Quick Watch dialog box, which shows the variable at the cursor position or the selected expression. The Quick Watch dialog box is similar to the Watch window. However, you mainly use Quick Watch for a quick exploration of the current values in an array or a pointer-based data structure, rather than as a method to constantly display the values.

The Quick Watch dialog box automatically expands structures, arrays, and pointers to their first level. You can expand or contract an element just as you can in the Watch window. If the expanded item needs more lines than the Quick Watch dialog box can display, you can scroll the view up and down.

Choose the Add Watch button to add a Quick Watch item to the Watch window. Expanded items appear in the Watch window as they are displayed in the Quick Watch dialog box.

For complete information on using the Quick Watch dialog box, see the "Quick Watch" command on page 453.

The Options Menu

The Options menu contains commands to change the default behavior of CodeView commands and the display status of CodeView windows. You can also set display options with various Command-window commands. When the cursor is in one of the Source, Memory, or Local windows, you can press CTRL+O to display the window's Options dialog box.

For menu items that are toggles, a bullet appears to the left of the item when the option is turned on. No bullet appears when it is turned off.

The following table summarizes the commands on the Options menu:

Command	Purpose
Source1 Window	Sets Source window 1 display options
Source2 Window	Sets Source window 2 display options
Memory1 Window	Sets Memory window 1 display options
Memory2 Window	Sets Memory window 2 display options
Local Options	Sets Local window display options
Trace Speed	Sets animation speed
Language	Sets the expression evaluator
Horizontal Scrollbars	Toggles horizontal scroll bars on windows
Vertical Scrollbars	Toggles vertical scroll bars on windows
Status Bar	Toggles the status bar display
Colors	Changes colors of CodeView screen elements
Screen Swap	Toggles screen exchange
Case Sensitivity	Toggles case sensitivity of symbols
32-Bit Registers	Toggles display of 32-bit registers
Native	Toggles display of p-code or machine code instructions

Source Window

The Source Window command displays the Source Window Options dialog box. In this dialog box, you can set the source display mode and other options for the current Source window. These options are as follows:

Option	Description
Follow CS:IP thread of control	Keeps the current program location visible in the active Source window.
Source	Displays the source code for the program.
Mixed Source and Assembly	Displays each source line followed by the disassembly of the code generated for that line.
Assembly	Displays a disassembly of the machine code in your program.
Tab Length	Sets the number of spaces to which tab characters expand in the source file.
Show Machine Code	Shows the address and hexadecimal representation of the machine code in Mixed and Assembly modes.
Show Symbolic Name	Shows the symbol name in assembly-language displays instead of the numeric value of the symbol.

The Source Window Options dialog box also contains all the options available with the VS (View Source) command. For information on the VS command, see “VS (View Source)” on page 433.

Memory Window

The Memory Window command displays the Memory Window Options dialog box, where you can set the starting address and display format of the active Memory window. For details, see “The Memory Windows” on page 330 and the “View Memory” command on page 431.

Local Options

You can specify the scope of variables to be displayed in the Local window. When you select Local Options from the Options menu, a dialog box appears in which you can select any combination of lexical, function, module, executable, and global scopes. You can also toggle the display of addresses in the Local window from the Local Options dialog box. When you turn Show Addresses on, the BP-relative address of each local variable is shown in the Local window. Otherwise, the Local window shows only the names of the variables.

You can also use the Options (**OL**) command in the Command window to specify the scope of variables to be displayed in the Local window. For information about the Options command, see page 422.

Trace Speed

The Trace Speed command displays the Trace Speed dialog box, which presents a list of three speeds from which you can select.

When you use the Animate command to run your program in slow motion, CodeView pauses execution between each step. The duration of the pause is set with the Trace Speed command. **Slow** pauses for 1/2 second. **Medium** pauses for 1/4 second. **Fast** runs the program as fast as possible while still updating CodeView windows and evaluating breakpoints and watch expressions.

Language

The Language command displays the Language dialog box, which presents a list of the expression evaluators that CodeView has loaded, plus the **Auto** option.

In your TOOLS.INI file, you can configure CodeView to load a number of different expression evaluators. You can also load expression evaluators by choosing Load from the Run menu. Only one expression evaluator can be active at a time.

The **Auto** setting is the default. It tells CodeView to set the expression evaluator automatically based on the extension of the source file you are debugging in the current Source window. For more information on expression evaluators, see “Configuring CodeView with TOOLS.INI” on page 301.

For more information on using expression evaluators, see Chapter 11, “Using Expressions in CodeView.”

Horizontal Scrollbars

The Horizontal Scrollbars command toggles the horizontal scroll bars on and off. When scroll bars are off, you can drag the bottom window frame, as well as the size box, to resize the window.

Vertical Scrollbars

The Vertical Scrollbars command toggles the vertical scroll bars on and off. When scroll bars are off, you can drag the right window frame, as well as the size box, to resize the window.

Status Bar

The Status Bar command toggles the status bar on and off. When the status bar is off, you gain an extra line of space for windows.

Colors

The Colors command displays a dialog box that lets you change the colors of CodeView screen elements. The Item list box displays all the elements of the debugging screen. The Foreground and Background list boxes show the current color settings for the highlighted element in the Item list box.

To change the color of a screen element, choose an element in the Item list box, then choose foreground and background colors. When you are done, click the OK button. Your new color settings take effect as soon as you exit the dialog box.

If you make a number of changes and want to go back to your previous color settings, click the Reset button. You can then start changing colors again. To close the dialog box without making any changes, click the Cancel button. To reset to the standard CodeView colors, click the Use Default button.

When you specify colors using the Colors command in CodeView, the colors are saved in CLRFILE.CVW if you are using CodeView for the Windows Operating System and in CLRFILE.CV4 if you are using CodeView for DOS. CodeView saves these files in the directory specified by the INIT environment variable or in the current directory if no INIT environment variable is set. These settings become the new default colors.

Screen Swap

The Screen Swap command toggles screen exchange on or off. By default, CodeView switches to your program's output screen whenever you execute code in the program. CodeView uses either screen flipping or screen swapping, depending on the command-line options you used to start the debugger. See “Set Screen-Exchange Method” on page 313.

If your program sends no output to the screen, you'll probably want to turn Screen Swap off. This setting continuously displays CodeView's screen while your program executes.

If Screen Swap is off and your program writes to the screen, a portion of the CodeView display may be overwritten. If this happens, type the Refresh (**@**) command in the Command window.

Case Sensitivity

The Case Sensitivity command toggles case sensitivity on or off. When Case Sensitivity is on, CodeView treats symbol names as case sensitive (that is, a lowercase letter is different from its corresponding uppercase letter). This option affects only commands that deal with symbols in your program; it does not affect the text-searching commands.

32-Bit Registers

The 32-Bit Registers command toggles 386 mode on and off. When 386 mode is on, a bullet appears next to the command on the menu, and CodeView displays the 32-bit registers in the Register window. In this mode, CodeView can also assemble instructions that use 32-bit registers or memory operands.

Native

When you are debugging a program that uses p-code, you use the Native command to toggle between p-code instructions and native machine instructions. With Native mode on, CodeView displays your program's native CPU instructions. With Native mode off, CodeView displays the instructions in p-code.

For more information on debugging p-code, see page 363.

The Calls Menu

The Calls menu shows what routines have been called into your program during debugging. Its contents change to reflect their current status. The current routine is at the top of the menu; the routine that called it appears just below. Routines are listed in the reverse order in which they were called. At the bottom of the list is your program's main routine. In C, for example, **main** appears at the bottom. When you are debugging a Windows-based application, **winmain** is at the bottom of the list.

The Calls menu is empty until the program enters at least one routine that creates
a stack frame. Listed with each routine name are the arguments to each routine in parentheses. The menu's width expands to accommodate the widest entry.

Arguments are shown in the current radix, except for pointers, which are always shown in hexadecimal.

When you choose a routine from the Calls menu, CodeView displays the source code for that routine and updates the Local window to show the local variables in that routine. The cursor moves to the return location to show the next line or instruction that will be executed when control returns to that routine.

To step out of deeply nested code, choose a routine and then press F7.

Choosing a routine from the Calls menu does not affect program execution; it provides you with a convenient way to view a routine's source code and local variables. However, since the cursor is positioned at the return location, you can press F7 to execute through the stack of nested calls to that line. This is especially convenient when you find you've accidentally traced into a deeply nested set of routines which you know to be bug-free. Rather than continue a tedious trace until you work your way out of the stack of routines, you can choose a routine from the Calls menu and press F7. CodeView executes through the nested routines until control returns to the point you chose.

A routine may not be visible in the Calls menu under the following circumstances:

- You have traced only startup or termination routines from the run-time library.
- Routine calls are nested so deeply that not all routines appear on the menu.
- The stack has been corrupted.
- CodeView cannot trace through the stack frame because the BP register is overwritten.

The Windows Menu

If you get lost among your windows, try the Arrange command.

The Windows menu contains commands that activate, open, close, tile, arrange, and manipulate CodeView windows. There is also a command to view your program's output screen. A bullet appears to the left of the active window when you open this menu.

All the windows are numbered. You can quickly open or switch to a window by pressing ALT plus the window's number.

The following table summarizes the commands on the Windows menu and the corresponding shortcut keys:

Command	Shortcut Key	Purpose
Restore	CTRL+F5	Restores the active window to its size and position before it was maximized or minimized
Move	CTRL+F7	Moves the active window using the keyboard
Size	CTRL+F8	Sizes the active window using the keyboard
Minimize	CTRL+F9	Shrinks the active window to an icon
Maximize	CTRL+F10	Enlarges the active window to full screen
Close	CTRL+F4	Closes the active window
Tile	SHIFT+F5	Arranges all open windows to fill the entire window area
Arrange	ALT+F5	Arranges all open windows to an effective layout for debugging
Help	ALT+0	Opens or switches to the Help window
Local	ALT+1	Opens or switches to the Local window
Watch	ALT+2	Opens or switches to the Watch window
Source 1	ALT+3	Opens or switches to Source window 1
Source 2	ALT+4	Opens or switches to Source window 2
Memory 1	ALT+5	Opens or switches to Memory window 1
Memory 2	ALT+6	Opens or switches to Memory window 2
Register	ALT+7	Opens or switches to the Register window
8087	ALT+8	Opens or switches to the 8087 window
Command	ALT+9	Opens or switches to the Command window
View Output	F4	Swaps the CodeView screen for the program's output screen

Source and Memory Windows

You can open as many as two Source and two Memory windows. At least one Source window must be open at all times. To close a window, use the Close command (CTRL+F4).

Help, Local, Watch, Register, 8087, and Command Windows

CodeView can display one of each of these windows. The Register window has an additional shortcut key (F2) you can use to open or close it.

When you open the Help window, CodeView displays the last Help screen you viewed. If you have not yet viewed Help during the session, CodeView displays the top-level contents in the Microsoft Advisor.

View Output

To view your program's output screen, choose View Output or press ALT+F4. CodeView displays the output screen until you press a key.

The Help Menu

The Help menu contains commands to access the Microsoft Advisor Help system. When you choose a Help command, CodeView opens the Help window if it is not already open and displays the appropriate part of the Microsoft Advisor.

When the Help window is open, you can browse through Help with mouse and keyboard commands. All Microsoft environments provide the same mouse and keyboard commands to access the Microsoft Advisor. For more information on getting the most out of Help, see Chapter 21.

The following table summarizes the commands on the Help menu:

Command	Purpose
Index	Displays the table of Microsoft Advisor indexes
Contents	Displays the Microsoft Advisor contents screen
Topic	Displays Help on the current word
Help on Help	Displays Help on using the Microsoft Advisor
About	Displays CodeView copyright and version information

Index

The Index command displays a table of available indexes. Each part of the Help system has its own index.

Contents

The Contents command (SHIFT+F1) displays the contents for the entire Help system. This screen lists the table of contents for each Help system component.

Topic

The Topic command (F1) displays help on the word at the cursor or the selected text. When you open the Help menu, CodeView displays the topic in the menu. When you choose the Topic command, CodeView displays information on the indicated topic in the Help window.

Help on Help

The Help on Help command displays information on the Microsoft Advisor itself. It describes how the system is organized, how the mouse and keyboard

commands are used to browse through Help, and how to use the various kinds of buttons you encounter.

About

The About command displays the CodeView copyright and version information in a dialog box.

CHAPTER 10

Special Topics

Debugging in the Windows Operating System

The Microsoft CodeView for the Windows operating system debugger (CVW) is a powerful tool for analyzing the behavior of Microsoft Windows-based programs. With CVW, you can test the execution of your application and examine your application's data. You can isolate problems quickly because you can display any combination of variables—global or local—while you halt or trace your application's execution.

Comparing CVW with CV

The CVW windows, menus, and commands are used in the same way as for CV. See Chapter 9, "The CodeView Environment," for details on the format of CodeView windows and how to use the windows and menus. Like the MS-DOS CodeView, CVW allows you to display and modify any program variable, section of addressable memory, or processor register. However, CodeView for Windows differs from CV in the following ways:

- Because the Windows operating system has a special use for the ALT+/ key combination used by CV to repeat a search, CVW uses CTRL+R.
- CVW tracks your application's segments and data as the Windows operating system moves them in memory. Thus, when you refer to an object by name, CVW always supplies the correct address.

CVW also provides six additional Command-window commands for Windows-based program debugging, which are summarized in the following list:

Windows Display Global Heap (**WDG**)
: Displays memory objects in the global heap.

Windows Display Local Heap (**WDL**)
: Displays memory objects in the local heap.

Windows Dereference Local Handle (**WLH**)
Dereferences a local heap handle to a pointer.

Windows Dereference Global Handle (**WGH**)
Dereferences a global heap handle to a pointer.

Windows Display Modules (**WDM**)
Displays a list of the application and DLL modules currently loaded in the Windows operating system.

Windows Kill Application (**WKA**)
Terminates the task that is currently executing by simulating a fatal error.

For details on using these commands, see "CVW Commands" on page 357.

The following CV features are not available in CVW.

- The Print command from the File menu.
- The DOS Shell command from the File menu and the corresponding Shell (**!**) Command-window command.
- The Screen Swap command from the Options menu and the corresponding Options (**OF**) Command-window command.

Preparing to Run CVW

Before beginning a CVW debugging session, you must ensure that your system is configured correctly and the Windows-based application you are going to debug is compiled and linked with the options that generate CodeView debugging information.

For information on setting up your system and configuring CodeView, see "Setting up CodeView" on page 299. For information on preparing programs for use with CodeView, see "General Programming Considerations" on page 294 and "Compiling and Linking" on page 295.

The SETUP program installed the two files in your MASM\BIN subdirectory: CVW.EXE and CVW1.386. These two files must be in the current path. Also, in order for CodeView to run properly with the Windows operating system, the line:

```
device = drive:\MASM\BIN\CVW1.386
```

must appear under the [386 Enh] section of your SYSTEM.INI file, where *drive* is the hard disk drive where MASM resides.

The window that CodeView uses cannot be sized or moved as can other Windows operating system applications. You can specify a different starting position for CodeView using the /X and /Y command-line options. For information on the CodeView command-line options, see page 310.

If You Use the Windows Operating System Version 3.0

If the Windows operating system Version 3.0 is running in Standard Mode and CodeView is invoked with the /X option or with no parameters, Windows Version 3.0 will generate an error when CodeView attempts to switch to protected mode. This conflict only occurs with Windows Version 3.0 running in standard mode. You can avoid this by configuring a PIF file.

For CodeView to run under Windows Version 3.0, create a PIF file using the PIF Editor. In the Optional Parameters field, enter only a question mark (?). This instructs Windows 3.0 to prompt for additional options when CodeView is invoked. When the PIF file is run, it will prompt for the command line. Specify the appropriate parameter as listed below, followed by the name of the program to be debugged.

Windows Mode	Switch(es)
Real	/D or none
Standard	/D
386 enhanced	/D (or /E if expanded memory is available)

Starting a Debugging Session

Like most Windows-based applications, CVW can be started in several ways. You can double-click the CVW icon and respond to CVW's prompts for arguments, or you can run CVW by using the Run command from the Program Manager File menu.

To specify CVW options, choose the Run command from the Program Manager File menu. The Windows operating system displays a dialog box where you can enter the appropriate options for your debugging session. For specific information on CodeView command-line syntax and options, see "The CodeView Command Line" on page 308.

You can run CVW to perform the following tasks:

- Debug a single application
- Debug multiple instances of an application
- Debug multiple applications
- Debug dynamic-link libraries (DLLs)

This section describes the methods you use to perform these tasks and summarizes the syntax of the CVW command line for each task.

Starting CVW for a Single Application

After you start CVW from Windows, CodeView displays the Load dialog box.

To start debugging a single application:

1. Type the name of the application in the File to Debug text box. CVW assumes the .EXE filename extension if you do not include an extension for the application name. You can also pick the program that you want to debug by choosing it from the Files List box.
2. If you want to specify command-line arguments, move the cursor to the Arguments text box and type the program's command line.
3. Choose OK.

   CVW loads the application and displays the source code for the application's **WinMain** routine.
4. Set breakpoints in the code if you desire.
5. Use the Go (**G**) command (F5) to begin executing the application.

To avoid the startup dialog boxes:

1. Choose the Run command from the Windows File menu.
2. Type the application name and arguments on the CVW command line. Use the following syntax to start debugging a single application:

   CVW [[*options*]] *appname*[[.EXE]] [[*arguments*]]
3. Choose OK.

Starting CVW for Multiple Instances of an Application

The Windows operating system can run multiple instances of an application, which can cause problems. For example, each instance of an application might corrupt the other's data. To help you solve such problems, CVW allows you to debug multiple instances of an application. The breakpoints you set in your application apply to all of the instances. To determine which instance of the application has the focus in CVW, examine the DS register.

To debug multiple instances of an application:

1. Start CVW as usual for one instance of your application.
2. Run additional instances of your application by choosing the Run command from the Windows File menu.

You cannot specify the application name more than once on the CVW command line. Any additional application names are passed as arguments to the first application.

Starting CVW for Multiple Applications

You can debug two or more applications at the same time, such as a dynamic data exchange (DDE) client and server.

To debug several applications at the same time:

1. Start CVW as usual for a single application.
2. Choose Load from the Run menu and choose other applications that you also want to debug.
3. Set breakpoints in either or both applications. You can use the Open Module command from the CVW File menu to display the source code for the different modules. If you know the module and the location or function name, you can use the context operator (**{ }**) to directly set breakpoints in the other applications.
4. Use the Go (**G**) command (F5) to start running the first application.
5. Choose the Run command from the Windows File menu to start running the second application.

You can also use the /L option on the CVW command line to load the symbols for additional applications, as shown in this example:

```
CVW /Lsecond.exe /Lthird.exe first
```

The /L option and name of each additional application must precede the name of the first application on the command line. You must specify the .EXE filename extension for the additional applications. Repeat the /L option for each application to be included in the debugging session.

Once CVW starts, choose the Run command from the Windows File menu to start executing the additional applications.

Note Global symbols with the same name in several applications (such as `WinMain`) may not be distinguished. You can use the context operator to specify the exact instance of a symbol.

Starting CVW for DLLs

You can debug one or more DLLs while debugging an application.

To debug a DLL at the same time as an application:

1. Start CVW as usual for the application.
2. Choose Load from the Run menu and type the name of the DLL.

3. Set breakpoints in the application or DLL. You can use the Open Module command from the CVW File menu to display the source code for the different modules.
4. Use the Go (**G**) command (F5) to continue executing the application.

You can also use the /L option on the CVW command line to specify the DLLs, as shown in this example:

```
CVW /Lappdll appname
```

The /L option must precede the name of the application. Repeat the /L option for each DLL you want to debug.

Debugging the LibEntry DLL Initialization Routine

CVW allows you to debug the **LibEntry** initialization routine of a DLL. If your application implicitly loads the library, however, a special technique is required to debug the **LibEntry** routine.

An application implicitly loads a DLL if the library routines are imported in the application's module-definition (.DEF) file or if your application imports library routines through an import library when you link the application. An application explicitly loads a DLL by calling the **LoadLibrary** routine.

If your application implicitly loads the DLL and you specify the application in the Command Line dialog box, Windows automatically loads the DLL and executes the **LibEntry** routine when it loads the application. This gives you no opportunity to debug the **LibEntry** routine since it is executed when the application is loaded and before CVW gains control.

To gain control before the **LibEntry** routine is executed, you must set a breakpoint in the **LibEntry** routine before the DLL is loaded.

To set this breakpoint:

1. In the CVW Load dialog box, provide the name of a "dummy" application that does not load the library instead of the name of your application. The CALC.EXE program is provided for this purpose.
2. Load the DLL by using the Load command from the Run menu.
3. Choose the Open Module command from the CVW File menu and select the module containing the **LibEntry** routine.
4. Set at least one breakpoint in the **LibEntry** routine.
5. Use the Go (**G**) command (F5) to start the dummy application.
6. Run your application using the Run command from the Windows File menu. CVW resumes control when the breakpoint in the **LibEntry** routine is taken.

You can also specify the dummy application or DLL on the CVW command line.

To begin a DLL debugging session from the command line:

1. Type the command line:

```
CVW /Lmydll winstub
```

2. After CVW starts, do steps 3 – 6 in the previous procedure to begin debugging.

CVW Commands

CVW recognizes several commands for Windows-based program debugging in addition to the Command-window commands recognized by CV. These commands allow you to inspect objects in the global and local Windows heaps, list the currently loaded modules, trace and set breakpoints on the occurrence of Windows operating system messages, and terminate the currently executing task.

Windows Display Global Heap

The Windows Display Global Heap (**WDG**) command lists the memory objects in the Windows global heap.

Syntax

WDG [[*ghandle*]]

If *ghandle* is specified, **WDG** displays the first five global memory objects that start at the object specified by *ghandle*. The *ghandle* argument must be a valid handle to an object allocated on the global heap. If *ghandle* is not specified, **WDG** displays the entire global heap in the Command window.

Global memory objects are displayed in the order in which Windows manages them, which is typically not in ascending handle order. The output format is:

Format

handle address size **PDB** *locks type owner*

Any field may not be present if that field is not defined for the block.

Field	Description
handle	Value of the global memory block handle.
address	Address of the global memory block.
size	Size of the block in bytes.
PDB	Block owner. If present, indicates that the task's Process Descriptor Block is the owner of the block.
locks	Count of locks on the block.

type	The memory-block type.
owner	The block owner's module name.

Windows Display Local Heap

The Windows Display Local Heap (**WDL**) command displays the entire heap of local Windows operating system memory objects. This command's syntax takes no arguments.

Syntax

WDL

The output has the following format:

Format

handle address size flags locks type heaptype blocktype

Any field may not be present if that field is not defined for the block.

Field	Description
handle	Value of the global memory block handle
address	Address of the block
size	Size of the block in bytes
flags	The block's flags
locks	Count of locks on the block
type	The type of the handle.
heaptype	The type of heap where the block resides
blocktype	The block's type

Windows Display Modules

The Windows Display Modules (**WDM**) command displays a list of all the DLL and task modules that the Windows operating system has loaded. To see a list of known modules, type the **WDM** command in the Command window.

Syntax

WDM

Each entry in the list is displayed with the following format:

handle refcount module path

Field	Description
handle	The module handle
refcount	The number of times the module has been loaded
module	The name of the module
path	The path of the module's executable file

Watching Windows Operating System Messages

You can trace the occurrence of a Windows operating system message or an entire class of Windows operating system messages by using the Breakpoint Set (**BP**) command. You can stop at each message, or you can execute continuously and display the messages in the Command window as they are received.

To trace a Windows operating system message or message class, set a breakpoint using the following options:

Syntax

BP *winproc* **/M**{*msgname*|*msgclass*} [[**/D**]]

winproc
Symbol name or address of a window function.

msgname
The name of a Windows operating system message, such as WM_PAINT. The *msgname* is case sensitive.

msgclass
A case-insensitive string of characters that identifies one or more classes of messages to watch. Use the following characters to indicate the class of Windows operating system message:

Class	Type of Windows Message
m	Mouse
w	Window management
n	Input
s	System
i	Initialization
c	Clipboard
d	DDE
z	Nonclient

/D
When specified, CodeView displays the message in the command window, but your program continues executing. The message is displayed similar to the following example:

```
HWND:lc00 wParm:0000 lParm:000000 msg:000F WM_PAINT
```

For each matching message that is sent to the specified *winproc*, CVW lists the hexadecimal values of the window handle (`HWND`), word parameter (`wParm`), long parameter (`lParm`), and message (`msg`) arguments, along with the name of the message.

You can also specify a pass count and commands to be executed when the breakpoint is taken. For details on the full Breakpoint (**BP**) command syntax,

see “BP (Breakpoint Set)” on page 405. Note that you can also use the Breakpoint Set command from the Data menu to set all types of breakpoints.

Windows Kill Application

The Windows Kill Application (**WKA**) command terminates the currently executing task by simulating a fatal error. Since a fatal error terminates the application without performing any of the normal program exit processing, use **WKA** with caution.

To terminate your application, type the following command in the Command window:

Syntax

WKA

As a result of the simulated fatal error, Windows displays an Unexpected Application Error box. After you close the box, Windows may not release subsequent mouse input messages from the system queue until you press a key.

If this happens, the mouse pointer moves on the Windows screen but Windows does not respond to the mouse. After you press any key, Windows responds to the queued mouse events.

The currently executing task is not necessarily your application, so you should use the **WKA** command only when your application is the currently executing task. You can be sure that your application is the currently executing task when CVW shows the current location at a breakpoint in your application.

For more information on using the **WKA** command, see “Terminating Your Program” on page 362.

CVW Debugging Techniques

Debugging Windows-based programs can be challenging. Objects move around in memory. The thread of execution can be a twisting maze where it is difficult to know what code is executing or to control what code in your program is executed.

This section describes the **WLH** and **WGH** commands that you use to examine movable memory objects by their handles. It also describes ways to control your application’s execution, how to interrupt and resume debugging your application, how to handle abnormal termination from fatal errors and general protection faults, and how to resume debugging your application after a normal termination.

Dereferencing Memory Handles

In a Windows-based application, the **LocalLock** and **GlobalLock** routines are used to lock memory handles so that they can dereference them into near or far pointers.

In a debugging session, you may know the memory object's handle. However, you may not know what near or far address the handle references unless you are debugging in an area where the program has just completed a **LocalLock** or **GlobalLock** routine call. To get the near and far pointer addresses for unlocked local and global handles, use the **WLH** and **WGH** commands.

For detailed information on the **WLH** and **WGH** commands, see "WLH (Windows Dereference Local Handle)" on page 441 and "WGH (Windows Dereference Global Handle)" on page 439.

Controlling Application Execution

In CVW, all of the CodeView execution commands (Go, Program Step, Trace, and Animate) can be used to control your application's execution. However, you should keep these restrictions in mind while using CVW:

- Attempting to use the Program Step or Trace commands to execute Windows operating system startup code in Assembly mode causes unpredictable results. To step through your application in Assembly mode, first set a breakpoint at the **WinMain** routine and begin stepping through the program only after the breakpoint is taken.
- Directly calling a Windows-based application procedure or dialog routine in the Watch window, in the Quick Watch dialog box, or with the Display Expression (**?**) command can have unpredictable results.

The rest of this section describes techniques and special considerations for controlling program execution in CVW.

Interrupting Your Program

There may be times when you want to halt your program immediately. You can interrupt your program by pressing CTRL+ALT+SYSREQ. After you press CTRL+ALT+SYSREQ, CVW gains control and displays code corresponding to the current CS:IP location. You then have the opportunity to examine registers and memory, set breakpoints and watch expressions, and modify variables. To resume execution, use one of the CodeView program execution commands.

You should take care when you interrupt execution. If you interrupt execution while Windows operating system code or other system code is executing, attempting to use the Program Step or Trace commands can produce unpredictable results. When you interrupt execution, it is safest to set

breakpoints in your code and then resume continuous execution with the Go command, rather than using the Program Step, Trace, or Animate commands.

For example, an infinite loop in your code presents a special problem. Since you should avoid using the Program Step or Trace commands after interrupting your application, you should try to locate the loop by setting breakpoints in places you suspect are in the loop, then resume continuous execution. When one of these breakpoints is taken, you can be sure that the currently executing code is your application code.

Terminating Your Program

At times (such as when your application is executing an infinite loop), you may have to terminate the application. The Windows Kill Application (**WKA**) command terminates the currently executing task. Since this task is not necessarily your application, you should use the **WKA** command only when your application is the currently executing task.

If your application is the currently executing task and is executing a module containing CodeView information, the Source window highlights the current line or instruction. However, if your application contains modules that are compiled without CodeView information, it is more difficult to determine whether the assembly-language code displayed in the Source window belongs to your application or to another task.

In this case, use the Windows Display Global Heap (**WDG**) command with the value in the CS register as the argument. CVW displays a listing that indicates whether the code segment belongs to your application.

If the current code is in your application, you can safely use the **WKA** command without affecting other tasks. However, the **WKA** command does not perform all the cleanup tasks associated with the normal termination of a Windows-based application. For example, global objects created during program execution but not destroyed before you terminated the program remain allocated in the system-wide global heap. This reduces the amount of memory available during the rest of the Windows operating system session. For this reason, you should use the **WKA** command to terminate the application only if you cannot terminate it normally.

You should exercise caution when using the WKA command on an application that loads a DLL. If you terminate the application before it frees the DLL, the DLL remains loaded. If you rebuild the DLL and then run CVW again, the new version of the DLL doesn't get loaded.

Note The **WKA** command simulates a fatal error in your application, causing

the Windows operating system to display an Unexpected Application Error message box. After you close this message box, Windows may not release subsequent mouse input messages from the system queue until you press a key.

If this happens, the mouse pointer moves on the Windows operating system screen, but the Windows operating system does not respond to the mouse. After you press any key, the Windows operating system responds to the queued mouse events.

Handling Abnormal Termination of the Application

Your application can terminate abnormally in one of two ways while you are debugging it with CVW. It can cause a fatal exit, or it can cause a general protection fault. In both cases, CVW gains control, giving you the opportunity to examine the state of the system when your application terminated. CVW allows you to view registers, display the global and local heaps, display memory, and examine your source code.

Handling a General Protection Fault

If the abnormal termination is caused by a general protection fault (GPF) while executing your application code, CVW displays the line of code where the error occurred. Also, the Command window displays the following message:

```
Trap 13 (0DH) -- General Protection Fault.
```

If the general protection fault occurred while executing the Windows operating system code, the CVW Command window displays a stack trace that is useful for finding the error in your source code.

Restarting a Debugging Session

You can terminate your application without leaving CVW. The Windows operating system notifies CVW that it is terminating the application, and CVW displays the following message:

```
Program terminated normally (0)
```

The value in parentheses is the return value of the **WinMain** routine. This value is usually the *wParam* parameter of the WM_QUIT message, which in turn is the value of the *nExitCode* parameter passed to the **PostQuitMessage** routine.

You can then use the Go command to continue the debugging session for additional DLLs or applications. You can also restart the application by using the Restart command on the Run menu.

Debugging P-Code

Although MASM does not support p-code, certain Microsoft compilers can generate space-saving p-code instead of machine code. P-code cannot be run by the processor itself because it is not native machine code. However, when you compile a program into p-code, LINK and the Make P-Code (MPC) utility add an interpreter to your program that reads and interprets p-code instructions.

The interpreter implements a “stack machine.” The p-code instructions generally assume operands on the stack rather than take explicit registers or addresses. Because p-code instructions do not explicitly specify operands, they are extremely small. The trade-off for compact code is reduced execution speed. You use p-code when saving space is more important than speed.

CodeView allows you to debug p-code in the same way you debug native code. At the source level, debugging works the same way for p-code as it does for native code. With CodeView’s p-code execution model, you can view p-code instructions in Mixed and Assembly modes just as you view native machine instructions. The Register window displays the p-code registers and the top eight entries of the p-code stack. If your program contains both p-code routines and native routines, CodeView automatically switches between p-code display and native display. You can also force CodeView to stay in Native mode when you want to view the native machine code of the p-code interpreter itself.

The rest of this section describes:

- How to configure CodeView to use the p-code execution model.
- How to prepare p-code programs for debugging.
- Techniques for debugging p-code.
- Limitations while debugging p-code.

Requirements

To debug a program that contains p-code, make sure you set up CodeView with the p-code execution model. To do so, you will need a **Model** entry under the CodeView tag in TOOLS.INI.

The p-code execution model gives CodeView information about p-code instructions, addressing modes, registers, and so forth, which you need to debug p-code. With this execution model, you can debug p-code just as you can debug native machine code. Without the p-code execution model, you cannot view the source lines for p-code routines, unassemble p-code instructions, or view the p-code registers or stack. For information on the syntax of the **Model** entry, see page 305.

There is a dynamic-link library (DLL) for each p-code execution model, depending on the operating environment. The following list shows the filenames of the DLLs and the environment with which they run:

Filename	Description
NMD1PCD.DLL	Execution model for MS-DOS p-code
NMW0PCD.DLL	Execution model for the Windows operating system p-code

Specify the appropriate filename in the **Model** entry. For example, if you are debugging a Windows-based application that contains p-code, add an entry to the [CVW] section of TOOLS.INI such as:

```
Model:NMW0PCD.DLL
```

The exact syntax can vary, depending on your system configuration and other settings in TOOLS.INI.

Preparing Programs

To debug an application that contains p-code, you must first successfully compile, link, and run the MPC utility on the application. For information on how to build p-code applications and how to mix p-code with native machine code, see your high-level language documentation.

During compilation into p-code, the compiler saves space by using p-code quoting. P-code quoting reduces program size by minimizing repeated sequences of instructions. It replaces all but one of the sequences with a special quote instruction which calls the retained sequence.

Quoting makes debugging difficult because each routine jumps to other routines that contain the quoted instructions. When you compile a program for debugging, specify the /Of– option to turn quoting off. When you build a release version of the program, specify /Of to turn quoting back on so that the compiler can generate the smallest possible code.

By default, the compiler sorts local variables by frequency of use. It arranges them on the stack so that the program can access the most frequently used variables with the shortest instructions. This optimization is called frame sorting.

Frame sorting can make debugging more difficult because local variables do not appear on the stack in the order in which you declared them. You should turn off frame sorting by specifying the /Ov– option to the compiler. When you build a release version, specify /Ov to turn frame sorting on so that the compiler generates the smallest possible code.

P-Code Debugging Techniques

Debugging p-code is like debugging native machine code. If you are examing your program at the instruction level, you should be familiar with the machine's operation. With p-code, this is the stack machine implemented by the p-code interpreter.

For general information on the interpreter and p-code instructions, see your high-level language documentation. For information on the p-code instruction set, choose the P-Code Help button from the Microsoft Advisor's top-level contents. Help is available on each p-code instruction.

When you are debugging native code, you normally view two levels of execution: source code and machine code. P-code introduces another level between the two. You can debug at any of these levels by setting the right combination of Source, Mixed, Assembly, and Native display modes.

The next section shows how to choose the different levels and describes what happens when you trace between native and p-code.

The Native Command

The Native command from the Options menu toggles CodeView's display of native machine code and p-code. When Native mode is turned on, a bullet appears to the left of the command on the menu.

With Native mode turned on, CodeView displays native machine instructions in the Source and Mixed display modes. The Register window and the Register command show the native CPU registers.

With Native mode turned off, CodeView displays:

- Native machine instructions in those parts of your program that contain native code.
- P-code instructions in those parts of your program that contain p-code.

Also, the Register window and the Register command show the native CPU registers when debugging a native routine, and they display the p-code interpreter's registers when debugging a p-code routine.

The distinction between Native mode on and off becomes important when you trace from a native routine into a p-code routine or from a p-code routine to a native routine. Generally, you turn Native mode off to view p-code instructions. Turn Native mode on when you want to see the action of the p-code interpreter.

Tracing From Native Code to P-Code

With Native mode turned off, tracing into a p-code routine causes CodeView to display p-code instructions. You can animate, step, and trace each p-code

instruction in your program. You can also set breakpoints at individual p-code instructions. When tracing p-code, the Register window displays the registers and stack of the p-code machine.

With Native mode turned on, tracing into a p-code routine causes CodeView to display the native machine code of the p-code interpreter. Because the p-code interpreter is a library module that does not contain debugging information, CodeView switches to Assembly mode.

Tracing From P-Code to Native Code

With Native mode turned off, tracing from a p-code routine to a native routine causes CodeView to display native machine instructions. The Register window displays native CPU registers.

With Native mode turned on, you don't trace from p-code to native code. You trace out of the p-code interpreter and into your program's native code.

Unassembling P-Code

You can use the View Source and Unassemble commands to display p-code instructions in the Source window. With the View Source command, change to Mixed or Assembly display mode. The Unassemble command automatically displays p-code instructions when Native mode is turned off.

CodeView can display p-code and native code in the Source window at the same time. If you use the View Source or Unassemble commands in an area with both p-code and native code, CodeView displays both types of instructions. This commonly occurs when you view a routine with a native entry point as well as a p-code entry point. The different sections of code are separated by the assembly-language **Data** directive.

If you try to unassemble p-code with Native mode turned on, CodeView interprets p-code as native code and displays meaningless instructions.

P-Code Debugging Limitations

While CodeView makes debugging p-code as similar to debugging native machine code as possible, there are some limitations. The following list describes the commands that you cannot use with p-code:

- You cannot assemble p-code instructions.

  The Assemble command allows you to assemble instructions at any location in your program, but it accepts only native machine mnemonics. It does not recognize p-code mnemonics. If you accidentally overwrite p-code, use the Restart command. The Restart command restores your progam's code.

- You cannot call p-code functions.

With native code, you can use the Display Expression command to call any function. However, the Display Expression command cannot call p-code functions.

Remote Debugging

Microsoft CodeView versions 4.00 and later support remote debugging. This allows you to debug using two machines. CodeView runs on your development machine (the host), and the program you are debugging runs on another machine (the target). You run a remote monitor program on the target machine to control the program you are debugging. The monitor communicates with CodeView through a serial connection.

Remote debugging isolates CodeView from the program being debugged so that errors in the program do not affect the debugger, and the debugger does not affect the target system. If the program crashes the remote system, your development system continues to run.

The remote monitor demands fewer system resources than the full debugger and has fewer dependencies on the hardware and operating system. It does not use the display, the keyboard, extended memory, or expanded memory. After starting and loading the program to be debugged, it does not use the file system. Therefore, the monitor has no effect on these resources that can change your program's behavior.

You can debug large programs or programs that destabilize the operating system. You can also debug programs on older hardware or smaller systems such as laptops that cannot support the full debugger. Some bugs that you cannot reproduce while running under the full debugger appear under the remote monitor.

The process of debugging a program on a remote machine is almost the same as for local debugging. The only difference is in how you start the session. The following sections describe the hardware and files required for remote debugging and how to configure the debugger components on the host and target machines. Also included are the command-line syntax for the remote monitor and the steps you take to start a remote debugging session.

Requirements

Remote debugging requires two computers. The host system must support the Microsoft C/C++ development system. The target system needs only enough resources to run the remote monitor and your program. You run the MS-DOS CodeView on the host system, and you run either the MS-DOS remote monitor

or the Windows operating system remote monitor, according to the type of program you are debugging.

You connect the host and target machines with a null-modem cable plugged into the serial ports on the two machines. A null modem is a serial cable that connects the transmitting line at each end to the receiving line at the opposite end. For CodeView, you can tie all control lines to a TRUE signal. Note that such a cable may not be suitable for use with other software. You cannot use an extension cable with “straight-through” connections.

Any good computer store can assemble a null-modem cable for you with the correct wiring and the appropriate connectors for your host and target machines.

CodeView’s serial transport layers use interrupt-driven input and output, which is supported in MS-DOS only with the COM1 and COM2 ports. Therefore, your machines must be connected using the COM1 or COM2 ports. You can use different ports on the two machines.

If you plan to debug with two machines, you must have the correct files in the correct locations on the host and target. You can start a remote session with a TOOLS.INI file that configures CodeView for local debugging. However, it is recommended that you configure CodeView for remote debugging in TOOLS.INI.

MS-DOS Host Files

For remote debugging, you must have the CodeView debugger CV.EXE and its associated DLLs on the host machine. The SETUP program copies all the required files when you install the development system.

You configure CodeView for remote debugging by setting entries in the TOOLS.INI configuration file. The settings for CodeView appear in the `[[CV]]` tagged section of TOOLS.INI. Your settings should specify the DLLs for remote debugging. Most of the entries are the same for local and remote debugging. The only differences are the **Native** and **Transport** entries.

The remote debugging configuration is described in the following table:

Entry	Value	Description
Symbolhandler	SHD1.DLL	MS-DOS symbol handler.
Eval	EED1*lang*.DLL	Expression evaluator. You must load at least one expression evaluator. Use EED1CAN.DLL for C or MASM. Use EED1CXX.DLL for C++, C, or MASM.
Model	NMD1PCD.DLL	P-code execution model. To debug p-code, you must load the p-code nonnative execution model. Specify this entry only if you are debugging p-code.

Transport	TLD1COM.DLL	The serial transport layer. (For local debugging, use TLD1LOC.DLL.)
Native	EMD1D1.DLL EMD1W0.DLL	Execution model. The execution model that you use depends on the target. Use EMD1D1.DLL for MS-DOS targets or EMD1W0.DLL for Windows operating system targets.

For more information on configuring CodeView, see "Configuring CodeView with TOOLS.INI" on page 301.

You must have your program's executable file on both the host and target machines. The program must have the same path on the host and target machines, including drive letter and all directories. The filenames must be identical. For Windows-based applications, you must also have your application's DLLs (if any). The DLLs that you want to debug must also have the same path on the host and target machines.

MS-DOS Target Files

For remote debugging of an MS-DOS program, you need the MS-DOS remote monitor RCVCOM.EXE on the target machine along with your program's executable file. The program must have the same path on the host and target machines, including drive letter and all directories. The filenames must be identical.

You can set default parameters for the remote monitor in the `[[RCVCOM]]` section of a TOOLS.INI file on the target machine. For more information, see "Remote Monitor Settings in TOOLS.INI" on page 371.

Windows-Based Target Files

For remote debugging of a Windows-based application, you need the Windows operating system remote monitor RCVWCOM.EXE and its support DLLs on the target machine along with your application's executable files. The application and DLL files that you are debugging must have the same path on the host and target machines, including drive letter and all directories. The filenames must be identical.

The Windows operating system remote monitor (RCVWCOM.EXE) and its support DLLs (TOOLHELP.DLL and DMW0.DLL) must be in a directory listed in the PATH environment variable.

You can set default parameters for the remote monitor in the `[[RCVWCOM]]` section of a TOOLS.INI file on the target machine. For more information, see "Remote Monitor Settings in TOOLS.INI" on page 371.

Remote Monitor Command-Line Syntax

Syntax

{ **RCVCOM** | **RCVWCOM** } [[**/P** *port*:[[*rate*]]]] [[**/R**]]

Option	Description
RCVCOM	
RCVCOM	Remote monitor for MS-DOS.
RCVWCOM	Remote monitor for the Windows operating system.
/P	Parameters. The specified settings override any settings made in TOOLS.INI.
port:	Communications port. Must be `COM1:` or `COM2:`. The default setting is `COM1:`.
rate	Bit rate. Specifies the rate at which to drive the port, up to 19,200 bits per second (bps). To specify a rate, you must also specify a port. There can be no space between *port*: and *rate*. You must specify the same bit rate for the host and target. The default rate is 9600 bits per second. The possible rates are 50, 75, 110, 150, 300, 1200, 1600, 1800, 2000, 2400, 3600, 4800, 7200, 9600, and 19,200 bps.

Option	Description
/R	
/R	Resident. The monitor stays running when the host debugger exits. When /R is not specified, the monitor terminates when the host debugger exits.

Example

To start the remote monitor for several MS-DOS debugging sessions that use the COM2 port at 2400 bits per second, type the command:

```
RCVCOM /P COM1:2400 /R
```

Remote Monitor Settings in TOOLS.INI

You can set default parameters for the MS-DOS and the Windows operating system remote monitors in TOOLS.INI. If you do not specify parameters on the command line, the monitors look for TOOLS.INI in the directory specified by the INIT environment variable. You must place the settings for the remote monitors in tagged sections of TOOLS.INI. Settings for RCVCOM appear in the `[[RCVCOM]]` tagged section; settings for RCVWCOM appear in the `[[RCVWCOM]]` tagged section.

The remote monitors recognize a single **Parameters** entry. The syntax for this entry is:

Syntax **Parameters**: [[*port*:[[*rate*]]]]

You specify the *port* and *rate* as for the /P command-line option. The command-line option overrides the TOOLS.INI settings.

Starting a Remote Debugging Session

After the CodeView components are in their locations and properly configured, you can begin a remote debugging session.

To start a remote debugging session:

1. Transfer your program and its DLLs to the target machine.

   You can copy the files to a floppy disk, transfer them across a network, or transfer them across the serial line using communications software or serial file-transfer software.

   Make sure that the full path of the program on the target machine exactly matches the full path of the program on your host machine. The directory structures for your program's files on the host and target machines must also match exactly. If the paths of the files do not match, the remote monitor is unable to locate the program.

2. Start the remote monitor. If you are debugging a Windows-based application, double-click the Windows operating system remote monitor icon or use the Run command from the Program Manager File menu. For an MS-DOS program, start the monitor from the command line.

   The remote monitor starts and begins polling the communications port. It waits for the host debugger to initiate the debugging session.

3. Start CodeView on the host machine. How you start CodeView depends on your settings in TOOLS.INI.

   If you have configured CodeView for remote debugging in TOOLS.INI, you can specify a program on the CodeView command line or use the Load command on the Run menu to load the program. You have already configured the transport layer and execution model.

   If you are running the host-machine's CodeView as a non-Windows-based application, the application window must be full screen and exclusive in foreground. You handle this through the application's .PIF file. For more information on non-Windows-based application .PIF file settings, see your Windows operating system *User's Guide*.

   If you have configured CodeView for local debugging, you can start a remote session as described in the following section, "Starting CodeView for a Different Configuration."

CodeView starts, loads your program, and initiates communication with the remote monitor. You are now ready to debug.

Once the debugging session is started, you can use CodeView just as you would for a local debugging session. When you quit CodeView, the remote monitor quits (unless you specified /R when you started the monitor).

If your system has trouble maintaining the communications link between the host and target machines, reduce the bit rate.

Starting CodeView for a Different Configuration

If you have CodeView configured for local debugging in TOOLS.INI, start CodeView without specifying a program on the command line. This allows you to change CodeView's configuration before it loads your program. It is recommended that you configure CodeView for remote debugging.

To start a remote session from a local configuration:

1. Transfer your program and its DLLs to the target machine.
2. Start the remote monitor on the target machine.
3. On the host machine's command line, start CodeView with the following syntax:

   CV [[*options*]]

   Do not specify the program's filename or arguments.

   CodeView starts and displays the Load dialog box. Instead of specifying a program and its arguments, you must first reconfigure CodeView for remote debugging.
4. Choose Configure Remote. CodeView displays the Configure Remote dialog box. Load the remote transport layer and target execution model, as follows:
   a. Choose the TLD1COM.DLL transport layer.

      Select the communications port and bit rate for the session. Make sure that the bit rate is the same on the host and target machines.
   b. Choose the execution model for the appropriate target:
      - EMD1D1.DLL for debugging an MS-DOS program.
      - EMD1W0.DLL for debugging a Window-based application.
   c. Choose OK.

      CodeView returns to the Load dialog box.
5. Type the name of your program in the File to Debug text box, or select the name in the Files List box. Type your program's command-line arguments in the Arguments text box.
6. Choose OK.

CodeView starts, and initiates the remote session.

CHAPTER 11

Using Expressions in CodeView

The arguments to most CodeView commands are expressions. A source-level expression is a reference to a variable or a function call or one or more operations involving constants, variables, addresses, or function calls. A physical location is a register, a memory address or range, or a source-code line number that CodeView maps to an address.

To interpret expressions while maintaining its own programming language independence, CodeView uses dynamic-link libraries (DLLs) to look up symbols, parse, and evaluate expressions. These DLLs are called “expression evaluators.” This release of CodeView has two expression evaluators—one for source-level expressions in Microsoft and ANSI C and the other that handles C++. If you do not specify an expression evaluator, CodeView uses the C++ expression evaluator by default.

The C and C++ expression evaluators recognize most C operators and provide additional debugging operators that are not part of the languages. The C++ expression evaluator places certain restrictions on C++ expressions. Although there is no expression evaluator for the Microsoft Macro Assembler (MASM), the C and C++ expression evaluators support operators that simulate essential assembly-language operations. You use one of these expression evaluators when debugging MASM code.

Common Elements

When debugging, you use a few common elements in arguments to CodeView commands that are independent of the source language or the current expression evaluator. You often refer to line numbers in source files, and, less often, to lines in text files. You also specify registers and addresses. Some CodeView commands such as Memory Fill (**MF**) accept address ranges.

This section presents the ways to specify line numbers, refer to objects in memory, and use values stored in the processor registers. It also describes the syntax for memory ranges.

Line Numbers

Line numbers are useful for source-level debugging. In Source mode, you see a program displayed with each line numbered consecutively. CodeView allows you to use line numbers to specify the address of code generated for a line or to specify a certain line in a text file.

Syntax

[[*context*]] **.***linenumber*
[[*context*]] **@***linenumber*

The optional *context* is the context operator used to specify a certain file. When it is omitted, CodeView assumes that the line is in the current source file. The *linenumber* specifies the line in the file (numbered starting at 1). Some commands, such as the Breakpoint Set (**BP**) command, display an error message if the compiler does not generate code for the specified line. For more information on the context operator, see "The Context Operator" on page 382.

With most CodeView commands, the two forms are interchangeable because CodeView automatically maps between source lines and code addresses. The **.***linenumber* form specifies a line offset from the beginning of a file. Use this form with the View Source (**VS**) command to display any text file, including files that are not source files for the program you are debugging. The **@***linenumber* form specifies the address of the beginning of the code generated by the compiler for the specified line. Use this form with the Breakpoint Set (**BP**) command.

Examples

The following example uses the Go (**G**) command to execute the program from the current location up to line 100. Since no file is indicated, CodeView assumes the current source file.

```
>G @100
```

The following commands use the View Source (**VS**) command to display text files at specific lines as follows: line 10 of the current file, line 301 of EXAMPLE.CPP, and line 22 of TESTFILE.TXT.

```
>VS .10
>VS {,EXAMPLE.CPP}.301
>VS {,TESTFILE.TXT}.22
```

Registers

Syntax [[@]]*register*

The *register* is the name of a CPU or p-code register. You can specify a register name if you want to use the current value stored in the register. Registers are rarely needed in source-level debugging. However, they are frequently used for lower-level debugging.

When you specify an identifier, CodeView first checks the program's symbol table for the name. If the debugger does not find the name, it checks to see if the name is a register. If you want the identifier to name a register regardless of any name in the symbol table, use an at sign (@) before the register name.

For example, if your program has a symbol called `AX`, specify `@AX` to refer to the AX register. You can avoid this conflict by making sure that your program does not use register names as identifiers.

Table 11.1 lists the registers known to CodeView. The p-code registers are available when you are debugging p-code. The 32-bit registers are available on 80386 and 80486 machines when you turn the 386 option on.

Table 11.1 Registers

Register Type	Register Names
8-bit high byte	AH, BH, CH, DH
8-bit low byte	AL, BL, CL, DL
16-bit general purpose	AX, BX, CX, DX
16-bit segment	CS, DS, SS, ES
16-bit pointer	SP, BP, IP
16-bit index	SI, DI
16-bit high word*	TH
16-bit low word*	TL
Quoting*	PQ
32-bit general purpose †	EAX, EBX, ECX, EDX
32-bit pointer †	ESP, EBP
32-bit index †	ESI, EDI

* Available only when debugging p-code

† Available only when 386 option is turned on

Addresses

Syntax [[*context*]][[*symbol*]]

The *context* is the context operator and specifies the point at which to begin searching for *symbol*. If *context* is omitted, the current location is used. The *symbol* is a label, variable, or function name.

Syntax [[*context*]][[*@linenumber*]]

The *linenumber* is the number of a line in the specified file. If *context* is omitted, CodeView assumes the current file. Line numbers start at 1.

Syntax [[*segment*:]]*offset*

A full address is a *segment* and an *offset*, separated by a colon. The *segment* and *offset* can be numeric expressions, symbols, or register names. A partial address has only an *offset*; CodeView assumes a default segment address, depending on the command. Commands that refer to data (Memory Dump, Memory Enter, for example) assume the value of the data segment (DS) register. Commands that refer to code (such as Assemble, Breakpoint Set, and Go) assume the value of the code segment (CS) register.

In source-level debugging, full *segment*:*offset* addresses are seldom necessary. Occasionally they may be convenient for referring to addresses outside the program, such as display memory.

Examples

In the following example, the Memory Dump Bytes (**MDB**) command dumps memory starting at offset address `100`. Since no segment is given, the data segment (the default for Memory Dump commands) is assumed.

```
>MDB 100
```

In the following example, the **MDB** command dumps memory starting at the address of the element `array[[COUNT]]`.

```
>MDB array[[count]]
```

In the following example, the Unassemble (**U**) command shows a disassembly of memory starting at a point 10 bytes beyond the symbol `label`.

```
>U label+10
```

In this example, the **MDB** command dumps memory at the address having the segment value in the ES register and the offset address `200` in the current radix.

```
>MDB ES:200
```

Address Ranges

Syntax *start* [[*end*]]

A range is a pair of addresses that defines the boundary of a sequence of contiguous memory locations. You can specify a range by giving the starting address and the ending address. In this case, the range covers *start* to *end*, inclusively. If a command takes a range but you do not supply a second address, CodeView displays enough data to fill the current size of the window.

Syntax *start* **L** *length*

You can also specify a range by giving its starting point and the number of objects you want included in the range. This type of range is called a "length range." In a length range, *start* is the address of the first object, **L** indicates that this is a length range, and *length* specifies the number of objects in the range.

The size of the object is the size taken by the command. For example, the Memory Dump Bytes (**MDB**) command dumps bytes, the Memory Dump Words (**MDW**) command dumps words, the Unassemble (**U**) command unassembles instructions (which can vary in size), and so on.

Examples

The following example dumps a range of memory starting at the `buffer` symbol. Since the end of the range is not given, the default size is assumed (128 bytes for the Memory Dump Bytes (**MDB**) command in this example).

```
>MDB buffer
```

The following example dumps a range of memory starting at `buffer` and ending at `buffer+20` (the point 20 bytes beyond `buffer`).

```
>MDB buffer buffer+20
```

The following example uses a length range to dump the same range of memory as in the previous example.

```
>MDB buffer L 20
```

The following example uses the Memory Fill (**MF**) command to fill memory with dollar sign (**$**) characters starting 30 bytes before `right_half` and continuing to `right_half`.

```
>MF right_half-30 right_half '$'
```

Choosing an Expression Evaluator

CodeView loads all the expression evaluators that you specify with **Eval** entries in TOOLS.INI. However, you need to load only one expression evaluator for most debugging tasks. This section discusses how to choose the appropriate one for your debugging environment.

If you place more than one **Eval** setting in TOOLS.INI, CodeView loads all the expression evaluators. You can specify the active evaluator by using the Language command on the Options menu or with the **USE** command. By default, CodeView automatically selects the appropriate expression evaluator based on the current source file's extension. For more information on the Language command, see page 344. For details on the **USE** command, see page 430. For information on the **Eval** entry and complete instructions for configuring CodeView, see "Setting Up CodeView" on page 299.

When you are debugging C or MASM source code, you can normally use either the C or the C++ expression evaluator. C++ is mostly a superset of C at the expression level, and both evaluators support operators for debugging MASM code. Therefore, CodeView loads the C++ expression evaluator by default when no other expression evaluators are specified.

However, you might want to use only the C expression evaluator. If you are debugging C or MASM source code, it is recommended that you specify only the C expression evaluator. If your C program uses C++ keywords as variable, function, or label names, you must use the C expression evaluator. C variable names that are C++ keywords are not recognized as variables by the C++ expression evaluator. The C++ expression evaluator requires more memory than the C evaluator. Therefore, load only the C expression evaluator when running CodeView in an environment with limited memory.

You must use the C++ expression evaluator to debug C++ because the C evaluator does not recognize C++ expressions or keywords and cannot translate the decorated names produced by the C++ compiler. If you try to debug a C++ application with the C expression evaluator, C++ expressions generate an error, and you must use the decorated symbol names.

Using the C and C++ Expression Evaluators

When you specify the C or C++ expression evaluator, you can use most Microsoft or ANSI C and many Microsoft C++ expressions as arguments to CodeView commands.

CodeView evaluates C and C++ expressions according to the same rules as the compiler, including operator associativity and order of precedence. There are, however, a few additional operators and some exceptions to the standard syntax. See your high-level language documentation for descriptions of C and C++ expression syntax.

Additional Operators

Both expression evaluators support the following additional operators:

- The "context" operator (**{ }**) to specify the context of a symbol.
- The colon operator (**:**) to form addresses. The colon operator has the same precedence as the multiplication, division, and remainder operators.
- The memory operators (**BY**, **WO**, and **DW**) to access memory. Each of the memory operators has the same precedence, which is the lowest of any operator recognized by the expression evaluators.

The colon and memory operators are used mostly to debug assembly-language code. For information about the colon operator, see "Addresses" on page 378. The memory operators are described in "Memory Operators" on page 389. For more information about using the context operator, see "The Context Operator" on page 382.

Unsupported Operators

The comma operator (**,**) and the conditional (**?:**) operator are not supported by the C and C++ expression evaluators. The C++ operators **.*** and **->*** are also unsupported.

The ampersand (**&**) is not supported as a bitwise AND operator. However, both expression evaluators recognize the ampersand (**&**) as an address-of operator. The C++ expression evaluator also recognizes the ampersand in type casts to create a reference type. For example, `(int &)curIndex` casts the `curIndex` variable to an **int** reference type.

Restrictions and Special Considerations

When you are debugging C and C++ programs, the following general restrictions apply:

- When you use an expression as an argument to a CodeView command that takes multiple arguments, such as the Memory Fill (**MF**) command, the expression cannot contain spaces. For example, `&count+6` is allowed, but `&count + 6` is interpreted as three separate arguments. Some commands, such as the Display Expression (**?**) command, permit spaces in expressions.
- CodeView command names are not case sensitive, but C and C++ identifiers are case sensitive unless you turn off case sensitivity with the commands on the Options menu or the Options (**O**) command.
- You cannot call an intrinsic function or an inlined function in a CodeView expression unless it appears at least once as a normal function.
- CodeView limits casts of pointer types to one level of indirection. For example, `(char *)sym` is accepted, but `(char **)sym` is not. An expression such as `char far *( far *)` is also not supported.
- The C++ scope operator (**::**) has lower precedence in CodeView expressions than in the C++ language. In CodeView, its precedence falls between the base (**:>**) and postfix (++, —) operators and the unary operators (**!**, **&**, *****, for example). In C++, it has the highest precedence.

CodeView imposes additional restrictions on C++ expressions. These restrictions and other special considerations when debugging C++ are described in “Using C++ Expressions” on page 386.

The Context Operator

The context operator (**{ }**) is unique to CodeView. It is not part of the C or C++ languages. You use it to specify the exact context of an expression or line number that appears in more than one place in your source code. For example, you might use this operator to specify a symbol defined in an include file when the file is included more than once or to specify a name in an outer scope that is otherwise hidden by a local name.

When you use a symbol in a CodeView expression, the C and C++ expression evaluators search for that symbol in the following order:

1. Lexical scope outward. The expression evaluator searches for the symbol starting with the current block (a series of statements enclosed in curly braces) and continuing with the enclosing block. The current block is the code containing the current location (CS:IP address).
2. Function scope. The expression evaluator searches for the symbol in the current function.

3. Class scope. If debugging C++ and within the scope of a member function, the expression evaluator searches symbols of that member function's class and all its base classes. The C++ expression evaluator uses the normal dominance rules.
4. Current module. The expression evaluator searches all symbols in the current module.
5. Global symbols. The expression evaluator searches all global symbols in the program.
6. Other modules. The expression evaluator searches the global symbols in all other modules in the program.
7. Public symbols in the program.

If the name is not found in any of these places, and the name is not the name of a register, CodeView displays an error message.

The context operator lets you specify the starting point of the search and bypass the current location. Note that you cannot specify a block because a block has no name. You cannot specify a class, but you can specify a member function of the class and let the expression evaluator search outward.

Syntax

{[[*function*]]**,**[[*module*]]**,**[[*dllexe*]]**}** [[*object*]]

function
The name of a function in the program.

module
The name of a source file. You must specify a source file if the function is outside the current scope. If the file is not in the current directory, you must specify the path.

dllexe
The path of a program's DLL or .EXE file.

object
A line number or symbol.

The context operator has the same precedence and associativity as the type-cast operator. You can omit *function*, *module*, or *dllexe*, but you must specify all leading commas. You can omit trailing commas. If a name contains a comma, you must enclose the name in parentheses.

Example

The following example displays the value of the variable `Pos`, which is local to the function `make_box`, which is defined in the source file `DRAWBOX.C`. Assuming that there is more than one source file called `DRAWBOX.C`, the third parameter specifies that the source file containing the function `make_box` is the one used by `DISPTXT.DLL`.

```
? {make_box,C:\TREE1\DRAWBOX.C,C:\TREE2\DISPTXT.DLL}Pos
```

Numeric Constants

Numbers used in CodeView commands represent integer constants. They are expressed in octal, hexadecimal, or decimal radix; the default is the current radix. The default input radix for the C expression evaluator is decimal. However, you can use the Radix (**N**) command to specify a different radix, as explained on page 444. CodeView displays the current radix in the lower-right corner of the status bar.

To override the current radix, you can use the C and C++ syntax for entering a constant of a different radix. In addition, CodeView supports the **0n***digits* syntax to specify decimal numbers independently of the current radix.

The following table summarizes the syntax for different radixes:

Syntax	Radix
digits	Current radix
0*digits*	Octal (base 8)
0n*digits*	Decimal (base 10)
0x*digits*	Hexadecimal (base 16)

When hexadecimal is the current radix, it is possible to enter a value that could be either a symbol or a hexadecimal number. CodeView resolves the ambiguity by first searching for a symbol with that name. If no symbol is found, the value is a number. If you want to enter a number that is the same as a symbol in your program, use the explicit hexadecimal format (**0x***digits*).

For example, if the program contains a variable named `abc` and you enter `abc`, CodeView interprets the argument as the symbol. If you want to enter it as a number, enter it as `0xabc`.

String Literals

Syntax "*string*"

Strings can be specified as expressions in the C format. You can use all ANSI C escape sequences within strings. For example, double quotation marks within a string are specified with the escape sequence `\"`.

A string that you specify in a CodeView command is volatile, and you cannot rely on its existence for longer than the execution of the command. This means that you can pass a string to a function, but you cannot assign a string to a character pointer variable. For example, the command:

```
? pChar = "string"
```

is not valid. However, you can change a pointer to refer to a different string in your program. Also, if the pointer addresses a section of memory large enough to accommodate the string, you can use the Memory Enter (**ME**) command to fill the memory with a new string.

Symbol Formats

For modules that are compiled with full CodeView debugging information (/Zi), the expression evaluators automatically translate the decorated names into source form. You specify and view names as they appear in your source. Therefore, debugging is easier when all modules in the program are compiled with full CodeView debugging information. For large programs, however, you may need to compile some modules to include only line numbers and public symbols (/Zd).

CodeView accepts and displays public symbol names as “decorated” names. The decorated name is the form of the name in the object code produced by the compiler. Public symbols are names in library routines or names in modules compiled without CodeView information (that is, compiled with the /Zd option, or compiled without any line or symbolic information and linked with the /CO option).

To get a listing of all names in their decorated and undecorated forms, specify the LINK /MAP:FULL option.

Name decoration is the mechanism used to enforce type-safe linkage. This means that only the names and references with precisely matching spelling, case, calling convention, and type are linked together.

Names declared with the C calling convention (either implicitly or explicitly using the **_cdecl** keyword) begin with an underscore (_). For example, the function `main` can be displayed as `_main`. Pascal names are converted to uppercase and

have no prefix. Names declared as **_fastcall** are converted to uppercase and begin with an at sign (**@**).

For C++, the decorated name encodes the type of the symbol in addition to the calling convention. This form of the name can be long and difficult to read. The name begins with at least one question mark (**?**). For C++ functions, the decoration includes the function's scope, the types of the function's parameters, and the function's return type.

Using C++ Expressions

The C++ expression evaluator accepts almost all C++ expressions, with some restrictions and some additions. This section describes these special considerations.

Access Control

You can examine any member of a class object including base classes and embedded member objects. In CodeView, all members are available without regard to access control (public, protected, or private visibility). For example, if `myDate`
has a private data member named `month`, you can examine it with the following command:

```
>? myDate.month
3
```

Ambiguous References

If an expression makes an ambiguous reference to a member name, you must use the class name to qualify it. For example, suppose that class C inherits from both class A and class B, and that A and B define a member function named `expand`. If `Cthing` is an instance of class C, the following expression is ambiguous:

```
Cthing.expand()
```

The following expression resolves the ambiguity and uses B’s `expand` function:

```
Cthing.B::expand()
```

The C++ expression evaluator applies normal dominance rules regarding member names to resolve ambiguities.

Inheritance

When you display a class object that has virtual base classes, the members of the virtual base class are displayed for each inheritance path, even though only one instance of those members is stored. You can access members of an object through a pointer to the object, and you can call virtual functions through a pointer.

For example, when the `Employee` class defines a virtual function that is named `computePay`, which is redefined in the class that inherits from `Employee`, you can call `computePay` through a pointer to `Employee` and have the proper function executed:

```
>? empPtr->computePay()
```

You can cast a pointer to a derived class object into a pointer to a base class object; the reverse conversion is not permitted. For example, if the class `List` is derived from the class `Collection`, the cast `(Collection *)pListCustomer` is valid, but the cast `(List *)pCollectClients` is illegal.

Constructors, Destructors, and Conversions

You can set a breakpoint on a class's constructor or a destructor (unless they are inline functions). The breakpoint is taken whenever an object of that class is created or destroyed. You can specify a breakpoint that halts execution so that you can examine your program's status. You can also specify a breakpoint that executes a command, such as displaying a message in the Command window or incrementing a counter, and then continues execution. This technique is especially useful for monitoring the creation and destruction of temporary objects created by the compiler.

You cannot call a constructor or destructor for an object, either explicitly or implicitly, by using an expression that calls for construction of a temporary object. For example, the following illegal command explicitly calls a constructor and results in an error message:

```
>? Date( 2, 3, 1985 )
```

You cannot call a conversion function if the destination of the conversion is a class because such a conversion involves the construction of an object. For example, suppose that `myFraction` is an instance of the `Fraction` class, which defines the conversion function `operator FixedPoint`. The following command results in an error:

```
>? (FixedPoint)myFraction
```

However, you can call a conversion function if the destination of the conversion is a built-in type. For example, suppose that the `Fraction` class defines a conversion function named `operator float`. The following command is legal:

```
>? (float)myFraction
```

You can also call functions that return an object or that declare local objects.

You cannot call the **new** or **delete** operators. The command

```
? pDate = new Date(2,3,1985)
```

is illegal and CodeView displays an error message.

Overloading

You can call overloaded functions as long as there exists an exact match or a match that does not require a conversion involving the construction of an object. For example, if the `calc` function takes a `Fraction` object as a parameter, and the `Fraction` class defines a single-argument constructor that accepts an integer, the following command results in an error:

```
>? calc( 23 )
```

Even though a legal conversion exists to convert the integer into the `Fraction` object that the `calc` function expects, such a conversion involves the creation of an object and is not supported.

Operator Functions

Operator functions for a class can be invoked implicitly or explicitly. For example, suppose that `myFraction` and `yourFraction` are instances of a class that defines **operator+**. You can display the sum of those two objects using expression syntax:

```
>? myFraction + yourFraction
```

You can also use the functional notation to call an operator function:

```
>? myFraction.operator+( yourFraction )
```

If an operator function is defined as a friend, you can call it implicitly using the same syntax as for a member function, or you can invoke it explicitly, as follows:

```
>? operator+( myFraction, yourFraction )
```

Note that operator functions, like ordinary functions, cannot be called with arguments that require a conversion involving the construction of an object.

Debugging Assembly Language

MASM versions 5.0 and later provide type and size information for CodeView. With this information, CodeView can correctly evaluate expressions derived from assembly code (except for arrays, which require a different syntax, as discussed later in this section).

CodeView does not have an assembly-language expression evaluator. If you are using Microsoft C/C++, you can use either the C or C++ expression evaluators for debugging assembly language The C and C++ expression evaluators provide special operators to simulate essential MASM operations.

You cannot always specify an expression in CodeView exactly as it would appear in assembly-language source code. You have to write an equivalent CodeView expression. This section describes the CodeView equivalents for MASM expressions.

Memory Operators

A memory operator is a unary operator that returns the result of a direct memory operation. The memory operators are **BY**, **WO**, and **DW**. The C and C++ expression evaluators add the memory operators to the operators in the C and C++ languages. The memory operators are used mainly to debug assembly-language code.

Syntax

{**BY** | **WO** | **DW**} *address*

The **BY** operator returns a short integer that contains the first byte at *address*. This operator simulates **BYTE PTR**.

The **WO** operator returns a short integer that contains the value of the word (two bytes) at *address*. This operator simulates the Microsoft Macro Assembler **WORD PTR** operation. The **DW** operator returns a long integer that contains the value of the first four bytes at *address*. This operator simulates **DWORD PTR**.

The examples that follow use the Display Expression (**?**) command, which is described on page 452. The x format specifier used in some of these examples causes the result to be displayed in hexadecimal.

Examples

The following example displays the first byte at the address of the variable `sum`.

```
>? BY sum
101
```

The following example displays the byte pointed to by the BP register with a displacement of 6.

```
>? BY bp+6,x
0042
```

The following example displays the first word at the address of the variable `new_set`.

```
>? WO new_set
13120
```

The following example displays the word pointed to by the stack pointer (the last word pushed onto the stack). Because the stack pointer (SP) offset register is used with no segment address, the stack segment (SS) register is assumed.

```
>? WO sp,x
2F38
```

The following example displays the doubleword at the address of `sum`.

```
>? DW sum
132120365
```

The following example displays the doubleword pointed to by the SI register. Because the SI index register is used without specifying a segment address, the DS register is assumed.

```
>? DW si,x
3F880000
```

Register Indirection

The C expression evaluator does not recognize brackets (**[ ]**) to indicate a memory location pointed to by a register. Instead, you use the **BY**, **WO**, and **DW** operators to reference the corresponding byte, word, or doubleword values.

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**, and **DW** operators 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 C address-of operator (**&**) instead of the MASM **OFFSET** operator.

MASM Expression	CodeView Equivalent
`OFFSET Var`	`&Var`

PTR Operator

Use type casts or the **BY**, **WO**, and **DW** operators with the address-of operator (**&**) to replace the assembly-language **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 the string format specifier `,s` after the variable name.

MASM Expression	CodeView Equivalent
`String`	`String,s`

Because C strings end with a null (ASCII 0) character, CodeView displays all characters from the first byte of the variable up to the next null byte in memory when you request a string display. If you intend to debug an assembly-language program, and you want to view strings in the Watch window or with the Display Expression (**?**) command, you should delimit string variables with a null character. You can also view null-terminated or unterminated strings in a Memory window or with the Memory Dump ASCII (**MDA**) command.

Array and Structure Elements

Prefix an array name with the address-of operator (**&**) and add the desired offset. The offset can be an expression, number, register name, or variable.

The following examples (using byte, word, and doubleword arrays) show how to do this.

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)`

CHAPTER 12

CodeView Reference

This chapter describes the CodeView Command-window command format, explains the common items in CodeView expressions, and summarizes all Command-window commands in a convenient table. The final section describes each command in detail. The nonalphabetic commands appear at the end of the chapter.

Command-Window Command Format

Syntax

command [[*arguments*]] [[**;** *command* [[*arguments*]]]]

Parameters

command
A command name. The *command* is not case sensitive; any combination of uppercase and lowercase letters can be used.

arguments
Expressions that represent values or addresses used by the command. Source-level expressions used as arguments may or may not be case sensitive, depending on the current expression evaluator. The first argument can be placed immediately after *command* with no space separating the two fields.

If a command takes more than one argument, you must separate the arguments with spaces.

Remarks

Additional commands may be specified on the same line. A semicolon (**;**) must separate each command from the next.

CodeView Expression Reference

When debugging, you use a few common elements in arguments to CodeView commands that are independent of the source language or the current expression evaluator. You often refer to line numbers in source files and, less often, to lines

in text files. You also specify registers and addresses. Some CodeView commands such as Memory Fill (**MF**) accept address ranges.

This section presents the ways to specify line numbers, refer to objects in memory, and use values stored in the processor registers. It also describes the syntax for memory ranges. Moreover, the context operator, which you use to specify the point at which to begin searching for a symbol, is summarized. For detailed information on the context operator and CodeView expressions, see Chapter 11.

Line Numbers

Syntax

[[*context*]]*@linenumber*
[[*context*]]**.***linenumber*

Description

Line numbers are useful for source-level debugging. They correspond to the lines in source-code files. In source mode, a program is displayed with each line numbered sequentially. The CodeView debugger allows you to use these numbers to access parts of a program.

The memory address of the code corresponding to a source-line number is specified as:

@linenumber

The actual file line number is:

[[*context*]]**.***linenumber*

CodeView assumes that the source line is in the current source file. To specify a source line in a different file, you must specify the line's context using the context operator:

{,*file***}***@linenumber*

CodeView displays an error message if *file* does not exist or no source line exists for *linenumber*.

Examples

The following example uses the View Source (**VS**) command to display code starting at source line 100. Since no file is indicated, the current source file is assumed.

```
>VS @100
```

This next example uses **VS** to display source code starting at line 301 of the file DEMO.C.

```
>VS {,demo.c}.301
```

Registers

Syntax [[@]]*register*

A register name represents the current value stored in the register. Table 12.1 summarizes the register names known to the CodeView debugger.

Table 12.1 Register Names

Register Type	Register Names
8-bit high byte	AH, BH, CH, DH
8-bit low byte	AL, BL, CL, DL
16-bit general purpose	AX, BX, CX, DX
16-bit segment	CS, DS, SS, ES
16-bit pointer	SP, BP, IP
16-bit index	SI, DI
16-bit high word*	TH
16-bit low word*	TL
Quoting*	PQ
32-bit general purpose†	EAX, EBX, ECX, EDX
32-bit pointer†	ESP, EBP
32-bit index†	ESI, EDI

*Available only when debugging p-code

† Available only when 386 option is turned on

To force a symbol to represent a register, prefix the symbol with an at sign (@). For example, to make `AX` represent a register rather than a variable, use `@AX`.

Addresses

Syntax

[[*context*]]*@linenumber*
[[*context*]][[*segment*:]]*offset*
register:*offset*
[[*context*]]*function*
[[*context*]]*symbol*

Description

If only an *offset* is specified, the *segment* is determined by the command in which the address appears. Commands that refer to data (Memory Dump, Memory Enter) use the segment in the DS register. Commands that refer to code (Assemble, Breakpoint Set, Go, Unassemble, and View Source) use the segment in the CS register.

The Display Expression (**?**) and Add Watch Expression (**W?**) commands interpret numeric arguments as constants rather than as offsets. However, if you cast the argument to a pointer, as in

```
W? (char *)0
```

the argument is treated as an offset from DS.

Address Ranges

Syntax

start end
start **L** *count*

Description

An address range is a pair of memory addresses that specify the higher and lower boundaries of contiguous memory. You can specify a range in two ways:

- Give the starting and ending addresses:

  start end

  The range covers *start* to *end*, inclusively. If you don't supply an ending address, CodeView assumes the default range. Each command has its own default range; the most common default range is 128 bytes.

- Give the starting address and the number of objects you want included in the range:

  start **L** *count*

This type of range is called an "object range." The starting address is the address of the first object in the list, and *count* specifies the number of objects in the range. The way the size of an object is measured depends on the command. For example, the Memory Dump Bytes (**MDB**) command has byte objects, the Memory Dump Words (**MDW**) command has words, the Unassemble (**U**) command has instructions, and so on.

Examples

This example dumps a range of memory starting at the symbol `buffer`. Since the end of the range is not given, the default size (128 bytes for the Memory Dump Bytes command) is assumed.

```
MDB buffer
```

The following example dumps 21 bytes starting at `buffer` and ending at `buffer+20` (the point 20 bytes beyond `buffer`).

```
MDB buffer buffer+20
```

The following example uses an object range to dump a range of memory. The L indicates that the range is an object range, and 20 indicates the number of objects in the range.

```
MDB buffer L 20
```

Here, each object has a size of 1 byte since that is the size of objects dumped by the Memory Dump Bytes (**MDB**) command.

Context Operator ({})

Syntax

{ [[*function*]] **,** [[*module*]] **,** [[*dllexe*]] **}** [[*object*]]

Parameters

function
The name of a function or procedure in the program. Case is significant for case-sensitive languages.

module
The name of a source file. If the file is not in the current directory, you must specify the path.

dllexe
The full path of a dynamic-link library (DLL) in the program or the program's .EXE file.

object
A variable name, line number, or expression.

Description

The context operator specifies the exact starting point to search for a symbol or line. You apply it the same way as a type cast is applied in C. When you do not use the context operator, the current context (CS:IP) is used.

You can omit *function*, *module*, or *dll*, but all leading commas must be given. Trailing commas can be omitted. If a name contains a comma, the name must be enclosed in parentheses.

For complete information on the context operator, see "The Context Operator" on page 382.

Example

This example displays the value of the variable `Pos`, which is local to the function `make_box` defined in the source file BOXDRAW.C.

```
? {make_box,C:\PROJ\boxdraw.c}Pos
```

CodeView Command Overview

Table 12.2 summarizes the CodeView Command-window commands. The next section describes each command in detail.

Table 12.2 CodeView Command Summary

Command	Name	Description
A	Assemble	Inserts assembly-language instructions
BC	Breakpoint Clear	Clears one or more breakpoints
BD	Breakpoint Disable	Disables one or more breakpoints
BE	Breakpoint Enable	Enables one or more breakpoints
BL	Breakpoint List	Lists all breakpoints
BP	Breakpoint Set	Sets a breakpoint
E	Animate	Executes the program in slow motion
G	Go	Executes the program
H	Help	Provides Help information
I	Port Input	Reads a byte from a hardware port
K	Stack Trace	Displays active routines K command
L	Restart	Restarts the program
MC	Memory Compare	Compares two blocks of memory byte by byte
MD	Memory Dump	Displays sections of memory in the Command window in various formats
ME	Memory Enter	Modifies memory
MF	Memory Fill	Fills a block of memory

Table 12.2 CodeView Command Summary (*continued*)

Command	Name	Description
MM	Memory Move	Copies one block of memory to another
MS	Memory Search	Scans memory for specified byte values
N	Radix	Changes current radix for entering arguments and displaying values
O	Options	Views or sets options
O	Port Output	Outputs a byte to a hardware port
P	Program Step	Executes the current line and steps over functions
Q	Quit	Terminates CodeView
R	Register	Displays the values of registers and flags and optionally changes them
T	Trace	Executes the current line and traces into functions
T	Trace Speed	Specifies speed for the Animate command
U	Unassemble	Displays assembly-language instructions
USE	Use Language	Specifies the active expression evaluator
VM	View Memory	Displays sections of memory in a Memory window in various formats
VS	View Source	Displays source code in varying formats in a Source window
W?	Add Watch	Sets an expression to be watched
WC	Delete Watch	Deletes one or more watch expressions
WDG	Windows Display Global Heap	Displays memory objects in the global heap
WDL	Windows Display Local Heap	Displays memory objects in the local heap
WDM	Windows Display Modules	Displays a list of the applications and DLL modules known by Windows
WGH	Windows Dereference Global Handle	Dereference a global handle
WKA	Windows Kill Application	Terminates the current task by simulating a fatal error
WL	List Watch	Lists current watch expressions
WLH	Windows Dereference Local Handle	Dereference a local handle
X	Examine Symbols	Displays the addresses and types of symbols
!	Shell Escape	Runs an MS-DOS shell

Table 12.2 CodeView Command Summary (*continued*)

Command	Name	Description
“	Pause	Interrupts execution of redirected commands and waits for keystroke
#	Tab Set	Sets number of spaces for each tab character
*****	Comment	Displays explanatory text during redirection
.	Current Location	Displays the current location
/	Search	Searches for a regular expression in the source
7	8087	Shows the values of the 8087 or emulator registers and flags
:	Delay	Delays execution of redirected commands
<	Redirect Input	Reads input from specified device
>	Redirect Output	Sends output to specified device
=	Redirect Input and Output	Sends output and reads input from specified device
?	Display Expression	Evaluates and displays expressions or symbols
??	Quick Watch	Displays variables and data structures in a dialog box
@	Redraw	Redraws the screen
****	Screen Exchange	Exchanges the CodeView and output screens

CodeView Command Reference

The rest of this chapter is an alphabetical reference to all CodeView Command-window commands. Nonalphabetic commands such as the Pause (**”**) command are listed after the alphabetic reference.

A (Assemble)

Syntax **A** [[*address*]]

Parameter *address*

Begins assembly at this address. If *address* is not given, assembly begins at the current assembly address (see following).

Description

The Assemble (**A**) command assembles 8086-family (8086/87/88, 80186/286, 80287/387, and 80286/386/486 unprotected) instruction mnemonics and places the resulting instruction code into memory at a specified address. The only 8086-family mnemonics that cannot be assembled are 80386/486 protected-mode mnemonics. In addition, the CodeView debugger can assemble 80286 instructions that use the 32-bit 386/486 registers.

If *address* is specified, the assembly starts at that address; otherwise, the current assembly address is assumed.

The assembly address is normally the current address or the address pointed to by CS:IP. However, when you use the Assemble command, the assembly address is set to the address immediately following the last assembled instruction.

When you enter any command that executes code (Trace, Program Step, Go, or Animate), the assembly address is reset to the current address.

Entering Instructions

Use the following procedure to assemble instructions:

1. Type the Assemble (**A**) command in the command window and press ENTER. CodeView displays the assembly address and waits for you to enter a new instruction.
2. Type a mnemonic and press ENTER. CodeView assembles the instruction into memory and displays the next available address. If an instruction you enter contains a syntax error, CodeView displays the message:

   ```
   ^ Syntax error
   ```

   Then CodeView redisplays the current assembly address and waits for you to enter a correct instruction. The caret (^) in the message points to the first character that CodeView could not interpret.
3. Continue entering new instructions until you have assembled all the instructions you want.
4. Press ENTER without entering any mnemonic to conclude assembly and return to the CodeView prompt.

Remarks

Consider the following principles when you enter instruction mnemonics:

- The far-return mnemonic is RETF.
- String mnemonics must explicitly state the string size. For example, MOVSW must be used to move word strings and MOVSB must be used to move byte strings.
- CodeView automatically assembles short, near, or far jumps and calls, depending on byte displacement to the destination address. These may be overridden with the NEAR or FAR prefix. The NEAR prefix can be abbreviated to NE, but the FAR prefix cannot be abbreviated.

- CodeView cannot determine whether some operands refer to a word memory location or to a byte memory location. In these cases, the data type must be explicitly stated with the prefix **WORD PTR** or **BYTE PTR**. Acceptable abbreviations are **WO** and **BY**.
- CodeView cannot determine whether an operand refers to a memory location or to an immediate operand. CodeView uses the convention that operands enclosed in brackets (**[ ]**) refer to memory.
- CodeView supports all forms of indirect register instructions.
- All instruction-name synonyms are supported. If you assemble instructions and then examine them with the Unassemble (**U**) command, CodeView may show synonymous instructions, rather than the ones you have assembled.
- Do not assemble and execute 8087/287 instructions if your system is not equipped with a math coprocessor chip.

The effects of the Assemble command are temporary. Any instructions that you assemble are lost as soon as you exit the program.

The instructions you assemble are also lost when you restart the program with the Restart command. The original code is reloaded, possibly writing over parts of memory that you have changed.

Example

This example places two new instructions in a program, replacing any instructions already there.

```
>a 0x47:0xb3e
0001:0B3E  mov ax,bx
0001:0B40  mov si,0x9ce
0001:0043
```

You can modify a portion of code for testing, as in the example, but you cannot save the modified program. You must modify your source code and recompile.

BC (Breakpoint Clear)

Syntax

BC [*list* | *start-end* | *****]

Parameters

list
List of breakpoints to be removed, with breakpoint numbers separated by spaces. A number identifies each breakpoint. You can use the Breakpoint List (**BL**) command to display currently set breakpoints and their numbers.

start-end
Range of breakpoints to clear. The command clears breakpoints numbered from *start* to *end*, inclusive.

*****

Removes all currently set breakpoints.

Description

The Breakpoint Clear (**BC**) command permanently removes one or more previously set breakpoints.

Mouse and Keyboard

In addition to typing the **BC** command, you can clear breakpoints with the following shortcuts:

- From the Data menu, choose Edit Breakpoints.
- Double-click the line containing the breakpoint.
- Using the keyboard, move to the line containing the breakpoint, and press F9.

Examples

The following example removes breakpoints 0, 4, and 8:

```
>BC 0 4 8
```

The following example removes all breakpoints:

```
>BC *
```

The following example removes breakpoints 4, 5, 6, and 7:

```
>BC 4-7
```

BD (Breakpoint Disable)

Syntax

BD [[*list* | *start-end* | *****]]

Parameters

list

List of breakpoints to be disabled, with breakpoint numbers separated by spaces. A number identifies each breakpoint. You can use the Breakpoint List (**BL**) command to display currently set breakpoints and their numbers.

start-end

Range of breakpoints to disable. The command disables breakpoints numbered from *start* to *end*, inclusive.

*****

Disables all currently set breakpoints.

Description

The Breakpoint Disable (**BD**) command temporarily disables one or more existing breakpoints. The breakpoints are not deleted; they can be restored at any time using the Breakpoint Enable (**BE**) command.

A disabled breakpoint can be cleared using the Breakpoint Clear (**BC**) command.

In the Source window, enabled breakpoints are highlighted. However, the highlighting disappears once the breakpoint is disabled.

Mouse and Keyboard

As an alternative to typing the **BD** command, choose Edit Breakpoints from the Data menu. There is no keyboard shortcut.

Examples

The following example temporarily disables breakpoints 0, 4, and 8:

```
>BD 0 4 8
```

The following example temporarily disables all breakpoints:

```
>BD *
```

The following example disables breakpoints 4, 5, 6, and 7:

```
>BD 4-7
```

BE (Breakpoint Enable)

Syntax

BE [[*list* | *start-end* | ***]]

Parameters

list
List of breakpoints to be enabled, with breakpoint numbers separated by spaces. A number identifies each breakpoint. You can use the Breakpoint List (**BL**) command to display currently set breakpoints and their numbers.

start-end
Range of breakpoints to enable. The command enables breakpoints numbered from *start* to *end*, inclusive.

***
Enables all currently disabled breakpoints.

Description

The Breakpoint Enable (**BE**) command enables breakpoints that have been temporarily disabled with the Breakpoint Disable (**BD**) command.

Mouse and Keyboard

In addition to typing the **BE** command, you can also enable breakpoints from the Data menu by choosing Edit Breakpoints. There is no keyboard shortcut.

Examples

The following example reenables breakpoints 0, 4, and 8:

```
>BE 0 4 8
```

The following example enables all disabled breakpoints:

```
>BE *
```

The following example enables breakpoints 4, 5, 6, and 7:

```
>BE 4-7
```

BL (Breakpoint List)

Syntax

BL

Description

The Breakpoint List (**BL**) command lists current information about all breakpoints.

For each breakpoint, the command displays the following:

- The breakpoint number.
- The breakpoint status, where “E” is for enabled, “D” is for disabled, and “V” is for “virtual.” A virtual breakpoint is a breakpoint set in an overlay or a DLL that is not currently loaded. A virtual breakpoint may be enabled or disabled.
- The address, function, file, and line number where the breakpoint is set.
- The expression, pass count, and break commands, if set.

Mouse and Keyboard

In addition to typing the **BL** command, you can also list breakpoints from the Data menu by choosing Edit Breakpoints. There is no keyboard shortcut.

BP (Breakpoint Set)

Syntax

BP [[*address*]] [[[[**=***expression* [[**/R***range*]]]] | [[**?***expression*]]]] [[**/P***passcount*]] [[**/C"***commands***"**]] [[**/M***msgname*|*msgclass* [[**/D**]]]]

Parameters

address
An expression giving the address at which to set the breakpoint. If omitted, sets a breakpoint on the current line, unless **=***expression* is also specified.

expression
Breaks program execution when the value of *expression* changes. If *address* is given, the expression is checked for changes only at that address. The expression is usually the name of a variable.

/R*range*
Watches all addresses in the given range for changes. The range is determined by multiplying *range* with the size of *expression*.

?*expression*
Breaks program execution when *expression* becomes true (nonzero). If *address* is given, the breakpoint *expression* is evaluated only at that address. You cannot specify both *=expression* and **?***expression* in the same breakpoint. Also, you cannot have more than one local context in *expression*. If the *expression* contains spaces, it must be enclosed in double quotation marks ("*expression*").

/P*passcount*
Specifies the first time the breakpoint is to be taken. For example, if *passcount* is 5, the breakpoint will be ignored the first four times it is encountered and taken the fifth time. From that point on, the breakpoint is always taken until the program is restarted.

/C"*commands*"
A list of command-window commands to be executed when the breakpoint is encountered. The *commands* must be enclosed in double quotation marks (") and separated by semicolons (**;**).

/M*msgname*
(CVW only) Breaks program execution whenever the specified message is received. When /D is also specified, the message received is displayed, but the breakpoint is not taken.

/M*msgclass*
(CVW only) Breaks program execution whenever a message belonging to one of the specified classes is received. When /D is also specified, the message received is displayed but the breakpoint is not taken. Can be one or more of the following:

Message Class	Type of Windows Message
m	Mouse
w	Window management
n	Input
s	System
i	Initialization
c	Clipboard
d	DDE

z	Nonclient

Description

The Breakpoint Set (**BP**) command creates a breakpoint at a specified address. Whenever a breakpoint is encountered during program execution, the program halts and waits for a new command.

You can set breakpoints at source lines, functions, explicit addresses, or labels in any module of a program. If no arguments are given, **BP** sets a breakpoint at the current line.

Windows Breakpoints

In CodeView for Windows (CVW), use of the /M options requires that *address* be the name or address of a window function ("winproc").

When the /D option is specified, CVW displays each message in the Command window as it is sent to the application's window function. The message is displayed in the following format:

HWND:*wh* wParm:*wp* lParm:*lp* msg:*msgnum msgname*

where *wh* is the window handle, *wp* is the message's word-sized parameter, *lp* is the message's long-sized parameter, *msgnum* is is the message number, and *msgname* is the name of the message. The following is a typical display:

```
HWND:lc00 wParm:0000 lParm:000000 msg:000F WM_PAINT
```

The Windows operating system breakpoints appear in the list of breakpoints and may be enabled, disabled, and cleared with the usual CodeView breakpoint commands.

Breakpoint Options

For any breakpoint, you can also specify:

- A pass count to tell CodeView how many times to pass over the breakpoint.
- Commands to be executed after the program reaches the breakpoint.

Breakpoints are numbered, beginning with the number 0. Each new breakpoint is assigned the next available number. Breakpoints remain in memory until you explicitly delete them. Breakpoints are saved in the CURRENT.STS file when you exit CodeView and are restored the next time you debug the program.

Types of Breakpoints

You can set breakpoints to break execution when any of the following conditions are true:

- The program reaches a given source line, function, label, or address.

- An expression becomes true (nonzero). The CodeView expression evaluator evaluates this type of expression based on the the currently visible function.
- The value of an expression or memory range changes. CodeView references this type of expression by memory location. As a result, the original value of the expression is checked no matter which function is currently visible.

Mouse and Keyboard

In addition to typing the **BP** command, you can also set a breakpoint with the following shortcuts:

- From the Data menu, choose Set Breakpoint.
- Double-click a source line.
- Move the cursor to a source line, and press F9.

Examples

Command	Action
`BP @47`	Set a breakpoint at line 47 of the currently executing module.
`BP 0x23f0:3c84`	Set a breakpoint at address 23F0:3C84.
`BP =curr_sum`	Halt execution whenever the value in `curr_sum` changes.
`BP =myint /R8`	Halt execution whenever a change occurs in the range of eight integers that begins at `myint`. If `myint` is a 2-byte value, the range is 16 bytes in size.
`BP @47 =int_array[[0]] /R20`	Set a breakpoint at line 47 of the currently executing module. The breakpoint will be taken at that line if any 20 elements of the array `int_array` changes. Since `int_array` is a 2-byte value, the range is 40 bytes in size.
`BP {,mymod}@47 ?myptr==0`	Set a breakpoint at line 47 of the module `mymod`. The breakpoint is taken only if `myptr` is zero.
`BP stats /P10 /C"?counter+=1"`	Set a breakpoint at the address of the function `stats` but ignore the breakpoint the first nine times the function is executed. On the tenth and later call to `stats`, stop execution, and use the Display Expression (**?**) command to increment the value of `counter`. If `counter` is set to 0 when the breakpoint is set, `counter` can be used to count the number of times the breakpoint is taken.

E (Animate)

Syntax

E

Description

The Animate (**E**) command traces through a program one step at a time, with a user-selectable pause between each step, beginning at the current instruction. In the Source mode, CodeView pauses after each line of source text. In the Mixed or Assembly-only mode, CodeView pauses after each instruction. The Animate command allows you to see how execution proceeds in your program.

You can set the time the command pauses with the Trace Speed (**T**) command or by choosing Trace Speed from the Options menu.

Mouse and Keyboard

To begin animating, you can also choose Animate from the Run menu. There is no keyboard shortcut.

G (Go)

Syntax

G [[*address*]]

Parameters

address
Address at which to stop execution.

Description

The Go (**G**) command starts execution at the current address. If *address* is given, CodeView executes the program until it reaches that address. If the specified address is never reached, the program executes until it terminates. If no address is given, CodeView executes the program until it terminates, until it reaches a breakpoint, or until you interrupt execution.

When CodeView reaches the end of the program in MS-DOS, it displays a message with the format:

`Program terminated normally` (*number*)

The *number* is the program's return value (also known as the "exit" or "errorlevel" code). This is the value in the AX register at the time your program terminates. For example, the C function call

```
exit(1);
```

places 1 in the AX register and terminates the program.

Mouse and Keyboard

In addition to typing the **G** command, you can start execution using the following shortcuts:

- Click the `<F5=Go>` button in the status bar.
- Press F5.

To execute up to a certain location, you can use the following shortcuts:

- Click the right mouse button on the source line.
- Move the cursor to the source line and press F7.

Example

The following example executes up to the label `panic_exit` in the **main** function. Because labels are always local to a procedure, you must specify the context (procedure or function name) if the label is not in the current function.

```
>G {main}panic_exit
```

H (Help)

Syntax

H [[*topic*]]

Parameter

topic
Provides help on *topic*, which can be a command-window command. If no *topic* is given, the table of contents is displayed.

Description

The Help (**H**) command displays help information in a separate window. You can get help on CodeView commands, CodeView error messages, and any other topic within the Microsoft Advisor Help system.

Mouse and Keyboard

In addition to typing the **H** command, you can get help using the following shortcuts:

- With the right mouse button, click the keyword to display the corresponding Help topic. This method works in all CodeView windows except the Source, Memory, and 8087 windows.
- Move the cursor to a topic and press F1 to display the corresponding Help topic.
- Choose one of the commands on the Help menu.

I (Port Input)

Syntax

I *port*

Parameter

port
A 16-bit port address.

Description

The Port Input (**I**) command reads and displays a byte from a specified hardware port. The specified port can be any 16-bit address. CodeView displays the byte read in the Command window.

This command is often used in conjunction with the Port Output (**O**) command. Use this command to write and debug hardware-specific programs in Assembly mode.

Note This command may affect the status of the hardware using the port.

Example

The following example reads the input port numbered 2F8 and displays the result, E8. You can enter the port address using any radix, but the result is always displayed in current radix.

```
>I 2F8 ;* hexadecimal radix assumedE8
```

K (Stack Trace)

Syntax

K

Description

The Stack Trace (**K**) command displays functions that have been called during program execution, including their arguments in the Command window. It also displays the address of the instruction that will be executed when control returns to each function.

Output from the Stack Trace command gives you the following information:

- Functions listed in the reverse order in which they were called.
- Arguments to each function, listed in parentheses.
- The address or line number of the next instruction to be executed when control returns to that function.

Thus, the current function is listed first, and the address of the next instruction to be executed is the current CS:IP address. At the bottom is the main function of your program and the address of the next instruction to be executed when execution returns to the main function.

For each function, the command shows argument values in the current radix in parentheses after the function name.

You can use the address displayed for each line of the stack trace as an argument to the View Source (**VS**) or Unassemble (**U**) commands to see the code at the point where each function is called.

Mouse and Keyboard

In addition to typing the (**K**) command, you can use the Calls menu to see the stack trace.

Remarks

The term “stack trace” is used because as each function is called, its address and arguments are stored on or pushed onto the program stack. CodeView traces through the program stack to find out which functions were called. With C programs, the function **main** is always at the bottom of the stack.

The Stack Trace (**K**) command does not display anything until the program executes the beginning of the main function. The main function sets up the stack trace through frame pointers (the BP register), which CodeView uses to locate parameters, local variables, and return addresses.

If the main module is written in assembly language, the program must execute at least to the beginning of the first procedure called. In addition, your procedures must follow the standard Microsoft calling conventions.

Example

The following example shows the functions executed in a program so far, where hexadecimal is the current radix under CodeView:

```
>K
convert(0x3:0x17FC,1,2) address 1:ada
make_header(0x3:0x17FC) address 1:314
main(4,0x3:0x181E) address 1:c98
>
```

Here, `convert` is the currently executing function, at address ADA. It ispassed three parameters: a pointer and two integers. When it returns control to `make_header`, the program is executing at address 314. When `make_header` returns, the program is executing at address C98.

L (Restart)

Syntax

L [[*arguments*]]

Parameter

arguments
New arguments to the program. No other CodeView commands may be specified after the Restart command. They are interpreted as additional program arguments.

Description

The Restart (**L**) command resets execution to the beginning of the program and optionally sets a new program command line.

After you issue the Restart command:

- The program's variables are reinitialized.
- The program's instructions are reset. Any modifications you may have made to the code with the Assemble (**A**), Memory Enter (**ME**), Memory Fill (**MF**), or View Memory (**VM**) commands are lost.
- Any existing breakpoints or watch statements are retained. The pass counts for all breakpoints are reset.

Used alone, the Restart command keeps the previous command-line arguments specified for your program. You can change the command-line arguments using the Restart command followed by any new arguments to your program.

You can use Restart any time execution has stopped: at any kind of breakpoint, while single-stepping, or when execution is complete.

Mouse and Keyboard

In addition to typing the **L** command, you can also restart from the Run menu by choosing Restart. To set a new command line and restart the program, choose Set Runtime Arguments from the Run menu. There is no keyboard shortcut.

Remarks

The Restart command does not reset system resources, such as open files or video mode, and does not free allocated system objects. If the application redefines interrupts, the system may no longer work correctly.

MC (Memory Compare)

Syntax

MC *range address*

Parameters

range
Range of first block of memory.

address
Starting address of second block of memory.

Description

The Memory Compare (**MC**) command compares the bytes in a given range of memory with the corresponding bytes beginning at another address. If one or more pairs of corresponding bytes do not match, the command displays each pair of mismatched bytes.

You can enter arguments to the Memory Compare (**MC**) command in any radix, but the output of the command is always in hexadecimal.

Examples

The following example compares the block of memory from 100 to 1FF with the block from 300 to 3FF. CodeView reports that the third and ninth bytes differ in the two ranges.

```
>MC 100 1FF 300 ;* hexadecimal radix assumed
004E:0102 0A 00 004E:0302
004E:0108 0A 01 004E:0308
>
```

The following example compares the 100 bytes starting at the address of `arr1[[0]]` with the 100 bytes starting at the address of `arr2[[0]]`.

```
>MC arr1[[0]] L 100 arr2[[0]] ;* decimal radix assumed
>
```

Because CodeView produced no output, the first 100 bytes of each array are identical.

MD (Memory Dump)

Syntax

MD[[*format*]] [[*address*|*range*]]

Parameters

format

Specifies the format to dump data. The format can be one of the following:

Specifier	Format
A	ASCII characters
B	Byte (hexadecimal)
C	Code (instructions)
I	Integer (2-byte decimal)
IU	Integer unsigned (2-byte decimal)
IX	Integer hex (2-byte hexadecimal)
L	Long (4-byte decimal)
LU	Long unsigned (4-byte decimal)
LX	Long hex (4-byte hexadecimal)
R	Real (4-byte float)
RL	Real long (8-byte float)
RT	Real ten-byte (10-byte float)

If *format* is not given, the Memory Dump command defaults to the format last used. If never used before, it defaults to an 8-bit dump.

address
Starting address of memory to be dumped. This can be any expression that evaluates to an address. The amount of memory dumped depends on the format specified. If *address* is omitted, the Memory Dump command defaults to the byte immediately following the last byte in the previous dump command. If the Memory Dump command was never used before, it defaults to DS:0000.

range
Range of memory to be dumped. Maximum range is 32K.

Description

The Memory Dump (**MD**) command displays the contents of memory in the command window, using the format you specify. This command can be used with the Redirection commands to send the contents of memory to another device. Use the View Memory (**VM**) command to display the contents of memory in a separate window.

The Memory Dump Code (**MDC**) command is like the Unassemble (**U**) command, except that **MDC** displays instructions in the Command window instead of the active Source window. Although you normally specify a range with the L character, you can also use the I character with **MDC** to specify a range of instructions rather than bytes.

Examples

The following example displays 12 instructions starting from the address at line number 32 in the source code:

```
>mdc .32 I 12
```

The following example displays the byte values in the range between DS:0 and DS:1B. The data segment is assumed when no segment is given. ASCII characters are shown on the right.

```
>mdb 0x0 0x1b
0087:0000  00 00 00 00 00 00 00 00 4D 53 20 52 75 6E  ........MS Run
0087:000E  2D 54 69 6D 65 20 4C 69 62 72 61 72 79 20  -Time Library
>
```

The following example displays seven elements of `float_array` as 4-byte real values:

```
>mdr float_array[[0]]
0087:0D56  DC 0F 49 40  +3.141593E+000
0087:0D5A  A0 17 CE 3F  +1.610096E+000
0087:0D5E  66 66 5B C2  -5.485000E+001
0087:0D62  00 00 C0 3F  +1.500000E+000
0087:0D66  FF FF 1F 41  +9.999999E+000
0087:0D6A  00 00 00 00  +0.000000E+000
0087:0D6E  00 00 00 00  +0.000000E+000
>
```

ME (Memory Enter)

Syntax

ME*type address* [[*list*]]

Parameters

type

Specifies the type of the data to be entered into memory.

Specifier	Type
A	ASCII characters
B	Byte (hexadecimal)
I	Integer (2-byte decimal)
IU	Integer unsigned (2-byte decimal)
IX	Integer hex (2-byte hexadecimal)
L	Long (4-byte decimal)
LU	Long unsigned (4-byte decimal)
LX	Long hex (4-byte hexadecimal)
R	Real (4-byte float)
RL	Real long (8-byte float)
RT	Real ten-byte (10-byte float)

If no *type* is given, the command defaults to the last type used by a Memory Enter (**ME**), a Memory Dump (**MD**), or a View Memory (**VM**) command. If no such commands were used, it defaults to byte-sized data.

address

Indicates where the data will be entered. If no segment is given in the address, the data segment (DS) is assumed.

list
List of data to enter into memory at *address*. These expressions must evaluate to data of the size specified by *type*. If *list* is not given, CodeView prompts for new values.

Description

The Memory Enter (**ME**) command enters one or more byte values into memory at a given address.

The command may include a list of expressions separated by spaces. The expressions are evaluated and entered in the current radix. If an invalid value appears in the list, CodeView refuses to enter the invalid value and ignores remaining values. If no list is given, CodeView prompts for new values.

Because it can modify any part of your program's memory, the Memory Enter command can change your program's instructions. The Assemble (**A**) command, however, is better suited to that purpose.

Mouse and Keyboard

There is no keyboard shortcut to enter items into memory. You can use the Memory window, however, to alter items in memory.

Entering Values

If you do not give a list of expressions in a Memory Enter (**ME**) command, CodeView prompts for a new value at the address you specify by displaying the address and its current value. At this point, you can do one of the following:

- Replace the value by typing a new value.
- Skip to the next value by pressing the SPACEBAR. Once you have skipped to the next value, you can change its value or skip again. CodeView will automatically prompt with new addresses as necessary.
- Return to the preceding value by typing a backslash (\). When you return to the preceding value, CodeView starts a new display line and prompts with the address and current value.
- Stop entering values and return to the command prompt by pressing ENTER.

Example

The following example replaces the byte at DS:256 (DS:0100 hexadecimal) with 66 (42 hexadecimal).

```
>MEB 256
3DA5:0100  41  A. 66
>
```

MF (Memory Fill)

Syntax

MF *range list*

Parameters

range
Specifies the range of memory to be filled.

list
List of byte values used to fill *range*.

Description

The Memory Fill (**MF**) command fills the addresses in the specified range with the byte values specified in the argument list. You can enter byte values using any radix.

The values in the list are repeated until the whole range is filled. Thus, if you specify only one value, the entire range is filled with that same value. If the list has more values than the number of bytes in the range, the command ignores any extra values.

The Memory Fill command provides an efficient way to fill up a block of memory with any values you specify. You can use it to initialize large data areas, such as arrays or structures. Because it can modify any part of your program's memory, the Memory Fill command can change your program's instructions. However, the Assemble (**A**) command is better suited to that purpose.

Examples

The following example fills 255 (100 hexadecimal) bytes of memory starting at DS:0100 with the value 0; hexadecimal radix is assumed. This command could be used to reinitialize the program's data without having to restart the program.

```
>MF 100 L 100 0
>
```

This next example fills the 100 (64 hexadecimal) bytes starting at `table` with the following hexadecimal byte values: 42, 79, 74. These three values are repeated (42, 79, 74, 42, 79, 74,) until all 100 bytes are filled; hexadecimal radix is assumed.

```
>MF table L 64 42 79 74
>
```

MM (Memory Move)

Syntax

MM *range address*

Parameters

range
Specifies the range of memory to copy.

address
Destination address to copy the *range*.

Description

The Memory Move (**MM**) command copies all the values in one block of memory directly to another block of memory of the same size. All data in the source block is guaranteed to be copied completely over the destination block, even if the two blocks overlap.

When the source is at a higher address than the destination, the Move Memory command copies data starting at the source block's lowest address. When the source is at a lower address, the Memory Move command copies data beginning at the source block's highest address.

You use the Memory Move command to program in Assembly mode (to copy function fragments, for example) or to copy large amounts of data.

Examples

In the following example, the block of memory to copy begins with the first element of `array1` and is `array_size` bytes long. It is copied directly to a block of the same size, beginning at the address of the first element of `array2`.

```
>MM array1[[0]] L array_size array2[[0]]
>
```

MS (Memory Search)

Syntax

MS *range list*

Parameters

range
The range of memory to search.

list
A list of byte values separated by spaces or commas or an ASCII string delimited by quotation marks.

Description

The Memory Search (**MS**) command scans a range of memory for specific byte values. Use this command to test for the presence of certain values within a range of data.

You can specify any number of byte values to the Memory Search command. Unless the list is an ASCII string, each byte value must be separated by a space or a comma.

If the list contains more than one byte value, the Memory Search command looks for a series of bytes that precisely match the order and value of bytes in the list. If the command finds such a series of bytes, it displays the beginning address of that series.

Examples

The following example displays the address of each memory location containing the string `error`. The command searched the first 1,500 bytes at the address specified by the variable `buffer`. CodeView found the string at three addresses.

```
>MS buffer L 1500 "error"
2BBA:0404
2BBA:05E3
2BBA:0604
>
```

The following example displays the address of each memory location that contains the byte value 0A in the range DS:0100 to DS:0200; hexadecimal is assumed to be the default radix. CodeView found the value at two addresses.

```
>MS DS:100 200 A ;* hexadecimal radix assumed
3CBA:0132
3CBA:01C2
>
```

N (Radix)

Syntax

N [[*radix*]]

Parameter

radix
New radix while running CodeView. Can be 8 (octal), 10 (decimal), or 16 (hexadecimal). If omitted, the command displays the current radix.

Description

The Radix (**N**) command changes the current radix for entering arguments and displaying the values of expressions. The new radix number can be 8 (octal), 10 (decimal), or 16 (hexadecimal). Binary and other radixes are not allowed. With no arguments, the command displays the current operating radix.

Note Changing the radix does not convert the l-value of displayed expressions, only the r-value.

When you start up CodeView, the default radix is 10 (decimal), unless your main program is written with the Microsoft Macro Assembler (MASM). In this case, the default radix is 16 (hexadecimal).

Remarks

The following conditions are not affected by the Radix command:

- The radix for entering a new radix is always decimal.
- Format specifiers given with the Display Expression (**?**) command override the current radix.
- Addresses are always shown in hexadecimal.
- In Assembly mode, all values are shown in hexadecimal.
- The display radix for the Memory Dump (**MD**) and Breakpoint Set (**BP**) commands is always hexadecimal if the size is bytes, words, or doublewords; it is always decimal if the size is integers, unsigned integers, short reals, long reals, or 10-byte reals.
- The input radix for the Memory Enter (**ME**) command's prompt is always hexadecimal if the size is bytes, words, or doublewords; it is always decimal if the size is integers, unsigned integers, short reals, long reals, or 10-byte reals.
- The current radix is used for all values given as part of a list, except real numbers, which must be entered in decimal.
- The register display is always in hexadecimal.

Example

The following example shows the decimal equivalents of the number 14 in octal and in hexadecimal.

```
>N8
>? 14,i
12
>N16
>? 14,i
20
>
```

Here, the Display Expression (**?**) command uses the **i** format specifier, which prints a number in decimal regardless of the current radix.

O (Options)

Syntax

O[[*option*[[+|-]]]]
OL[[*scope*]]

Parameters

option

Character indicating the option to be turned on or off.

Specifier	Option
A	Show Status Bar
B	Bytes Coded
C	Case Sense
F	Flip/Swap
H	Horizontal Scroll Bar
L	Show Address
N	Native Mode
S	Symbols
3	386
V	Vertical Scroll Bar

scope

For the **OL** command, you can specify a scope of variables to display in Local window using one or more of the following:

Specifier	Scope
L	Lexical
F	Function
M	Module
E	Executable
G	Global
*	All of the above

+

Turns option(s) on.

–

Turns option(s) off.

Description

The Options (**O**) command allows you to view or set the state of the following CodeView options:

Letter	Option	Display
A	Show Status Bar	If on, the status bar appears at the bottom of the screen. If off, the bottommost line becomes part of the window area.
B	Bytes Coded	If on, instruction addresses and machine code are displayed for assembly instructions.
C	Case Sense	If on, symbols are case-sensitive; if off, they are not.
F	Flip/Swap	If on, CodeView flips the program and output screens as the program executes. If off, no screen flipping is performed.
H	Horizontal Scroll Bar	If on, windows have a horizontal scroll bar.
L	Show Address	If on, addresses relative to BP for all local variables are displayed in the Local window.
N	Native Mode	If on, instructions are displayed in the native processor format. If off, p-code instructions are displayed.
S	Symbols	If on, symbols in assembly instructions appear in symbolic form. If off, they appear as addresses.
3	386	If on, registers appear in wide 80386 format, and you can assemble and execute instructions that reference 32-bit registers and memory.
V	Vertical Scroll Bar	If on, windows have a vertical scroll bar.

The Local window always displays variables local to the current routine. You can specify a scope of additional variables to display in the Local window with **OL** form of the Options command. Using **OL** with no options displays the current scope setting for the Local window.

The **O** form of the command (all options) takes no arguments; it displays the state of all options. The other forms of the command (**OF**, **OB**, **OC**, **OS**, **OL**, **O3**, **OA**, **ON**, **OH**, and **OV**) can be used as follows:

- With no arguments. The state of the option is displayed.
- With the + or – argument. The + argument turns the option on; the – argument turns the option off.

Mouse and Keyboard

As an alternative to typing the **O** command, you can view and set options on the Options menu.

Remarks

Use the Options (**O**) command to set options when you first start CodeView. You can set these options in the following ways:

- Give one or more **O** commands with the /C option on the CodeView command line or include a similar command line in the CodeView response file.
- Give one or more **O** commands as the **Autostart** entry in the TOOLS.INI file.

Example

In the following example, the **O** command is used to display current option settings. Then, the **O3** and **OF** commands are used to display and set options for 386 mode and for screen flip/swap mode. Finally, the **OL** command turns on symbol addresses in the Local window and displays not only local variables but global variables as well.

```
>O
Flip/Swap On
Bytes Coded On
Case Sense On
Show Symbol Address On
Symbols Off
Vertical scroll bar On
Horizontal scroll bar On
Status bar On
>O3
386 Off
>O3+
386 On
>OF
Flip/Swap On
>OF-
Flip/Swap Off
>OLG+
```

O (Port Output)

Syntax

O *port byte*

Parameters

port
A 16-bit port address.

byte
Byte to send to *port*.

Description

The Port Output (**O**) command sends a byte value to a hardware port. You use this command to debug a program that interacts directly with hardware.

The Port Output command is often used with the Port Input (**I**) command.

Example

In the following example, the byte value 4F hexadecimal is sent to output port 2F8.

```
>O 2F8 4F ;* hexadecimal radix assumed
>
```

P (Program Step)

Syntax

P [[*count*]]

Parameters

count
Repeat stepping *count* times.

Description

The Program Step (**P**) command executes the current line (in Source mode) or instruction (in Mixed or Assembly mode), stepping over functions. To trace into functions, use the Trace (**T**) command. If a value for *count* is specifed, CodeView steps through *count* lines or instructions. If not, only the current line or instruction is executed.

In Source mode, if the current source line contains a function call, CodeView executes the entire function and is ready to execute the line after the call.

In Mixed or Assembly Mode, if the current instruction is CALL, INT, or REP, CodeView executes the entire procedure, interrupt, or repeated string sequence.

Mouse and Keyboard

In addition to typing the **P** command, you can step through a program using the following shortcuts:

- Click the `<F10=Step>` button in the status bar to step once.
- Press F10 to step once.

Q (Quit)

Syntax

Q

Description

The Quit (**Q**) command terminates CodeView and returns control to the environment from which CodeView was invoked: Programmer's WorkBench (PWB), Windows, or the operating system.

CodeView always saves state information on exit.

Mouse and Keyboard

As an alternative to typing the **Q** command, choose Exit from the File menu. There is no keyboard shortcut.

R (Register)

Syntax

R [[*register* [[[[=]]*expression*]]]]

Parameters

register
Change the contents of the given register. If omitted, displays the values of all registers and flags and the current machine instruction.

[[=]]*expression*
Assign the value of the expression to the specified register. The equal sign (=) is optional; a space has the same effect.

Description

The Register (**R**) command displays and changes the values in the CPU registers. To display register contents without changing them, type the Register (**R**) command without any arguments. This form of the command shows the current values of all registers and flags. Flags are shown symbolically. It also shows the current instruction at the address given by CS:IP.

If an operand of the instruction contains memory expressions or immediate data, CodeView evaluates the operand and indicates the value to the right of the instruction. This value is referred to as the "effective address" and is also displayed at the bottom of the Register window.

Changing Registers

You can use the **R** command to change the values in CPU registers. Also, you can change the bits in the flag register symbolically without having to compute a value of the register.

Changing Register Values

To change the value in a register:

1. Type the command letter **R** followed by the name of a register. The register name can be any of the following: AX, BX, CX, DX, CS, DS, SS, ES, SP, BP, SI, DI, IP, or FL (for flags). If you have an 80386/486-based machine and the 386 option is turned on, the register name can be one of the 32-bit register names: EAX, EBX, ECX, EDX, ESP, EBP, ESI, or EDI.

   Note CodeView allows you to load different execution models which may specify a certain set of registers. For example, the valid registers in the p-code model are DS, SS, CS, IP, SP, BP, PQ, TH, and TL.

2. CodeView displays the current value of the register and prompts for a new value with a colon (**:**).

   - If you only want to examine the value, press ENTER.

- If you want to change the value, type an expression (in the current radix) for the value and press ENTER.
- As an alternative, you can use the Display Expression (**?**) command to change the value in a register:

  ?*register***=***expression*

Changing Flag Values

To change a flag value:

1. Type the command letter **R** followed by the letters FL.
2. The command displays the value of each flag as a two-letter name. At the end of the list of values, the command prompts for new flags with a dash (–).
3. Type the new values after the dash for the flags you wish to change, then press ENTER.
   - You can enter flag values in any order. If you do not enter a new value for a flag, it remains unchanged.
   - If you do not want to change any flags, press ENTER.

Note If you enter an illegal flag name, CodeView displays an error message. The flags preceding the error are changed; flags at and following the error are not changed.

The flag values are:

Flag	Set Symbol	Clear Symbol
Overflow	OV	NV
Direction	DN	UP
Interrupt	EI	DI
Sign	NG	PL
Zero	ZR	NZ
Auxiliary carry	AC	NA
Parity	PE	PO
Carry	CY	NC

Mouse and Keyboard

As an alternative to typing the **R** command, you can use the Register window to display CPU values. To change CPU values with the Register window, type over the old values.

Example

In the following example, the **R** command displays the current registers and CPU flags. Then the **R** command changes the value in the AX register.

```
>R
AX=0005  BX=299E  CX=0000  DX=0000  SP=3800  BP=380E  SI=0070  DI=40D1
DS=5067  ES=5067  SS=5067  CS=4684  IP=014F
NV UP EI PL NZ NA PO NC
0047:014F 8B5E06          lea     di,[[BP+06]] ss:ff38=299E
>R AX
AX 0005
:3
>
```

T (Trace)

Syntax

T [[*count*]]

Parameters

count
Repeat tracing *count* times. If omitted, trace once.

Description

The Trace (**T**) command executes the current line (in Source mode) or instruction (in Assembly or Mixed mode), tracing into functions or assembly-language CALL instructions. Use the Program Step (**P**) command to execute function calls without tracing into them.

In Source mode, the Trace command traces into functions whose source code is available and executes through those functions whose source is unavailable.

In Assembly or Mixed mode, CodeView always traces into functions. If the current instruction is CALL or INT, CodeView executes the first instruction of the procedure or interrupt. If the current instruction is REP, CodeView executes one iteration of the repeated string sequence.

CodeView executes MS-DOS function calls without tracing into them. CodeView can trace through BIOS calls in Assembly or Mixed mode.

Since the Trace command uses the hardware trace mode of the 8086 family of processors, you can also trace instructions stored in read-only memory (ROM). However, the Program Step command does not work in ROM; in this case, it has the same effect as the Go (**G**) command.

Mouse and Keyboard

In addition to typing the **T** command, you can trace once with the following shortcuts:

- Click the `<F8=Trace>` button in the status bar.
- Press F8.

T (Trace Speed)

Syntax

T{**S**|**M**|**F**}

Parameter

{**S**|**M**|**F**}
Specifies the trace speed for the Animate (**E**) command. You can specify the following speeds:

Specifier	Speed
S	Slow (1/2 second between steps)
M	Medium (1/4 second between steps; default)
F	Fast (no wait between steps)

Description

The Trace Speed command controls the speed at which CodeView executes a program with the Animate (**E**) command.

Mouse and Keyboard

In addition to typing the **TS**, **TM**, or **TF** commands, you can also set the trace speed from the Options menu. There is no keyboard shortcut.

U (Unassemble)

Syntax

U [[*context*]][[*address*]]

Parameters

context
Any legal context operator.

address
Shows unassembled instructions starting at this address. If omitted, unassemble at the current CS:IP address.

Description

The Unassemble (**U**) command displays assembly-language code beginning at the specified address in the active Source window. If you omit an address, the command uses the current CS:IP address. The command changes the Source window to Assembly mode.

Setting the Source window display mode to Assembly and giving the Unassemble command with no arguments causes the code to scroll to the next page of assembly-language instructions.

Note If you specify an address that is within an instruction or within program data, CodeView will still attempt to disassemble and display instructions. Instructions that CodeView cannot disassemble are shown as `???`.

Mouse and Keyboard

As an alternative to typing the **U** command, you can display assembly-language instructions using the following shortcuts:

- Click the `<F3=Src1 Fmt>` or `<F3=Src2 Fmt>` buttons until the active Source window is in Assembly mode.
- Press F3 until the Source window is in Assembly mode.
- From the Options menu, choose Source Window. Then set the display mode to Assembly.

Note that with these shortcuts, you cannot specify an address to start showing unassembled instructions.

Example

The following example sets the mode of the Source window to Assembly and displays assembly-language instructions beginning at address 0x7:0x11.

```
>U 0x7:0x11
>
```

USE (Use Language)

Syntax

USE *evaluator*

Parameter

evaluator
Selects the specified expression evaluator. If omitted, the command displays the currently selected expression evaluator. You can specify **AUTO** for the evaluator. With this setting, CodeView selects the appropriate expression evaluator based on the extension of the source file.

Description

The **USE** command specifies which expression evaluator CodeView is to use while debugging.

Mouse and Keyboard

As an alternative to typing the **USE** command, choose the Language command from the Options menu. There is no keyboard shortcut.

Remarks

When you switch expression evaluators, CodeView displays expressions in the Local and Watch windows with the nearest equivalent type in the new language. If the new language does not have an equivalent type, the results are unpredictable.

VM (View Memory)

Syntax **VM**[[*window*]] [[*type*]] [[*address*]] [[*options*]]

Parameters

window
Specifies the memory window to display or change (1 or 2). If a value for *window* is omitted, the command defaults to the active Memory window or Memory window 1 if no Memory windows are open.

type
Specifies the data-type format of the window's display.

Value	Format
A	ASCII characters
B	Byte (hexadecimal)
I	Integer (2-byte decimal)
IU	Integer unsigned (2-byte decimal)
IX	Integer hex (2-byte hexadecimal)
L	Long (4-byte decimal)
LU	Long unsigned (4-byte decimal)
LX	Long hex (4-byte hexadecimal)
R	Real (4-byte float)
RL	Real long (8-byte float)
RT	Real ten-byte (10-byte float)

If *format* is omitted, the command defaults to the last type used by a View Memory (**VM**) command or to byte-display format if the **VM** command was never used.

address
Starting address of memory to display or any expression that evaluates to an address. If *address* is omitted, the command defaults to the current address in the active Memory window or DS:00 if no Memory windows are open.

options
Specifies how to display and update the Memory window's contents.

/R[[+|–]]
Raw data display

Option	Description
+	CodeView displays formatted data along with the corresponding bytes in hexadecimal format.
– (default)	CodeView displays only formatted data.

/L[[+|–]]
Live expression

Option	Description
+	Dynamic: CodeView evaluates *address* at each step and adjusts the Memory window accordingly.
– (default)	Static: CodeView evaluates *address* only when the command is entered.

/F[[*****| *length*]]
Fixed-width data display

Option	Description
***** (default)	CodeView displays as many items as will fit in the window.
length	CodeView displays a fixed number of items on each line. Must be in the range 1–125.

Description

The View Memory (**VM**) command displays the contents of memory in a Memory window using the type and format you specify. The Memory window is updated whenever you execute a command. You can modify memory in the window directly by typing over the displayed memory.

If you enter the **VM** command with no arguments and no Memory windows are open, CodeView opens Memory window 1 in the default display format (variable-width byte display at a static address). If you enter the **VM** command with no arguments and at least one Memory window is open, CodeView displays the current settings for the Memory windows in the Command window.

You can directly modify memory using the Memory window. Type over the values displayed in the active Memory window.

To display the contents of memory in the Command window, use the Memory Dump (**MD**) command.

Mouse and Keyboard

In addition to typing the **VM** command, you can open and manipulate Memory windows with the following shortcuts:

- To open a Memory window from the Windows menu, choose Memory 1 or Memory 2.
- To set display format and enter expressions for a Memory window, choose Memory Window from the Options menu.

You can cycle through the display formats with the following shortcuts:

- Click the `<SH+F3=Mem1 Fmt>` or `<SH+F3=Mem2 Fmt>` buttons in the status bar.
- Press SHIFT+F3 to cycle forward.
- Press CTRL+SHIFT+F3 to cycle backward.

- When the cursor is in the Memory window, press CTRL+O to display the Memory Window Options dialog box.

Examples

The following example opens Memory window 2 and displays memory in integer format plus the raw bytes that make up the integers, beginning at the address of the variable `myint`.

```
>VM2I /R myint
```

The following example specifies ASCII format for the current Memory window. The memory displayed begins at the string referred to by element `i` of the array `argv`. The expression is live, so the display is updated as `i` changes.

```
>VMA /L *argv[[i]]
```

VS (View Source)

Syntax

VS[[*window*]] [[*format*]] [[*address*]] [[*/option*[[+|-]]]]...

Parameters

window
Specifies the Source window (1 or 2) to open or make active.

format
Specifies the way to display source code as one of the following:

Specifier	Format
+	Display source lines from the source file
–	Display assembly-language instructions
&	Display both source lines and assembly-language instructions

address
Address or line number at which to start displaying source code. The address must fall within the executable portion of your program.

[[/*option*[[+|-]]]]...

Zero or more source display options. The option can be any of the following specifiers:

Specifier	Option
a	Address When turned on (/a[[+]]), displays the address of each instruction. When turned off (/a-), does not display addresses.
b	Bytes coded When turned on (/b[[+]]), displays the hexadecimal form of the instructions. When turned off (/b-), does not display the encoded bytes.
c	Case of disassembly When turned on (/c[[+]]), displays instruction mnemonics and registers in uppercase. When turned off (/c-), displays instruction mnemonics and registers in lowercase.
l	Line-oriented display When turned on (/l[[+]]), displays mixed source and assembly in source-line order. When turned off (/l-), displays mixed source and assembly in instruction-code order.
s	Symbols in disassembly When turned on (/s[[+]]), symbols in instructions appear in symbolic form. If turned off (/s-), they appear as addresses.
t	Track current location (CS:IP) When turned on (/t[[+]]), the Source window follows the thread of execution (CS:IP). When turned off (/t-), the Source window does not automatically scroll to follow the current location.

Description

CodeView can display two Source windows at the same time. At least one source window must always be open. You can type the **VS 1** or **VS 2** command to make Source window 1 or 2 active. If the Source window you request is not open, CodeView opens it and makes it active.

The Source windows can show code in a number of display modes:

Source
: CodeView displays the lines from your program's source files.

Assembly
: CodeView displays the assembly instructions that make up your program.

Mixed
: CodeView displays each line of your program's source file, followed by the assembly instructions for that line. This ordering can be reversed by turning the Line-Oriented Display option off (/l-).

Source and Mixed modes are available only if the executable file contains debugging information.

Note Programs that do not contain debugging information are always displayed in Assembly mode.

In the Source and Mixed modes, tracing into a function for which no source lines are available, such as a library function, switches the Source window to Assembly mode. Once program execution returns to an area where source lines are available again, CodeView automatically switches back to Source or Mixed mode.

If you specify a line number or an address with the **VS** command, CodeView draws the Source window so that the source line corresponding to the given address appears in the middle of the Source window. If the address is in another file, CodeView loads that file into the Source window. If you specify an address for which there is no corresponding source text (in your program's data, for example), CodeView will respond with an error message.

You can scroll the contents of the active Source window down a page by typing the **VS** command with no arguments. You can also use the Source window scroll bars.

Mouse and Keyboard

To make a Source window active or to open a Source window:

- Click anywhere in an open Source window to make it active.
- Press ALT+3 or ALT+4 to activate or open Source window 1 or 2.
- From the Windows menu, choose Source 1 or Source 2.

To change the source display mode:

- Click the `<F3=Src1 Fmt>` or `<F3=Src2 Fmt>` buttons in the status bar to cycle through the three modes.
- Press F3 to cycle forward.
- Press CTRL+F3 to cycle backward.
- From the Options menu, choose Source Window to open the Source Window Options dialog box. Under Display Mode, select one of the option buttons.
- When the cursor is in the Source window, press CTRL+O to display the Source Window Options dialog box.

Examples

The following example opens Source window 2 in the mixed mode. The display will start at the function `toss_token`.

```
>VS 2 & toss_token
```

The next example changes the display format in Source window 2 to source lines only.

```
>VS 2 +
```

W? (Add Watch Expression)

Syntax

W? *expression* [[**,** *format*]]

Parameters

expression
Expression to add to the Watch window.

format
A CodeView format specifier that indicates the format in which *expression* is displayed.

Description

The Add Watch Expression (**W?**) command displays one or more specified values in the Watch window. Watch expressions allow you to watch how a variable changes as your program executes. CodeView updates the Watch window each time the value of the watch expression changes during program execution.

The Watch window shows variables in the default format for their types. To display a watch expression in a different format, type a comma after the expression, followed by a CodeView format specifier. You can also cast the expression to the format you want to use.

CodeView always evaluates watch expressions according to the current radix and reevalutes watch expressions if the radix changes.

For relational expressions, the Watch window shows 0 if the expression is false and 1 if the expression is true.

Mouse and Keyboard

As an alternative to typing the **W?** command, choose the Add Watch command from the Data menu. There is no keyboard shortcut.

Examples

Command	Action
`W? n`	Display the value of the variable `n` in the Watch window.
`W? high * 100`	Display the value of 100 times the variable `high` in the Watch window.
`W? (char *) 0`	Display the byte at DS:0. Because 0 is explicitly cast to a pointer type, CodeView treats it as an offset rather than a constant.

WC (Delete Watch Expression)

Syntax

WC *number*|*****

Parameters

number
Deletes the watch expression with this number.

*****
Deletes all watch expressions.

Description

The Delete Watch Expression (**WC**) command removes a watch expression from the Watch window.

When you set a watch expression, CodeView automatically assigns it a number, starting with 0 for the first watch expression in the window. Use the List Watch (**WL**) command to view the numbers of current watch statements.

Mouse and Keyboard

In addition to typing the **WC** command, you can use the following shortcuts to delete watch expressions:

- From the Data menu, choose Delete Watch.
- Select the Watch window, move the cursor to the watch expression, and press CTRL+Y.

Examples

The following example deletes watch expression 2 from the Watch window:

```
>WC 2
```

The following example deletes all watch expressions from the Watch window:

```
>WC *
```

WDG (Windows Display Global Heap)

Syntax

WDG [[*ghandle*]]

Parameter

ghandle
A handle to a global memory object. The **WDG** command displays the five memory objects in the global heap, starting at the specified object. The *ghandle* must be a valid handle to an object allocated on the global heap. If *ghandle* is not specified, **WDG** displays the entire global heap.

Description

Global memory objects are displayed in the order in which Windows manages them, which is typically not in ascending handle order. The output from the **WDG** command has the following format:

Format

handle address size **PDB** *locks type owner*

Any field may not be present if that field is not defined for the block.

Field	Description
handle	Value of the global memory block handle.
address	Address of the global memory block.
size	Size of the block in bytes.
PDB	Block owner. If present, indicates that that task's Process Descriptor Block is the owner of the block.
locks	Count of locks on the block.
type	The memory-block type.
owner	The block owner's module name.

WDL (Windows Display Local Heap)

Syntax

WDL

The output from the **WDL** command has the following format:

Format

handle address size flags locks type heaptype blocktype

Any field may not be present if that field is not defined for the block.

Field	Description
handle	Value of the global memory block handle
address	Address of the block
size	Size of the block in bytes
flags	The block's flags
locks	Count of locks on the block
type	The type of the handle
heaptype	The type of heap the block resides in
blocktype	The block's type

WDM (Windows Display Modules)

Syntax **WDM**

Description The **WDM** command displays a list of all DLL and application modules loaded by Windows. Each line of the display has the following format:

Format *handle refcount module path*

Field	Description
handle	The module handle
refcount	The number of times the module has been loaded
module	The name of the module
path	The path of the module's executable file

WGH (Windows Dereference Global Handle)

Syntax **WGH** *handle*

Parameter *handle*
Global memory handle of memory object to convert.

Description To convert a global memory handle to a pointer, use the **WGH** command. **WGH** converts a global memory handle into a near or a far pointer. Use **WLH** to convert local memory handles.

The **WDG** and **WDL** commands convert the handle into a pointer and display the value of the pointer in *segment*:*offset* format. You can then use that value to access the memory.

In a Windows-based program, the **GlobalLock** function is used to convert memory handles into near or far pointers. You may know the handle of the memory object, but you might not know what near or far address it refers to unless you are debugging in an area where the program has just called **GlobalLock**.

You use the **WGH** command at any time to find out what the pointer addresses are for global memory handles.

Example

The following example is used to display a string in Window's global heap. First, the following code sets up the string:

```
HANDLE hGlobalMem;
LPSTR  lpstr;

hGlobalMem = GlobalAlloc( GMEM_MOVEABLE, 10L )
lpstr = GlobalLock( hGlobalMem );
lstrcpy( lpstr, "ABCDEF" );
GlobalUnlock( hGlobalMem );
```

You can display the contents of the string with the following sequence of commands:

```
>wgh hGlobalMem
0192:6E30
>? *(char far*) 0x0192:0x6E30,s
```

WKA (Windows Kill Application)

Syntax

WKA

Description

The Windows Kill Application (**WKA**) command terminates the current task by simulating a fatal error.

There may be times when you want to halt your program immediately. You can force an immediate interrupt of a CVW session by pressing CTRL+ALT+SYSREQ. You then have the opportunity to change debugging options, such as setting breakpoints and modifying variables. To resume continuous execution, press F5; to single-step, press F10.

You should take care when you interrupt the CVW session. For example, if you interrupt the session while Windows-based code or other system code is executing, using the Step or Trace functions produces unpredictable results. When you interrupt the CVW session, it is usually safest to set breakpoints in your code and then resume continuous execution rather than using Step or Trace.

If the current code is in your application, you can safely use the **WKA** command without affecting other tasks. However, the **WKA** command does not perform all the cleanup tasks associated with the normal termination of a Windows-based application.

For example, global objects created during the execution of the program but not destroyed before you terminated the program remain allocated in the global heap. This reduces the amount of memory available during the rest of the

Windows operating system session. For this reason, you should use the **WKA** command to terminate the application only if you cannot terminate it normally.

For more information on using the Windows Kill Application (**WKA**) command, see Chapter 10.

WL (List Watch Expressions)

Syntax

WL

Description

The List Watch Expressions (**WL**) command lists all currently set watch expressions along with their numbers and values.

Mouse and Keyboard

As an alternative to typing the **WL** command, you can use the Watch window to view the current watch expressions.

Example

The following example displays watch expressions and their values:

```
>WL
0) code  :  17
1) (float)letters/words  :  4.777778
2) lines==11  :  0
>
```

In the example, three watch expressions are set:

1. The variable `code`, which is 17.
2. The arithmetic expression `letters` divided by `words` as a floating point number, currently 4.777778
3. The conditional expression `lines==11`, currently false (zero).

WLH (Windows Dereference Local Handle)

Syntax

WLH *handle*

Parameter

handle
Local memory handle of memory object to convert.

Description

To convert a local memory handle to a pointer, use the Dereference Local Handle (**WLH**) command. **WLH** converts a local memory handle into a near or a far pointer. Use **WGH** to convert global memory handles.

The **WDG** and **WDL** commands convert the handle into a pointer and display the value of the pointer in *segment*:*offset* format. You can then use that value to access the memory.

In a Windows-based program, the **LocalLock** function is used to convert memory handles into near or far pointers. You may know the handle of the memory object, but you might not know what near or far address it refers to unless you are debugging in an area where the program has just called **LocalLock**.

You use the **WLH** command to find out at any time what the pointer addresses are for local memory handles.

Example

The following example uses **WLH** to refer to an array during a debugging session. First, the following code sets up the array:

```
{
   HANDLE       hLocalMem;
   int near *   pnArray;
   hLocalMem = LocalAlloc( LMEM_MOVEABLE, 100 );
   pnArray = LocalLock( hLocalMem );


   /* load values into the array */

   LocalUnlock( hLocalMem );
. . .
```

Now, after setting a breakpoint immediately after the call to **LocalLock**, the following command displays the array location:

```
>mdw pnArray
```

Outside of this fragment, though, you cannot rely on the value of the `pnArray` variable since the actual data in the memory object may move. Therefore, use the following sequence to display the correct array location:

```
>wlh hLocalMem
0192:100A
>mdw 0192:100A
```

X (Examine Symbols)

Syntax

Xscope [[*context*]][[*regex*]]

Parameters

scope
Specifies the scope in which to search for symbols. Can be one or more of the following:

Specifier	Scope
L	Lexical
F	Function
M	Module
E	Executable
P	Public
G	Global
*	All of the above

context
Specifies context under which to search with the context operator.

regex
Specifies a CodeView regular expression.

Description

The Examine Symbols (**X**) command displays the names and addresses of symbols and the names of modules defined within a program. You can specify the scope in which to search and a regular expression against which to match symbols. You can further specify a context using the context operator.

For more information on regular expressions, see Appendix B.

Examples

The following example shows all the symbols and their addresses in the current lexical scope. The command uses the regular expression `.*` to match any symbol.

```
>XL .*
```

The following example displays all symbols and their addresses in the program that start with `s_`:

```
>XE s_.*
```

! (Shell Escape)

Syntax

![[**!**]*command*]

Parameter

command
Executes the given program or operating-system command without leaving CodeView. Use the second exclamation point to return to CodeView immediately after completing *command*.

Description

The Shell Escape (**!**) command (CV only) allows you to exit from the CodeView debugger to an MS-DOS shell. You can execute MS-DOS commands or programs from within the debugger, or you can exit from CodeView to MS-DOS while retaining your current debugging context.

If you want to exit to MS-DOS and execute several commands or programs, enter the Shell Escape command with no arguments (**!**). After the MS-DOS screen appears, you can run internal system commands or programs. When you are ready to return to CodeView, type the command **exit** (in any combination of uppercase and lowercase letters). The debugging screen appears with the same status it had when you left it.

If you want to execute a program or an internal system command from within CodeView, enter the Shell Escape command followed by the name of the command or program you want to execute, as in:

!*command*

The output screen appears, and CodeView executes the command or program. When the output from the command or program is finished, the message

```
Press any key to continue...
```

appears at the bottom of the screen. Press a key to make the debugging screen reappear with the same status it had when you left it. To suppress this prompt and return directly to CodeView after the command is executed, use two exclamation points (**!!**) for the Shell Escape command.

The Shell Escape command works by executing a second copy of COMMAND.COM.

Mouse and Keyboard

In addition to typing the **!** command, you can also invoke a command shell from the File menu.

Remarks

Opening a shell requires a significant amount of free memory since the following are all resident in memory:

- CodeView
- The debugging information
- The system's command processor
- The program being debugged

If your machine does not have enough memory, an error message appears. Even if there is enough memory to start a new shell, there may not be enough memory left to execute large programs from the shell.

In order for you to use the Shell Escape commands, the executable file being debugged must release unneeded memory. Programs created with Microsoft compilers release memory during startup.

Side effects of commands executed while in a shell, such as a change in the working directory, may not be seen when you return to CodeView.

Example

In the following example, the shell command **DIR** is executed with the argument `A:*.OBJ`. The directory listing will be followed by the prompt that asks you to press any key:

```
!DIR A:*.OBJ
```

In the following example, the **COPY** command is executed and control returns to CodeView. No prompt appears.

```
!!copy output.txt d:\backup
```

" (Pause)

Syntax

`"`

Description

The Pause (") command interrupts the execution of commands from a redirected file and waits for the user to press a key. Execution of the redirected commands begins as soon as a key is pressed.

Example

The following example is from a text file redirected to the CodeView debugger. A Comment (*****) command is used to prompt the user to press a key. The Pause (") command is then used to halt execution until the user responds.

```
* Press any key to continue
"
```

(Tab Set)

Syntax

#number

Parameter

number
Number of characters for new tab stops. The valid range for *number* is 1–19.

Description

The Tab Set (#) command sets the width in spaces that the CodeView debugger fills for each tab character. The default tab is eight spaces. You can specify values in the range 1–19.

You may want to set a smaller tab size if your source code has so many levels of indentation that source lines extend beyond the edge of the screen.

This command has no effect if your source code contains no tab characters.

* (Comment)

Syntax **comment*

Description The Comment command is an asterisk (*) followed by text. The CodeView debugger echoes the text of the comment to the screen or other output device. Use this command in combination with the redirection commands when you are:

- Saving a commented session.
- Writing a commented session that will be redirected to the debugger.

Example In the following example, the user is sending a copy of a CodeView session to the file OUTPUT.TXT. Comments are added to explain the purpose of the command. The text file will contain commands, comments, and command output.

```
> T>OUTPUT.TXT
> * Dump first 20 bytes of screen buffer
> MDB 0xB800:0 L 20
B800:0000 54 17 6F 17 20 17 72 17 65 17 74 17 75 17 72 17
B800:0010 6E 17 20 17
> >CON
```

. (Current Location)

Syntax **.**

Description The Current Location (**.**) command displays the source line or assembly-language instruction corresponding to the current program location. It puts the current program location in the center of the active Source window.

Use this command if you have scrolled the current source line or assembly instruction out of the active Source window.

The Current Location (**.**) command is equivalent to the command:

```
vs .
```

/ (Search)

Syntax

/ [[*regexp*]]

Parameter

regexp
Searches for the first line containing this regular expression. If omitted, the command searches for the next occurrence of the last regular expression given.

Description

The Search (/) command searches for a regular expression in a source file.

"Regular expressions" are patterns of characters that may match one or many different strings. The use of patterns to match more than one string is similar to the MS-DOS method of using wild card characters in filenames.

CodeView's regular expressions use a subset of the UNIX syntax supported by the Programmer's WorkBench (PWB). For complete information on regular expressions in PWB and CodeView, see Appendix B.

When you enter the Search command with a regular expression, CodeView searches the source file for the first line containing the expression. If you do not give a regular expression, CodeView searches for the next occurrence of the last regular expression specified.

Even if you do not understand regular expressions, you can still use the Search command with plain text strings, since text strings are the simplest form of regular expressions. For example, you can enter

```
>/ COUNT
```

to search for the word `COUNT` in the source file.

To find strings containing a special regular expression character (`.\^$*[[]]`), you must precede the character with a backslash (\); this cancels their special meanings. For example, use the command:

```
>/ x\*y
```

to find the string `x*y`.

In Source windows, CodeView starts searching at the current cursor position and places the cursor at the line containing the regular expression. The search wraps to the beginning of the file if necessary.

Mouse and Keyboard

In addition to typing the / command, you can also search for regular expressions by choosing Find from the Search menu.

Remarks

When you search for the next occurrence of a regular expression, CodeView searches to the end of the file, then wraps around and begins again at the start of the file. This search can have unexpected results if the expression occurs only once. For example, when you give the command repeatedly, there is no activity on the screen. Actually, CodeView is repeatedly wrapping around and finding the same expression each time.

The Case Sensitivity command on the Options menu and the Options Case Sense (**OC**) command affect regular expression searches.

If you want to find a label in your source code, you can also use the View Source (**VS**) command.

7 (8087)

Syntax

7

Description

The 8087 (**7**) command dumps the contents of the math processor registers. If you do not have an 8087 or equivalent math processor chip, this command dumps the contents of the software-emulated registers.

Example

The following example shows and describes the output from the **7** command:

```
cControl 037F  (Closure=Projective  Round=nearest, Precision=64-bit   )
                         IEM=0 PM=1 UM=1 OM=1 ZM=1 DM=1 IM=1
cStatus  6004  cond=1000 top=4 PE=0 UE=0 OE=0 ZE=1 DE=0 IE=0
Tag     A1FF   instruction=59380  operand=59360  op-code=D9EE
Stack          Exp  Mantissa           Value
cST(3) special 7FFF 8000000000000000 = + Infinity
cST(2) special 7FFF 0101010101010101 = + Not A Number
cST(1) valid   4000 C90FDAA22168C235 = +3.141592265110390E+000
cST(0) zero    0000 0000000000000000 = +0.000000000000000E+000
```

Here, the lowercase c that precedes several lines of the output indicates that the coprocessor is in use. If this command had been used with an emulated coprocessor, an e would precede the lines. The following is a line-by-line description of the output from the **7** command:

Line 1

This line shows the value in the control register, 037F. The rest of the line interprets the bits represented by the number in the control register:

- The closure method, which can be projective or affine.
- The rounding method, which can be `nearest` (even), `down`, `up`, or `chop` (truncate to zero).
- The precision, which can be 64, 53, or 24 bits.

Line 2

This line lists the status of the exception-mask bits, described in the following table:

Name	Description
IEM	Interrupt enable
PM	Precision
UM	Underflow
OM	Overflow
ZM	Zero divide
DM	Denormalized operand
IM	Invalid operation

Line 3

This line lists the value in the status register (6004 hexadecimal), the condition code (1000 binary), and the top of stack register (4 decimal). It then lists the exception flags, described in the following table:

Flag	Meaning
PE	Precision
UE	Underflow
OE	Overflow
ZE	Zero divide
DE	Denormalized operand
IE	Invalid operation

Line 4

This line lists the 20-bit address of the tag register, the offset of the instruction, the offset of the operand, and the offset of the op-code, all in hexadecimal. When using software-emulated coprocessor routines, this line does not appear.

Lines 5–9

The rest of the output from the 8087 command lists the contents of the stack register. In this example, `ST(3)` contains the value infinity, `ST(2)` contains a value that cannot be interpreted as any number, `ST(1)` contains a real number, and `ST(0)` contains zero.

: (Delay)

Syntax **:**

Description The Delay (**:**) command interrupts execution of commands from a redirected file and waits about half a second before continuing. You can put multiple Delay commands on a single line to increase the length of delay. The delay is the same length regardless of the processing speed of the computer.

Example In the following example, the Delay (**:**) command is used to slow execution of the redirected file into CodeView.

```
: ;* That was a short delay...
::::: ;* That was a longer delay...
```

< (Redirect CodeView Input)

Syntax *<device*

Parameter *device*
Device or file from which to read commands.

Description The Redirect Input (<) command causes CodeView to read all subsequent command input from a file or device.

Example The following example redirects command input from the file INFILE.TXT to CodeView. Use this method to run “scripts” of CodeView commands that you have prepared in advance.

```
> <INFILE.TXT
```

You can also start up CodeView with redirected input by typing the following at the operating-system prompt:

```
CV /C"<infile.txt"
```

> (Redirect CodeView Output)

Syntax [[**T**]>[[>]] *device*

Parameter *device*
Device or file to which to write output.

Description

The Redirect Output (>) command causes CodeView to write all subsequent command output to a device, such as another terminal, a printer, or a file. The term “output” includes not only output from commands but also the command characters that are echoed as you type them.

The optional **T** indicates that the output should be echoed to the CodeView screen. If you do not give a **T**, CodeView echoes only commands that you enter. Use the **T** option if you are redirecting output to a file to see output from the commands that you are typing.

Note If you are redirecting output to another terminal, you may not want to see the output on the CodeView terminal.

If you specify an existing file, CodeView truncates the file and then starts writing output. To preserve the contents of the file, use a second greater-than symbol (>>), which appends output to the file.

Example

In the following example, output is redirected to the device designated as COM1 (for example, a remote terminal). Enter this command when you are debugging a graphics program and you want CodeView commands to be displayed on a remote terminal while the program display appears on the originating terminal.

```
> >COM1
```

In the following example, output is redirected to the file OUTFILE.TXT. Use this command to keep a permanent record of a CodeView session.

```
> T>OUTFILE.TXT
  . . .
> >CON
  . . .
```

Note The optional **T** is used so that the session is echoed to the CodeView screen as well as to the file. After redirecting some commands to a file, use the command `>CON` to return output to the terminal.

= (Redirect CodeView Input and Output)

Syntax *=device*

Parameter *=device*
Device to which to redirect input and output. Specify `>CON` for the CodeView Command window.

Description The Redirect Input and Output (=) command causes the CodeView debugger to read all subsequent command input from the device and write all subsequent output to the device. You cannot redirect both input and output to a file.

To reset the input and output for CodeView after you have entered one of the other redirection commands, use the command:

```
>= con
```

? (Display Expression)

Syntax **?** *expression*[[**,** *format*]]

Parameters *expression*
The expression to display. This can be any valid CodeView expression.

, *format*
A CodeView format specifier that indicates the format in which to display *expression*.

Description The Display Expression (**?**) command displays the value of a CodeView expression. The simplest form of expression is a symbol representing a single variable or function. An expression may also call functions that are part of the executable file.

The Display Expression command can also set values. For example, with the C or C++ expression evaluator, you can increment the variable `n` by using an assignment expression:

```
? n=n+1
```

The command displays the value after incrementing `n`.

You can specify the format in which the values of expressions are displayed by the Display Expression command. After the expression, type a comma, followed by a CodeView format specifier.

Example

The following example displays the value stored in the variable `amount`, an integer. This value is first displayed in the system radix (in this case, decimal), then in hexadecimal, then in 4-byte hexadecimal, and then in octal.

```
>? amount
500
>? amount,x
01f4
>? amount,lx
000001f4
>? amount,o
764
>
```

?? (Quick Watch)

Syntax

?? *symbol*

Parameter

symbol
Displays the given variable, array, or structure in the Quick Watch dialog box.

Description

The Quick Watch (**??**) command displays the value of any selected expression in the Quick Watch dialog box. You can use Quick Watch to quickly check the value of a variable or structure and expand or contract items in a structure.

Expanding/Contracting Items

The Quick Watch dialog box allows you to:

- Expand or contract nested structures and arrays.
- View variables, structures, or arrays addressed by pointers.
- Add any structure or array to the Watch window.

Expandable items appear with a plus sign (+) in the Quick Watch dialog box. Once expanded, an item appears with a minus sign (-).

Expanding an item has the following effects:

Item	Action
Nested structure	Expands the structure so that the dialog box displays each member of the nested structure.
Pointer	Dereferences the pointer; that is, displays the data that the pointer addresses.

Array	Expands the array so that the dialog box displays each element of the array.

Contracting an item reverses the effects of expanding described above.

Note You can add any expression in the Quick Watch dialog box to the Watch window by choosing the Add Watch button.

Mouse and Keyboard

After opening the Quick Watch dialog, you can expand or contract an item using the following methods:

- Double-click the left mouse button on the item.
- Select the item, then choose the Expand/Contract button at the bottom of the dialog box.
- Use the arrow keys to select the item, and press ENTER.

@ (Redraw)

Syntax

@

Description

The Redraw (**@**) command redraws the CodeView screen. Use this command if the output of the program being debugged disturbs the CodeView display.

\ (Screen Exchange)

Syntax

\

Description

The Screen Exchange (\) command allows you to switch temporarily from the debugging screen to the output screen. The CodeView debugger uses either screen flipping or screen swapping to store the output and debugging screens.

To return to the CodeView screen, press any key.

P A R T 3

Compiling and Linking

Chapter 13 Linking Object Files with LINK 457
Chapter 14 Creating Module-Definition Files 491
Chapter 15 Using EXEHDR 513

CHAPTER 13

Linking Object Files with LINK

This chapter describes the Microsoft Segmented-Executable Linker (LINK) version 5.31. LINK combines compiled or assembled object files into an executable file. This chapter explains LINK's input syntax and fields and tells how to use options to control LINK.

LINK is distributed in the form of LINK.EXE for MS-DOS. LINK is DOS-extended and uses extended memory if available.

When you link for debugging using the /CO option, LINK calls the CVPACK utility. CVPACK version 4.00 must be available on the path. For more information, see "CVPACK" on page 631.

This version of LINK does not support the Microsoft Incremental Linker (ILINK). The LINK options for incremental linking are no longer supported. If /INCR, /PADC, or /PADD is specified, LINK issues a warning and ignores the option.

New Features

This version of LINK has several new or changed features. This section summarizes changes in options. Changes in module-definition statements are discussed in Chapter 14.

The following options are new or changed in LINK versions 5.30 and later. For more information about each option, see "LINK Options," page 471.

/DOSS[[EG]]
: The minimum unique abbreviation for /DOSSEG option has changed from /DO to /DOSS.

/DY[[NAMIC]][[**:***number*]]
: The new /DYNAMIC option lets you change the limit of interoverlay calls in an overlaid MS-DOS program.

/INC[[REMENTAL]]
The /INCR option is no longer supported.

/INF[[ORMATION]]
The /INFO option gives more detailed output. One new use is to get the number of interoverlay calls needed to specify with the /DYNAMIC option.

/M[[AP]][[**:***maptype*]]
The /MAP option has been enhanced. You can get more or less detail in the map file by specifying an optional qualifier.

/NOPACKC[[ODE]]
The minimum unique abbreviation for /NOPACKC has changed from /NOP to /NOPACKC to distinguish it from the new /NOPACKF option.

/NOPACKF[[UNCTIONS]]
The new /NOPACKF option keeps unreferenced packaged functions.

/OL[[DOVERLAY]]
The new /OLDOVERLAY option links an overlaid MS-DOS program using the Static Overlay Manager instead of the MOVE library. This option may not be supported in future versions of LINK.

/ON[[ERROR]]**:**N[[OEXE]]
The /ONERROR:NOEXE option prevents LINK from creating the program output if an error occurs.

/OV[[ERLAYINTERRUPT]]
The minimum unique abbreviation for this option has changed from /O to /OV, to distinguish it from the new /OLDOVERLAY option.

/PACKF[[UNCTIONS]]
The new /PACKF option removes unreferenced packaged functions.

/PADC[[ODE]]
The /PADC option is no longer supported.

/PADD[[ATA]]
The /PADD option is no longer supported.

/PC[[ODE]]
The new /PCODE option tells LINK to call MPC after linking.

/PM[[TYPE]]
The default for the /PM option has changed from **NOVIO** to **PM**.

/r
The new /r option tells the linker not to use extended memory.

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), which is based on the Intel 8086 OMF. LINK uses library files in Microsoft library format.

LINK creates "relocatable" executable files and DLLs. The operating system can load and execute relocatable files in any unused section of memory. LINK can create MS-DOS executable files with up to 1 megabyte of code and data (or up to 16 megabytes when using overlays). It can create segmented-executable files with up to 16 megabytes.

For more information on the OMF, the executable-file format, and the linking process, see the *MS-DOS Encyclopedia.*

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 on EXEHDR, see Chapter 15.

The Microsoft 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 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 the compiler or assembler. The Microsoft Advisor has more information about the compilers and assembler; select Help for the appropriate language from the Command Line box of the Help Contents screen.

Note Unless otherwise noted, all references to "library" in this chapter refer to a static library. This can be either a standard library created by the Microsoft Library Manager (LIB) or an import library created by the Microsoft Import Library Manager (IMPLIB), but not a DLL.

LINK Output Files

LINK can create executable files for MS-DOS or the Windows operating system. The kind of file produced is determined by the way the source code is compiled and the information supplied to LINK. LINK's output is either an executable file or a DLL. For simplicity, this chapter sometimes refers to this output as the "program" or "main output." LINK creates the appropriate file according to the following rules:

- If a .DEF file is specified, LINK creates a segmented-executable file. The type is determined by the EXETYPE and LIBRARY statements.

- If a .DEF file is not specified, LINK creates an MS-DOS program.
- If an overlay number is specifed in a **SEGMENTS** or **FUNCTIONS** statement, LINK creates an overlaid MS-DOS program. This overrides a conflicting .DEF file specification.
- If /DYNAMIC or /OLDOVERLAY is specified, or if parentheses are used in the *objects* field, LINK creates an overlaid MS-DOS program. This overrides a conflicting .DEF file specification.
- If an object file or library module contains an export definition (an EXPDEF record), LINK creates a segmented-executable file. This overrides an overlay specification. The **__export** keyword creates an EXPDEF record in a C program. Microsoft C libraries for protect mode contain EXPDEF records, so linking with a protect-mode default library creates a segmented-executable file.
- If an import library is specified, LINK creates a segmented-executable file.

LINK can also create a "map" file, which lists the segments in the executable file and can list additional information. The /LINE and /MAP options control the content of the map file.

Other options tell LINK to create other kinds of output files. 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 "LINK Options" on page 471.

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 creates a map file or 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 module-definition (.DEF) file.

Fields are separated by commas. You can specify all the fields, or you can leave one or more fields (including *objfiles*) blank; LINK then prompts you for the missing input. (For information on LINK prompts, see "Running LINK" on page 468.) 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 "LINK Options" on page 471.

The fields must be entered in the order shown, whether they contain input or are left blank. Use a space or plus sign (+) to separate multiple filenames in the *objfiles* and *libraries* fields. 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 path. Filenames are not case sensitive. If the filename contains a space (supported on some installable file systems), enclose the name in single or double quotation marks (**'** or **"**).

To interrupt LINK and return to the operating-system prompt, press CTRL+C at any time. Under certain circumstances you may need to press ENTER after CTRL+C.

The next five sections explain how to use each of the LINK fields.

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 (**.**) to 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, then press ENTER, and continue. Do not split a name across lines.

Note Using a special syntax for the *objfiles* field, you can assign the contents of object files to specific overlays in an MS-DOS program. This syntax, described in documentation for earlier versions of LINK, uses parentheses to place one or more object files in an overlay. This syntax may not be supported in future versions of LINK. For more information about overlays, see your high-level language reference documentation.

How LINK Searches for Object Files

When it searches for object 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 ends.
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 that contains the object file.

Load Libraries

If you specify a library in the *objfiles* field, it becomes a "load library." LINK treats a load library like any other object file. It does not search for load libraries in directories named in the *libraries* field. You must specify the library's filename extension; otherwise, LINK assumes an .OBJ extension.

LINK puts every object module from a load library into the executable file, regardless of whether a module resolves an 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 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.
- Debugging so you can call library routines that would not be included in the release version of the program.

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 assigns a default name to the main output. This name is the base name of the first file listed in the *objfiles* field (whether it is an object file or a load library), 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.

The mapfile Field

The *mapfile* field is used to specify a filename for the map file or to suppress the 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. If the field contains a dot (**.**), the map file is given the base name without an extension.
- When using LINK prompts, LINK assumes either the default described previously (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 and other information, use the /MAP option. Both /LINE and /MAP force a map file to be created unless NUL is explicitly specified in *mapfile*.

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 that the specification is for a library file instead of a directory.) 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.

Warning Library names must be unique. If multiple libraries are specified with the same name but different paths, LINK searches only the first library and ignores references in the other libraries.

You might need to specify library names 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 that is not in the current directory and not in a directory specified by the LIB environment variable.

Default Libraries

Most high-level language compilers insert the names of the required language libraries in the object files. (The Microsoft Macro Assembler does not support a default library.) 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 specified in the object file. 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. Therefore, you must use the *libraries* field to specify the alternative libraries that LINK is to search.

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. For more information on LIB, see Chapter 17. For information on IMPLIB, see Chapter 20.

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.

LINK searches object files and libraries for a definition of each external reference. When LINK finds a needed definition in a module in a library, LINK

adds the entire module (but not necessarily all modules in the library) to the program.

You provide a library to LINK in the following ways:

- Specify the name of a library in the *libraries* field.
- Specify the name of a library as a load library in the *objects* field. A load library adds all its modules to the program. For more information, see "Load Libraries" on page 462.
- Compile a program that uses the INCLUDELIB directive to specify the libraries you want linked. For more information, see "Associating Libraries with Modules" in Chapter 8 of the *Programmer's Guide*.
- Compile a program that uses definitions provided in a default library for that compiler. The compiler places a library comment record in the object file. LINK uses the library named in this record.
- Embed a library comment record in the object file by using the comment pragma in a C program. This record precedes a record for a default library placed in the object file by the compiler; therefore, LINK looks in this library before it searches a default library named in the same object file.

LINK first looks for a definition in files specified in the objects field, then it looks in libraries specified in the libraries field. The search order is the order in which the files are specified in the fields. LINK then looks in libraries specified in comment records in the object files, again in the specified order.

If LINK cannot find a needed definition, it issues an error message:

```
unresolved external
```

If a reference is defined in more than one library, LINK uses the first definition it finds as it searches the libraries in order. A duplicate definition may not be a problem if the later definition is in a module that is not linked into the program. However, if the duplicate definition is in a module that contains another needed definition, that module is linked into the program, and the duplicate definition causes an error:

```
symbol defined more than once
```

Multiple definitions can also cause a problem if LINK is using extended dictionaries in libraries. An extended dictionary is a summary of the definitions contained in all modules of a library. LINK uses this summary to speed the process of searching libraries. If LINK finds a previously resolved reference listed in an extended dictionary, it assumes that a duplicate definition exists and issues an error message:

```
symbol multiply defined, use /NOE
```

If this error occurs, link your program using the /NOE option.

How LINK Searches for Library Files

When searching for a library, LINK looks in the following locations in this order:

1. The directory specified for the file, if a path is included. (The default libraries named in object files by Microsoft compilers do not include path specifications.)
2. The current directory.
3. Any directories specified 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 looks in 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:
```

Enter either a path to the library or a path and filename for another library.

If unresolved references remain after LINK searches NEWLIBV3.LIB, it then searches the library C:\MYLIBS\SPECIAL.LIB. If LINK cannot find this library, it prompts you as described previously for NEWLIBV3.LIB. If there are still unresolved references, LINK searches the default libraries.

The deffile Field

Use the *deffile* field to specify a module-definition file. A module-definition file is required for an overlaid MS-DOS program or a DLL. It is optional for a Windows-based application. 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 a *deffile* needs to be specified. If you are linking without a .DEF file, use a semicolon to terminate the command line before the *deffile* field (or accept the default `NUL.DEF` at the `Definitions File` prompt).

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 14.

Examples

The following examples illustrate various uses of the LINK command line.

Example 1

```
LINK FUN+TEXT+TABLE+CARE, , FUNLIST, FUNPROG.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 FUNPROG.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 (**;**) ends 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, , GETPRINT.LST;
```

This command links the files MAIN.OBJ, GETDATA.OBJ, and PRINTIT.OBJ.

No module-definition file is specified, so if the files are assembled Macro Assembler files, LINK creates an MS-DOS real-mode application. If the files are compiled high-level language files, LINK creates an MS-DOS file if the real-mode default combined libraries are provided or a segmented-executable file if the protect-mode libraries are provided. The map file GETPRINT.LST is created.

Example 5

```
LINK GETDATA+PRINTIT, , , , GETPRINT.DEF
```

This command links GETDATA.OBJ and PRINTIT.OBJ, using the information in GETPRINT.DEF. LINK creates a map file named GETDATA.MAP.

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 any 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. Under certain circumstances you may need to press ENTER after CTRL+C.

LINK has five input fields, all optional except one (the *objfiles* field). 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 (preceded by **@**) in place of the expected input.

These methods can be used in combination. The LINK command line is covered on page 460. The following sections explain the other two methods.

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 it is specified.

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 path across lines.

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. You can use response files to hold frequently used options or responses or to overcome the 128-character limit on the length of an MS-DOS command line.

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 after a prompt) 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
```

Contents of the Response File

Each input field must appear on a separate line, or 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 it is 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 preceding response file 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. (Since options can be entered anywhere in any field, /MAP is in the field that would otherwise be blank to accept the .EXE executable-file extension.)
- 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.

LINK Options

This section explains how to use options to control LINK's behavior and modify LINK's output. It contains a brief introduction on how to specify options followed by a description of each option.

Specifying Options

The following paragraphs discuss rules for using options.

Syntax

All options begin with a slash (/). (A dash, –, is not a valid option specifier for LINK.) You can specify an option with its full name or an abbreviation, up to the shortest sequence of characters that uniquely identifies the option (except for /DOSSEG). 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 (except for /r), so you can also specify /batch or /Batch. This chapter uses meaningful yet legal forms of the option names. If an option is followed by a colon (**:**) and an argument, no spaces can appear before or after the colon.

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 "Setting Options with the LINK Environment Variable" on page 488.) 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.

Numeric Arguments

Some LINK options take numeric arguments. You can enter numbers either in decimal format or in standard C-language notation.

The /ALIGN Option

Option /A[LIGNMENT]**:***size*

The /ALIGN option aligns segments in a segmented-executable file at the boundaries specified by *size*. LINK ignores /ALIGN for MS-DOS programs.

The alignment size is in bytes and must be an integer power of two. LINK rounds up to the next power of two if another value is specified. The default alignment is 512 bytes.

This option reduces the size of the file as it is stored on disk by reducing the size of gaps between segments. It has no effect on the size of the file when loaded in memory. The size of an executable file is limited to 64K times the alignment.

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 path whenever it cannot find a library 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.

The /CO Option

Option /CO[[DEVIEW]]

The /CO option adds Microsoft Symbolic Debugging Information to the executable file. Debugging information can be used with the Microsoft CodeView debugger. If the object files do not contain debugging information (that is, if they were not compiled or assembled using either /Zi or /Zd), this option places only public symbols in the executable file.

You can run the resulting executable file outside CodeView; the debugging data in the file is ignored. However, it increases file size. 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, debugging 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 MS-DOS executable files.

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. MS-DOS 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 for MS-DOS programs.

LINK normally requests MS-DOS to set the maximum number of paragraphs to 65,535. Since this is more memory than MS-DOS can supply, MS-DOS always denies the request and allocates the largest contiguous block of memory it can find. If the /CPARM option is used, MS-DOS 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 the minimum value. This minimum 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. For more information on EXEHDR, see Chapter 15.

The /DOSSEG Option

Option /DOSS[EG]

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, the /DOSSEG option defines the following two labels:

```
__edata = DGROUP : BSS
__end   = DGROUP : STACK
```

The variables `__edata` and `__end` have special meanings for Microsoft compilers. It is recommended that you do 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); unassigned pointers point to this area. 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 Microsoft Macro Assembler (MASM) directive **.DOSSEG**.

Note The minimum abbreviation allowed for this option is /DOSS.

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 MS-DOS .EXE files.

The /DYNAMIC Option

Option /DY[NAMIC]**:***number*

The /DYNAMIC option changes the limit on the number of interoverlay calls in an overlaid MS-DOS program. (For more information on overlays, see your high-level language documentation.) The default limit is 256. The *number* is a decimal integer from 1 to 10,922. Specify a higher *number* to raise the limit if LINK issues the error `too many interoverlay calls`. Lower the limit to create a smaller table of interoverlay calls, saving space in your program.

To determine the most efficient *number*, run LINK using the /INFO option. The displayed information contains the line

`NUMBER OF INTEROVERLAY CALLS: requested` *number*`; generated` *calls*

The *number* of interoverlay calls requested is the *number* set by /DYNAMIC or the default of 256. The *calls* number reports the number of interoverlay calls actually generated. For maximum efficiency, run LINK using /INFO, then relink using /DYNAMIC**:***calls*.

The /EXEPACK option is ignored for overlaid programs.

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 issues a warning 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 MS-DOS files with CodeView because the unpacker that /EXEPACK appends to an MS-DOS program is incompatible with CodeView. In a Windows-based program, the unpacker is in the loader, and there is no conflict with CodeView.

The /EXEPACK option is not compatible with the /HIGH or /Q option. LINK ignores the /EXEPACK option for overlaid programs.

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 /FARCALL option is on by default for overlaid MS-DOS programs and programs created with the /TINY option. It is off by default for other programs. If an environment variable (such as LINK or CL) includes /FARCALL, you can use the /NOFARCALL option to override it.

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 because 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 5 bytes replace the 5-byte far-call instruction.

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 segment, 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.

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. Object modules produced by Microsoft high-level languages are safe from this problem because little immediate data is stored in code segments.

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.

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 MS-DOS to place the executable file as high as possible in memory. Without the /HIGH option, MS-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 MS-DOS .EXE files.

The /INFO Option

Option /I[NFORMATION]

The /INFO option displays to the standard output information about the linking process, including the phase of linking, the object files being linked, and the

library modules used. This option is useful for determining the locations of the object files and modules, the number of segments, and the order in which they are linked.

An important use of /INFO is to get the number of interoverlay calls generated. You can then specify this number with the /DYNAMIC option.

The /LINE Option

Option /L[[INENUMBERS]]

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 ML, FL, and CL to add line numbers to the object file.) If you also want to add public symbols or other information to the map file, use the /MAP option. For more information on the map file, see the description of /MAP.

The /LINE option causes a map file to be created even if you did not explicitly tell the linker to do so. LINK creates a map file when a filename is specified in the *mapfile* field or when the default map-file name is accepted. (The /MAP option also forces creation of a map file.) For more information, see the description of *mapfile* on page 463.

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-file name in the *mapfile* field or in response to the `List File` prompt.

The /MAP Option

Option /M[[AP]][[**:**{*maptype*}]]

The /MAP option controls the information contained in the map file. The /MAP option causes a map file to be created even if you did not explicitly tell the linker to do so.

LINK creates a map file when a filename is specified in the *mapfile* field or when the default map-file name is accepted. (The /LINE option also forces creation of a map file.) For more information, see the description of *mapfile* on page 463.

A map file by default contains only a list of segments. A map file created with /MAP contains public symbols sorted by name and by address, in addition to the segments list. Symbols in C++ appear in the form of decorated names. To add or omit information, specify /MAP followed by a colon (**:**) and a *maptype* qualifier:

A[DDRESS]
: Omits the list of public symbols sorted by name.

F[ULL]
: Adds information about each object file's contribution to a segment. Adds undecorated names following the decorated names for C++ symbols in the listings by name and by address.

If you also want to add line numbers to the map file, use the /LINE option.

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.

The /NOD Option

Option /NOD[EFAULTLIBRARYSEARCH][**:***libraryname*]

The /NOD option tells LINK not to search default libraries named in object files. Specify *libraryname* to tell LINK to exclude only *libraryname* from the search.
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*. For more information, see "Default Libraries" on page 464.

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. If you compile a program using Microsoft C 7.0 or later and you specify /NOD, you must also specify OLDNAMES.LIB.

The /NOE Option

Option /NOE[XTDICTIONARY]

The /NOE option prevents the linker from searching extended dictionaries when resolving references. An extended dictionary is a list of symbol locations in a library created with LIB. The linker consults extended dictionaries to speed up library searches. Using /NOE slows the linker.

When LINK uses extended dictionaries, it gives an error if a duplicate definition is found. Use this option when you redefine a symbol and an error occurs. For more information, see "How LINK Resolves References" on page 464.

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.

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 early versions of LINK and Microsoft compilers. This option is valid only for assembly-language programs that create MS-DOS programs.

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.

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.

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. Therefore, you can use /NONULLS to override the DOSSEG comment record found in run-time libraries. This option is for segmented-executable files.

The /NOPACKC Option

Option /NOPACKC[[ODE]]

This option turns off code-segment packing. Code-segment packing is on by default for segmented-executable files and for MS-DOS programs created with overlays or with the /TINY option. It is off by default for other MS-DOS programs. If an environment variable (such as LINK or CL) includes the /PACKC option to turn on code-segment packing, you can use /NOPACKC to override /PACKC. For more information on packing, see “The /PACKC Option” on page 482.

Note The minimum unique abbreviation for /NOPACKC has changed from /NOP to /NOPACKC.

The /NOPACKF Option

Option /NOPACKF[[UNCTIONS]]

This option prevents the removal of unreferenced packaged functions. Removal of such definitions (the /PACKF option) is usually on by default. Use /NOPACKF to preserve these definitions. For example, you may want to keep unreferenced code in a debugging version of your program. For more information on /PACKF and packaged functions, see page 484.

The /OLDOVERLAY Option

Option /OL[[DOVERLAY]]

This option links an overlaid MS-DOS program using the Static Overlay Manager instead of the MOVE library. This option may not be supported in future versions
of LINK. For information about overlays, see your high-level language documentation.

The /ONERROR Option

Option /ON[[ERROR]]**:**N[[OEXE]]

The /ONERROR option tells LINK what to do if an error occurs. By default, if certain errors occur, LINK writes an executable file to disk and overwrites any existing file having the same name. The resulting executable file has the error bit set in its header. Specify /ONERROR**:**NOEXE to prevent such a file from being written to disk and preserve any existing file having the same name. The /ONERROR option can be useful in makefiles.

The /OV Option

Option /OV[[ERLAYINTERRUPT]]**:***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 overlaid MS-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 MS-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 reserves interrupt 63 for some other purpose.

Note The minimum unique abbreviation for /OV has changed from /O to /OV.

The /PACKC Option

Option /PACKC[[ODE]][[**:***number*]]

The /PACKC option turns on code-segment packing. Code-segment packing is on by default for segmented executable files and for MS-DOS programs created with overlays or with the /TINY option. It is off by default for other MS-DOS programs. You can use the /NOPACKC option to override /PACKC.

The linker packs physical code segments by grouping neighboring logical 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*. It then 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.

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.

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
```

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. It then 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 only for segmented-executable files. It might be necessary to use the /PACKD option to get around the limit of 254 physical data segments per executable file imposed by an operating system. 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.

The /PACKF Option

Option /PACKF[[UNCTIONS]]

The /PACKF option removes unreferenced “packaged functions.” This behavior is the default. However, if an environment variable (such as LINK or CL) includes the /NOPACKF option, you can use /PACKF to override /NOPACKF.

A packaged function is visible to the linker in the form of a COMDAT record. Packaged functions are created when you use the /Gy option on the CL command line (or, in PWB, when you choose Enable Function Level Linking in the Additional Global Options dialog box, which is available from the C or C++ Compiler Options dialog boxes). Member functions in a C++ program are automatically packaged.

If a packaged function is defined but not called, this option removes the function definition from the executable file. /PACKF is not recursive; LINK does not remove any external definitions brought in by the unused packaged function. For more information about packaged functions, see your C or C++ language documentation.

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 compatibilty 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, end 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 "LINK Temporary Files" on page 489.

The /PCODE Option

Option /PC[ODE]

This option tells LINK to call the Make P-Code (MPC) utility after linking. MPC is included with Microsoft C/C++ versions 7.0 and later. If you link a p-code program using LINK instead of CL, you must use /PCODE to generate a valid executable file. The /PCODE option is not compatible with overlays.

The /PM Option

Option /PM[TYPE]**:***type*

This option specifies the type of Windows-based 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:

PM
The default. Windows-based application. The application uses the API provided by the Windows operating system and must be executed within the Windows operating system. This is equivalent to **NAME WINDOWAPI**.

VIO
Character-mode application to run in a text window in the Windows operating system session. This is equivalent to **NAME WINDOWCOMPAT**.

NOVIO
Character-mode application that must run full screen within the Windows operating system. This is equivalent to **NAME NOTWINDOWCOMPAT**.

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 because 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.

The /r Option

Option /r

Prevents LINK from using extended memory with MS-DOS. The /r option must appear first in the options field on the command line and cannot appear in a response file or the LINK environment variable. LINK.EXE is extender-ready and uses extended memory if it exists. This option forces LINK to use only conventional memory. The option name is case sensitive.

For LINK to run in DOS-extended mode, sufficient extended memory must be available. The memory must be provided by one of the following:

- An MS-DOS Protected-Mode Interface (DPMI) server, such as that provided in an MS-DOS session within the Windows operating system operating in enhanced mode
- A Virtual Control Program Interface (VCPI) server, such as Microsoft’s EMM386.EXE
- An XMS driver, such as Microsoft’s HIMEM.SYS

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 decimal 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 that 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.

The /STACK Option

Option /ST[ACK]**:***number*

The /STACK option lets you change the stack size from its default value of 2048 bytes. The *number* is any positive even value in decimal or C-language notation up to 64K – 2 bytes. If an odd number is specified, LINK rounds up to the next even value. Do not specify /STACK for a DLL.

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. For more information on EXEHDR, see Chapter 15.

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. LINK uses /FARCALL and /PACKC when /TINY is specified.

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-based programs cannot be converted to a .COM format.

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-based linker (LINK4) performed fixups without this displacement. This option is for linking segmented-executable files.

The /? Option

Option /?

The /? option displays a brief summary of LINK command-line syntax and options.

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 ML, FL, and CL also use the options in the LINK environment variable.)

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 values for LINK's input fields; filenames in the LINK variable cause an error.

Example

```
SET LINK=/NOI /SEG:256 /CO
LINK TEST;
LINK /NOD PROG;
```

In the preceding example, 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`.

Behavior of the LINK Environment Variable

You can specify options in the LINK input fields and in the LINK environment variable. LINK reads the options set in the LINK environment variable before it reads options specified in LINK input fields. This priority has the following effects:

- The option LINK considers to be first is the first one in the LINK environment variable, if set. The /NOLOGO option behaves differently depending on whether or not it is first. However, the /r option cannot be specified in the LINK variable and must be specified first on the command line.
- An option specified multiple times with different values will get the last value read by LINK. For example, if /SEG:512 is set in an input field, it overrides a setting of /SEG:256 in the LINK variable.
- For some options, if an option appears in the LINK variable and a conflicting option appears in an input field, the input-field option overrides the environment-variable option. For example, the input-field option /NOPACKC overrides the environment-variable option /PACKC.

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.

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 following message:

```
Temporary file tempfile has been created.
Do not change diskette in drive, letter.
```

In the preceding message, *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 with MS-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.

With MS-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.

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.

CHAPTER 14

Creating Module-Definition Files

This chapter describes the contents of a module-definition (.DEF) file. It begins with a brief overview of the purpose of .DEF files. The rest of the chapter discusses each statement in a module-definition file and describes syntax rules, arguments, fields, attributes, and keywords for each statement.

The statements in this chapter are supported by the following utilities:

- Microsoft Segmented-Executable Linker (LINK) version 5.31
- Microsoft Import Library Manager (IMPLIB) version 1.40

New Features

The latest version of the linker and other utilities support the statements and keywords described in this chapter. The following sections introduce features that are new with these versions.

MS-DOS Programs

You can now use a module-definition file when you link an MS-DOS application. LINK creates an MS-DOS executable file instead of a segmented-executable file if the .DEF file contains any of the following:

- An **EXETYPE** statement that specifies the type **DOS**
- A **SEGMENTS** statement that specifies an overlay number
- A **FUNCTIONS** statement that specifies an overlay number

Other conditions also determine the type of executable file that LINK creates; for details, see “LINK Output Files” on page 459. The only valid statements in a .DEF file for an MS-DOS program are **EXETYPE**, **SEGMENTS**, **FUNCTIONS**, and **INCLUDE**. All other statements are ignored.

Statements

Following are the new statements and changes to existing statements described in this chapter. For details on each statement, see the reference section of this chapter.

- The **NAME** statement's default *apptype* is now **WINDOWAPI** (formerly **NOTWINDOWCOMPAT**).
- The **EXETYPE** statement's default is now **WINDOWS**.
- **EXETYPE WINDOWS** now assumes **PROTMODE** by default.
- The **EXETYPE** statement has a new type argument, **DOS**.
- The new **SECTIONS** and **OBJECTS** keywords are synonyms for the **SEGMENTS** statement.
- The new **INCLUDE** statement inserts module statements from a separate text file.
- The new **FUNCTIONS** statement specifies the order in which functions appear in the executable file. It can also assign functions to a specific segment. In an overlaid MS-DOS program, **FUNCTIONS** can specify the overlay in which functions belong.
- The **SEGMENTS** statement accepts a new argument, **OVL**:*number*. This argument specifies the overlay in which the segment belongs in an overlaid MS-DOS program.

Overlays

A new overlay manager, the Microsoft Overlay Virtual Environment (MOVE), replaces the Microsoft Static Overlay Manager. MOVE is supported by high-level programming languages, such as Microsoft C/C++. For information on creating overlaid MS-DOS programs using MOVE, refer to your high-level language reference documentation.

Overview

A module-definition (.DEF) 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). This file should be used for DLLs and overlaid DOS programs. It is optional (but desirable) for other segmented executable files, such as Windows-based applications, and is usually not necessary for other MS-DOS programs.

You use module-definition files in the following two situations:

- You can specify a module-definition file in LINK's *deffile* field. The module-definition file gives the LINK utility the information that is necessary for linking the program. For specific information on using a .DEF file when linking, see page 466.
- You can use the Microsoft Import Library Manager utility (IMPLIB) to create an import library from a module-definition file for a DLL (or from the DLL created by a module-definition file). You then specify the import library in LINK's *libraries* field when linking an application that uses functions and data in the DLL. For information on IMPLIB, see page 652.

Note The term "function" as used in this chapter refers to any routine for the programming language being used: function, procedure, or subroutine.

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 14.1 summarizes the purpose of the module statements and shows the order in which they are discussed in this chapter.

Table 14.1 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 an MS-DOS executable file to the beginning of the file
APPLOADER	Replaces the default Windows operating system loader with a custom loader
EXETYPE	Identifies the target operating system
PROTMODE	Specifies a protected-mode Windows-based program
REALMODE	Specifies a real-mode Windows-based program
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

Table 14.1 Module Statements (*continued*)

Statement	Purpose
OLD	Preserves ordinals from a previous DLL
EXPORTS	Defines exported functions
IMPORTS	Defines imported functions
FUNCTIONS	Specifies function order and location
INCLUDE	Inserts a file containing module statements

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. User-specified identifiers are case insensitive by default; however, they can be made case sensitive by using LINK's (or IMPLIB's) /NOI option.
- Use one or more spaces, tabs, or newline characters to separate a statement keyword from its arguments and to separate statements from each other. A colon (**:**) or equal sign (**=**) that designates an argument is surrounded by zero or more spaces, tabs, or newline characters.
- 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 the statement is repeated with a different specification later in the file, the later statement overrides the earlier one.
- The **INCLUDE** statement can appear more than once in the file. Each statement takes one filename specification.
- The **SEGMENTS**, **EXPORTS**, **IMPORTS**, and **FUNCTIONS** statements can appear more than once in the file. Each statement can take multiple specifications, which must be separated by one or more spaces, tabs, or newline characters. 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 it can appear between lines of a multiline statement.
- Numeric arguments can be specified in decimal or in C-language notation.

- If a string argument matches a reserved word it must be enclosed in double quotation marks (**"**). Reserved words are listed at the end of this chapter.

Example

The following module-definition file gives a description for a DLL. This sample file includes one comment and five statements.

```
; Sample module-definition file
LIBRARY  FIRSTLIB WINDOWAPI
EXETYPE  WINDOWS 3.0
CODE     PRELOAD MOVABLE DISCARDABLE
DATA     PRELOAD SINGLE
HEAPSIZE 1024
```

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. If **NAME** is specified, the **LIBRARY** statement cannot be used. If neither is used, the default is **NAME**, and LINK creates an application. The **NAME** or **LIBRARY** statement must precede all other statements.

Syntax

NAME [[*appname*]] [[*apptype*]] [[**NEWFILES**]]

Remarks

The arguments can appear in any order.

If *appname* is specified, it becomes the name of the application as it is known by the operating system. This name can be any valid filename. If *appname* contains a space (allowed under some installable file systems), begins with a nonalphabetic character, or is a reserved word, enclose *appname* in double quotation marks. The name cannot exceed 255 characters (not including the 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* is one of the following keywords:

WINDOWAPI
The default. Creates a Windows-based application. The application uses the API provided by the Windows operating system and must be executed within the the Windows operating system. This is equivalent to the LINK option /PM:PM.

WINDOWCOMPAT
Creates a character-mode application to run in a text window within a Windows operating system session. This is equivalent to the LINK option /PM:VIO.

NOTWINDOWCOMPAT
Creates a character-mode application that must run full screen and cannot run in a text window within the Windows operating system. This is equivalent to the LINK option /PM:NOVIO.

The **NEWFILES** keyword sets a bit in the file header to notify the loader that the application may be using an installable file system. The synonym **LONGNAMES** is supported for compatibility.

Example

The following example assigns the name `calendar` to an application that can run in a text window within the Windows operating system:

```
NAME calendar WINDOWCOMPAT
```

The LIBRARY Statement

The **LIBRARY** statement identifies the executable file as a DLL. It can also specify the .DLL filename. The **LIBRARY** or **NAME** 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*]] [[**PRIVATELIB**]]

Remarks

The arguments can appear in any order.

If *libraryname* is specified, it becomes the base name of the .DLL file. This name can be any valid filename. LINK assumes a .DLL extension whether or not an extension is specified. If *libraryname* contains a space (allowed under some installable file systems), begins with a nonalphabetic character, or is a reserved word, enclose it in double quotation marks ("). The name cannot exceed 255 characters.

The *libraryname* filename overrides a name specified in LINK's *exefile* field.

Specify **PRIVATELIB** to tell the Windows operating system that only one application may use the DLL.

Example

The following example assigns the name `calendar` to the DLL being defined.

```
LIBRARY calendar
```

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 alternate 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 EXEHDR utility. The string appears in the `Description:` field. For more information, see Chapter 15.

The **DESCRIPTION** statement is different from a comment. A comment is a line that begins with a semicolon (**;**). LINK does not place comments into the program.

Example

The following example inserts the text `Tester's Version, Test "A"`, which contains a literal single quotation mark and a pair of literal double quotation marks, into the application or DLL:

```
DESCRIPTION "Tester's Version, Test ""A"""
```

The STUB Statement

The **STUB** statement adds an MS-DOS executable file to the beginning of a segmented executable file. The stub is invoked whenever the file is executed under MS-DOS. Usually, the stub displays a message and terminates execution. However, the program can be of any size and may perform other actions. The **STUB** statement is optional; by default, LINK adds a standard stub that is appropriate for the program's **EXETYPE**.

Syntax

STUB {'*filename*' | **NONE**}

Remarks

The *filename* specifies the MS-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. If you specify a path with *filename*, LINK looks only in that location. The *filename* must be enclosed in 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. However, the resulting file will hang the system if loaded under MS-DOS.

Example

The following example inserts the MS-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 MS-DOS.

The APPLOADER Statement

The **APPLOADER** statement tells the linker to replace the default Windows operating system loader with a custom loader. Use **APPLOADER** when you want your Windows-based program to be loaded by a different loader from the one the Windows operating system calls automatically at load time. This statement applies only to Windows-based programs.

Syntax

APPLOADER [['']]*loadername*[['']]

Remarks

The *loadername* is an identifier for an externally defined loader. The name is optionally enclosed in single or double quotation marks (' or "). The identifier is an external reference that must be resolved at link time in an object file or static library. It is not case sensitive unless the /NOI option is used with the linker.

When **APPLOADER** appears in a module-definition file, LINK sets a bit in the header of the executable file to tell the Windows operating system that a custom loader is present. At load time, the Windows operating system loads only the first segment of the program and transfers control to that segment.

At link time, LINK creates a new logical segment called **LOADER_***loadername* and makes it the first physical segment of the program. LINK places the *loadername* function in this segment. Nothing else is contained in **LOADER_***loadername*; the /PACKC option does not affect this segment.

Example

The following statement replaces the default loader with a loader called `__MSLANGLOAD`, which is defined in the Microsoft FORTRAN run-time libraries:

```
APPLOADER __MSLANGLOAD
```

Windows-based programs that use huge arrays will fail unless loaded by the custom loader provided in the default FORTRAN libraries. This statement appears in the default .DEF file used for FORTRAN QuickWin programs.

The EXETYPE Statement

The **EXETYPE** statement specifies under which operating system the program is to run. This statement provides an additional degree of protection against the program's being run under an incorrect operating system.

Syntax

EXETYPE [[*descriptor*]]

Note For an overlaid MS-DOS program, LINK assumes **EXETYPE DOS**; in that case, this statement is not required. For information on creating an overlaid program, see your high-level language reference documentation.

Remarks

The *descriptor* sets the target operating system. **EXETYPE** sets bits in the header that can be checked by operating systems. The *descriptor* argument is one of the following keywords:

WINDOWS [[*version*]]
: The default. Creates a Windows-based program. If a **STUB** statement is not specified, **WINDOWS** changes the default message to one that is the same as is provided in WINSTUB.EXE. The *version* is optional; for a description, see the next section, "Windows-Based Programming."

DOS
: Creates a nonsegmented executable file. For information on how LINK determines the type of executable file, see "LINK Output Files" on page 459.

UNKNOWN
: Creates a segmented executable file but sets no bits in the header.

Windows-Based Programming

The **WINDOWS** descriptor takes an optional version number. The Windows operating system reads this number to determine the minimum version of the Windows operating system needed to load the application or DLL. For example, if 3.0 is specified, the resulting application or DLL can run under the Windows operating system versions 3.0 and later. 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-based programming, use the **EXETYPE** statement with a **REALMODE** statement to specify a Windows-based application or DLL that runs under either real- or protected-mode.

Example

The following statement combination defines an application that runs under the Windows operating system version 3.0 in any mode:

```
EXETYPE WINDOWS 3.0
REALMODE
```

The PROTMODE Statement

The **PROTMODE** statement specifies that the application or DLL runs only under the Windows operating system in protected mode (either standard mode or 386 enhanced mode). **PROTMODE** lets LINK optimize to reduce both the size of the file on disk and its loading time. **PROTMODE** is assumed by **EXETYPE WINDOWS**. To define a program that runs under any Windows operating system mode, specify **REALMODE. Note that PROTMODE** and **REALMODE** are not legal statements if you have set **EXETYPE** to **DOS.**

Syntax **PROTMODE**

The REALMODE Statement

The **REALMODE** statement specifies that the application runs under the Windows operating system in either real mode or protected mode. By default, **EXETYPE WINDOWS** assumes **PROTMODE**. **PROTMODE** and **REALMODE** are not legal statements if you have set **EXETYPE** to **DOS.**

Syntax **REALMODE**

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 of these are specified, the **STACKSIZE** statement overrides the /STACK option. Do not specify the **STACKSIZE** statement for a DLL.

Syntax **STACKSIZE** *number*

Remarks The *number* must be a positive integer, in decimal or C-language notation, up to 64K (minus 2 bytes). If an odd number is specified, LINK rounds up to the next even value.

Example The following example allocates 4096 bytes of stack space:

```
STACKSIZE 4096
```

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 that no local heap is created.

Syntax

HEAPSIZE {*bytes* | **MAXVAL**}

Remarks

The *bytes* argument is 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.

Example The following example sets the local heap to 4000 bytes:

```
HEAPSIZE 4000
```

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** [[*attributes*]]

Remarks The *attributes* are one or more optional attributes: *discard*, *executeonly*, *load*, *movable*, and *shared*. Each can appear once, in any order. These attributes are described in "CODE, DATA, and SEGMENTS Attributes" on page 503.

Example The following example sets defaults for the program's code segments:

```
CODE PRELOAD MOVABLE DISCARDABLE
```

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 The *attributes* are one or more optional attributes: *instance*, *load*, *movable*, *readonly*, and *shared*. Each can appear once, in any order. These attributes are described in "CODE, DATA, and SEGMENTS Attributes" on page 503. By default, all data segments have the following attributes:

```
SHARED LOADONCALL READWRITE FIXED
```

Example The following example defines the application's data segment so that it cannot be shared by multiple copies of the program and it 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
```

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). 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 a section of segment definitions. Multiple definitions must be separated by one or more spaces, tabs, or newline characters. The **SEGMENTS** statement must appear once before the first definition (on the same or preceding line) and can be repeated before each additional definition. **SEGMENTS** statements can appear more than once in the file.

Syntax

SEGMENTS
[['] *segmentname* [[']] [[**CLASS** '*classname*']] [[*attributes*]]

or

[['] *segmentname* [[']] [[**CLASS** '*classname*']] [[**OVL:***overlaynumber*]]

Remarks

Each segment definition begins with *segmentname* and is 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**.

The *attributes* field applies to segmented executable files. This field accepts one or more optional attributes: *discard*, *executeonly*, *load*, *movable*, *readonly*, and *shared*. Each can appear once, in any order. These attributes are described in the next section, "CODE, DATA, and SEGMENTS Attributes." LINK ignores *attributes* if **OVL** is specified.

The **OVL** keyword tells LINK to create an MS-DOS program that contains overlays. If **OVL** is used, LINK assumes **EXETYPE DOS**. An alternate keyword is **OVERLAY**. The *overlaynumber* specifies the overlay in which the segment is to be placed. The value 0 represents the root, and positive decimal numbers through 65,535 represent overlays. By default, a segment is assigned to the root. For more information on overlays, see your high-level language reference documentation.

Example

The following example specifies segments named `cseg1`, `cseg2`, and `dseg`. The first segment is assigned the class `mycode`; the second is assigned **CODE** by default. Each segment is given different attributes.

```
SEGMENTS
    cseg1 CLASS 'mycode'
    cseg2                EXECUTEONLY PRELOAD
    dseg  CLASS 'data'   LOADONCALL READONLY
```

CODE, DATA, and SEGMENTS Attributes

The following attributes apply to the **CODE**, **DATA**, and **SEGMENTS** statements described previously. Refer to "Remarks" in each of the previous sections for the attributes used by each statement. Most attributes are used by all three statements; others are used as noted. Attributes can appear in any order.

Listed with each attribute are keywords that are legal values for that attribute. The attributes and keywords are described, and 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.

discard

{**DISCARDABLE** | **NONDISCARDABLE**}

Used 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. The default is **NONDISCARDABLE**.

executeonly

{**EXECUTEONLY** | **EXECUTEREAD**}

Used 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**}

Used 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.

This attribute and the *shared* attribute interact for data segments. The *shared* attribute has the keywords **SHARED** and **NONSHARED**. If `DATA SHARED` is specified, LINK assumes **SINGLE**; if `DATA NONSHARED` is specified, LINK assumes **MULTIPLE**. Similarly, `DATA SINGLE` forces **SHARED**, and `DATA MULTIPLE` forces **NONSHARED**.

load

{**PRELOAD** | **LOADONCALL**}

Used for **CODE**, **DATA**, and **SEGMENTS** statements. Determines when a segment is loaded.

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**}

Used for **CODE**, **DATA**, and **SEGMENTS** statements. Determines whether a segment can be moved in memory. This attribute is valid only for a Windows-based DLL or a real-mode Windows-based application. **FIXED** is the default. An alternative spelling for **MOVABLE** is **MOVEABLE**.

readonly

{**READONLY** | **READWRITE**}

Used 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**}

Used for real-mode Windows operating system sessions only. Determines whether all instances of the program can share **EXECUTEREAD** and **READWRITE** segments.

SHARED (the default for DLLs) specifies that one copy of the segment is loaded and shared among all processes that access 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**.

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 following sections 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. If an export in the DLL was specified with the **NONAME** attribute, the exported name is not available and its ordinal cannot be preserved. 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 ").

The EXPORTS Statement

The **EXPORTS** statement defines the names and attributes of the functions and data made available to other applications and DLLs. It also defines the names and attributes 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 a section of export definitions. Multiple definitions must be separated by one or more spaces, tabs, or newline characters. 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 **__export** keyword makes a function available from a DLL.

Syntax

EXPORTS
entryname[[**=***internalname*]] [[**@***ord*[[*nametable*]]]] [[**NODATA**]] [[**PRIVATE**]]

Remarks

The *entryname* defines the function or data-item name as it is known to other programs. If the function or data item is in a C++ module, the *entryname* must be specified as a decorated name. For specific information on decorated names, see your C++ language documentation.

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*. The use of *ord* is faster and can save space.

The *nametable* is one of two optional keywords that determine what happens to *entryname*. By default, with or without *ord*, the *entryname* is placed in the nonresident names table. If the *ord* number is followed by **RESIDENTNAME**, the name is placed in the resident names table. If **NONAME** is specified after *ord*, the *entryname* is discarded from the DLL being created, and the item is exported only by ordinal.

The optional keyword **NODATA** specifies that there is no static data in the function.

The optional keyword **PRIVATE** tells IMPLIB to ignore the definition. **PRIVATE** prevents *entryname* from being placed in the import library. The keyword has no effect on LINK.

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.

```
EXPORTS
    SampleRead = read2bin @8
    StringIn   = str1     @4 RESIDENTNAME
    CharTest
```

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 a section of import definitions. Multiple definitions must be separated by one or more spaces, tabs, or newline characters. 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* argument 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.) If the function or data item is in a C++ module, *entry* must be specified as a decorated name. For information on decorated names, see your C++ language documentation.

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* argument, the symbol has the

same name in all three contexts. If either of the programs uses the *internalname* argument, the symbol may have more than one distinct name.

Example

The **IMPORTS** statement that follows 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
```

The FUNCTIONS Statement

The **FUNCTIONS** statement places high-level language functions in a specified physical order and also assigns them to segments or overlays. For more information on overlays, see your high-level language reference documentation.

Syntax

FUNCTIONS[[**:**{*segmentname* | *overlaynumber*}]]
functionname

Remarks

The **FUNCTIONS** keyword marks the beginning of a section of functions. **FUNCTIONS** statements can appear more than once in the .DEF file.

FUNCTIONS can be followed by a colon (**:**) and a destination specifier, which is either *segmentname* or *overlaynumber*.

The *segmentname* specifies a defined segment in which a function is to be placed. The *segmentname* does not have to be previously defined in a **SEGMENTS** statement. LINK assumes the segment definition, using the class **CODE**; a later **SEGMENTS** statement can redefine the segment.

The *overlaynumber* specifies the overlay in which a function is to be placed. Valid overlay numbers are from 0 through 65,535. The number 0 represents the root.

The *functionname* is the identifier for a “packaged function.” A packaged function is visible to the linker in the form of a COMDAT record. To compile a C function as a packaged function, use the /Gy option on the CL command line (or, in [name of IDE], choose Enable Function-Level Linking). Only packaged functions can be specified in a **FUNCTIONS** statement. You can specify one or more function names, separated by one or more spaces, tabs, or newline characters. If the function is in a C++ module, *functionname* must be specified as a decorated name. For specific information on decorated names, see your C++ language documentation.

Ordering Functions

You can use **FUNCTIONS** to specify a list of ordered functions. LINK places ordered functions into a segment in the physical order that you specify before unordered functions in the same segment. You can let LINK choose the

segment, or you can specify the segment. If LINK makes the decision, it places ordered functions in segments called COMDAT_SEG*n*, where *n* is one of a sequence of numbers beginning with 0. As LINK places ordered functions in these segments, it creates a new segment when the current one reaches 64K (minus 36 bytes). You can specify the destination segment in one of two ways:

- Specify the segment using "explicit allocation." In explicit allocation, a function is assigned to a segment at compile time, either in the source code or when compiling. In C source code, you can use the **__based** keyword (or its predecessor, the **alloc_text** pragma) to specify the segment where an individual function is to reside. When compiling with the CL compiler, you can use the /NT option to specify the segment where all functions in an object file are to reside. A function not explicitly allocated to a segment is sometimes referred to as an anonymous function.
- Specify the segment after the **FUNCTIONS** keyword. The segment must already have been defined, either in a **SEGMENTS** statement or at compile time. An explicitly allocated function cannot be placed in a different segment from the one to which it was allocated.

LINK accumulates multiple specifications and treats them as one list of ordered functions. If segments or overlays are specified, LINK accumulates the functions with other functions that have the same destination.

The following statement places three functions in a specified order within the segment called `MySeg`:

```
FUNCTIONS:MySeg
    Func1
    Func2
    Func3
```

Creating Overlays

You can use **FUNCTIONS** to place a packaged function in an overlay. By default, a function is assigned to the root.

If a function is explicitly allocated (see the previous section), it can be placed in an overlay only if its segment and any other functions in that segment are not also assigned to an overlay. In this case, the **FUNCTIONS** statement implicitly assigns the entire segment to the specified overlay. An explicitly allocated function cannot be placed in a different overlay from the segment to which it is allocated.

For examples of how to use the **FUNCTIONS** statement to create overlays, see your high-level language reference documentation.

The INCLUDE Statement

The **INCLUDE** statement inserts the contents of a specified text file where it is specified in the .DEF file. The inserted file must contain module statements as they would appear in the .DEF file in which they are being inserted.

Syntax

INCLUDE [['']]*filename*[['']]

Remarks

You can specify a path with the filename. Wildcards are not permitted. If *filename* contains a space (allowed under some installable systems), begins with a nonalphabetic character, or is a reserved word, enclose it in single or double quotation marks (' or "').

Multiple **INCLUDE** statements can appear in a .DEF file; each specifies a single insertion. **INCLUDE** statements can be nested up to 10 levels deep.

Reserved Words

The following words are reserved by the linker for use in module-definition files. These names can be used as arguments in module-definition statements only if the name is enclosed in double quotation marks ("').

ALIAS	EXPORTS	MOVEABLE
APPLOADER	FIXED	MULTIPLE
BASECLASS	FUNCTIONS	NAME
CODE	HEAPSIZE	NEWFILES
CONFORMING	HUGE	NODATA
CONSTANT	IMPORTS	NOEXPANDDOWN
CONTIGUOUS	IMPURE	NOIOPL
DATA	INCLUDE	NONAME
DESCRIPTION	INITGLOBAL	NONCONFORMING
DEV386	INITINSTANCE	NONDISCARDABLE
DEVICE	INVALID	NONE
DISCARDABLE	IOPL	NONPERMANENT
DOS	LIBRARY	NONSHARED
DYNAMIC	LOADONCALL	NOTWINDOWCOMPAT
EXECUTE-ONLY	LONGNAMES	NT
EXECUTEONLY	MACINTOSH	NULL
EXECUTEREAD	MAXVAL	OBJECTS
EXETYPE	MIXED1632	OLD
EXPANDDOWN	MOVABLE	OS2

OVERLAY
OVL
PERMANENT
PHYSICAL
POSIX
PRELOAD
PRIVATELIB
PROTMODE
PURE
READONLY
READWRITE
REALMODE
RESIDENT
RESIDENTNAME
SECTIONS
SEGMENTS
SHARED
SINGLE
STACKSIZE
STUB
SUBSYSTEM
SWAPPABLE
TERMINSTANCE
UNIX
UNKNOWN
VERSION
VIRTUAL
WINDOWAPI
WINDOWCOMPAT
WINDOWS
WINDOWSCHAR
WINDOWSNT

CHAPTER 15

Using EXEHDR

The Microsoft EXE File Header Utility (EXEHDR) version 3.00 displays the contents of an executable-file header and can be used to alter some fields in the header. You can display or alter headers of MS-DOS programs and segmented-executable files (applications or DLLs). Some header fields have a different meaning in a Windows-based application file; see your Windows operating system documentation for more information. Examples of EXEHDR usage include:

- Determining whether a file is an application or a dynamic-link library (DLL).
- Viewing the attributes set by the module-definition file.
- Viewing the number and size of code and data segments.
- Setting a new stack allocation.

Many of the header fields contain information that was set in the module-definition file or as input options when the LINK utility created the file. This chapter assumes you are familiar with LINK and module-definition files. For information about LINK, see Chapter 13. For information about module-definition (.DEF) files, see Chapter 14. Many of the terms and keywords used in this section are explained in these chapters.

Running EXEHDR

This section describes the EXEHDR command line and the options available for controlling EXEHDR.

The EXEHDR Command Line

To run EXEHDR, use the following command line:

EXEHDR [[*options*]] *filenames*

The *options* field specifies options used to modify EXEHDR output or change the file header. Options are described in the next section.

The *filenames* field specifies one or more applications or DLLs. If you do not provide an extension, EXEHDR assumes .EXE. You can specify a path with the filename.

EXEHDR Options

Option names are not case sensitive and can be abbreviated to the shortest unique name. This section uses meaningful yet legal forms of the option names. Specify number arguments to options in decimal format or C-language notation. EXEHDR has the following options:

/HEA[[P]]**:***number*
: Sets the heap allocation field to *number* bytes. This option is only for segmented-executable files.

/HEL[[P]]
: Calls the QuickHelp utility. If EXEHDR cannot locate the Help file or QuickHelp, it displays a brief summary of EXEHDR command-line syntax.

/MA[[X]]**:***number*
: Sets the maximum memory allocation to *number* 16-byte paragraphs. The memory allocation fields tell MS-DOS the maximum memory needed to load and run the program. The *number* must equal or exceed the minimum allocation. This option is equivalent to LINK's /CPARM option and applies only to MS-DOS programs.

/MI[[N]]**:***number*
: Sets the minimum memory allocation to *number* 16-byte paragraphs. See the /MAX option for more information.

/NE[[WFILES]]
: Sets a bit in the header to notify the loader that the program may be using an installable file system.

/NO[[LOGO]]
: Suppresses the EXEHDR copyright message.

/P[[MTYPE]]**:***type*
: Sets the type of application. The *type* is one of the keywords for either LINK's /PM option or the NAME statement in a .DEF file. The keywords are **PM** (or **WINDOWAPI**), **VIO** (or **WINDOWCOMPAT**), and **NOVIO** (or **NOTWINDOWCOMPAT**).

/R[[ESETERROR]]
Clears the error bit in the header of a segmented-executable file. LINK sets the error bit when it finds an unresolved reference or duplicate symbol definition. The operating system does not load a program if this bit is set. EXEHDR displays the message `Error in image` if it finds the error bit set. This option allows you to run a program that contains LINK errors and is useful during application development.

/S[[TACK]]**:***number*
Sets the stack allocation to *number* bytes. The /STACK option is equivalent to LINK's /STACK option.

/V[[ERBOSE]]
Provides more information about segmented-executable files. The additional information includes the default flags in the segment table, all run-time relocations, and additional fields from the header. For more information, see "EXEHDR Output: Verbose Output" on page 521.

/?
Displays a brief summary of EXEHDR command-line syntax.

Executable-File Format

MS-DOS applications have a simple format, which consists of a single header followed by a relocation table and the load module. Segmented-executable files have two headers. The first header, usually called the MS-DOS header, has a simple format. The second header, sometimes called the new .EXE header, has a more detailed format. Figure 15.1 shows the arrangement of the headers in a segmented-executable file. When the executable file runs with MS-DOS, the operating system uses the old header to load the file. Otherwise, the system ignores the MS-DOS (or "old") header and uses the new header.

The listing generated by EXEHDR shows the contents of the file header and information about each segment in the file. The type of listing generated reflects the structure of the header for the kind of file being checked. (For more information about the structure of MS-DOS applications and segmented-executable files, see the *MS-DOS Encyclopedia.*)

Figure 15.1 Format for a Segmented-Executable File

EXEHDR Output: MS-DOS Executable File

The EXEHDR output for an MS-DOS executable file appears as follows:

```
.EXE size (bytes)
Packed .EXE file
Magic number:
Bytes on last page:
Pages in file:
Relocations:
Paragraphs in header:
Extra paragraphs needed:
Extra paragraphs wanted:
Initial stack location:
Word checksum:
Entry point:
Relocation table address:
Memory needed:
```

The meaning of each field is described in the following list:

`.EXE size (bytes)`
Gives the size of the file on disk.

`Packed .EXE file`
Is displayed only if the file is packed.

`Magic number:`
Tells the operating-system loader the format of the header.

`Bytes on last page:`
Tells the loader how much data exists in the last page on disk.

`Pages in file:`
Gives the number of whole 512-byte pages in the file on disk. If the program contains overlays, this field shows the number of pages in the root.

`Relocations:`
Tells the loader the number of entries in the relocation table.

`Paragraphs in header:`
Gives the size of the header in 16-byte paragraphs. This represents the offset of the load image within the file.

`Extra paragraphs needed:`
Tells the loader the required minimum number of paragraphs of memory in addition to the image size. The image size is equal to `Bytes on last page + (Pages in file x 512)`.

`Extra paragraphs wanted:`
Tells the loader the number of paragraphs of memory above the size on disk requested for loading the program. This value is set by LINK's /CPARM option.

`Initial stack location:`
Gives the address (SS:SP) of the MS-DOS program's stack.

`Word checksum:`
Confirms for the loader that the file is a valid executable file.

`Entry point:`
Gives the start address.

`Relocation table address:`
Gives the location of the table of relocation pointers as an offset from the beginning of the file.

`Memory needed:`
Tells the loader the total memory needed to load the application. The value in this field is equal to `(Extra paragraphs needed x 16) + .EXE size (bytes)`.

EXEHDR Output: Segmented-Executable File

The first part of the EXEHDR output for a segmented-executable file appears as follows:

```
Module:
Description:
Data:
Initial CS:IP:
Initial SS:SP:
Extra stack allocation:
DGROUP:
```

The meaning of each field is described in the following list:

`Module:`
Gives the name of the application as specified in the **NAME** statement of the .DEF file used to create the file or the name assumed by default.

`Description:`
Gives the text of the **DESCRIPTION** statement of the .DEF file or the description assumed by default.

`Data:`
Indicates the program's default data segment (DGROUP) type: **SHARED**, **NONSHARED**, or **NONE**. This type can be specified in a .DEF file.

`Initial CS:IP:`
Gives the application's starting address.

`Initial SS:SP:`
Gives the value of the initial stack pointer, which gives the location of the initial stack.

`Extra stack allocation:`
Gives the size in bytes of the stack, specified in hexadecimal.

`DGROUP:`
Gives the segment number of DGROUP in the program's object files. Segment numbers start with the number 1.

At the end of the list of fields, EXEHDR displays any module flags that were set for every segment in the module. For example, `PROTMODE` may appear.

The message `Error in image` may appear at the end of the list of fields. If a LINK error (such as "unresolved external") occurs when the file is created, LINK sets the error bit in the header. This prevents the file from being loaded. You can clear the error bit with the /RESET option, described on page 515.

DLL Header Differences

For a DLL, the output differs slightly and appears as

```
Library:
Description:
Data:
Initialization:
Initial CS:IP:
Initial SS:SP:
DGROUP:
```

The meaning of each field is described in the following list:

`Library:`
Gives the name of the library as specified in the **LIBRARY** statement in the module-definition file (or the default name).

`Description:`
`Data:`
Same as for other segmented-executable files.

`Initialization:`
Gives the type of initalization as specified in the **LIBRARY** statement in the module-definition file (or the default initialization).

`Initial CS:IP:`
Gives the address of the initialization routine. If the DLL has no initialization routine, the start address is zero.

`Initial SS:SP:`
May be zero for a DLL.

`DGROUP:`
May not appear for a DLL.

Segment Table

After the header fields for a segmented-executable file, EXEHDR displays the segment table. All values appear in hexadecimal except for the segment index number. An example of this table is:

```
no. type address  file  mem   flags
  1 CODE 00000400 00efb 00efb
  2 DATA 00001400 00031 0007d
  3 DATA 00001600 0003c 00040 SHARED
```

The following list describes each heading in the segment table:

`no.`
Segment index number (in decimal), starting with 1.

`type`
Identification of the segment as a code or data segment.

`address`
A seek offset for the segment within the file.

`file`
Size in bytes of the segment in the file on disk.

`mem`
Size in bytes of the segment in memory. If `mem` is greater than `file`, the operating system pads the extra space with zero values at load time.

`flags`
Segment attributes. If the /V option is not used, only nondefault attributes are listed. Attributes that are meaningful only to the Windows operating system are displayed in lowercase and in parentheses.

Exports Table

Following the segment table, EXEHDR displays a table of exports if they exist. An example of this table is:

```
Exports:
ord seg offset name
  1   3  0000  HELPWNDPROC exported
 19   3  032e  ICONWNDPROC exported
 21  35  0000  PATHWNDPROC exported
  5  30  0264  ANNOUPDATEDLG exported
  8  33  0000  BOOKMARKDLG exported
```

The following list describes each heading in the Exports table:

`ord`
The ordinal number as specified in the **@***ord* field in an **EXPORTS** statement in a module-definition file. If *ord* was not specified, this column entry is blank.

`seg`
The index of the segment where the exported name is defined.

`offset`
The offset in the segment where the exported name is defined.

`name`
The exported name of the routine plus all flags applied to the exported routine, as specified in the **EXPORTS** statement in the module-definition file.

EXEHDR Output: Verbose Output

The /V option provides more extensive information about a segmented-executable file. The verbose output more closely reflects the file's header structure. (For an illustration of this structure, see Figure 15.1 on page 516.)

MS-DOS Header Information

EXEHDR begins by displaying the MS-DOS fields described on page 517, with the addition of two fields:

`Reserved words:`
Displays the contents of space in an MS-DOS header that is normally unused.

`New .EXE header address:`
Holds the starting location of the part of the header describing the segmented-executable file.

New .EXE Header Information

EXEHDR then displays the header fields for the segmented-executable file. In addition to the default fields described on page 518, the verbose output includes many other fields.

A field called `Operating system:` follows the `Description:` field. This field tells the system under which the program is to run.

The following fields are then displayed:

```
Linker version:
32-bit Checksum:
Segment Table:
Resource Table:
Resident Names Table:
Module Reference Table:
Imported Names Table:
Entry Table:
Non-resident Names Table:
Movable entry points:
Segment sector size:
Heap allocation:
Application type:
Other module flags:
```

The meaning of each field is described in the following list:

`Linker version:`
Tells which version of LINK was used to create the segmented-executable file.

`32-bit Checksum:`
Confirms for the loader that the file is a valid executable file. (See the `Word checksum:` field for MS-DOS executable files.)

`Segment Table:`
`Resource Table:`
`Resident Names Table:`
`Module Reference Table:`
`Imported Names Table:`
`Entry Table:`
`Non-resident Names Table:`
Describe various tables in the segmented-executable file. Each description gives the table name, its address within the file, and its length in hexadecimal and in decimal.

`Movable entry points:`
Gives the number of entries to segments that have the **MOVABLE** attribute. This field is used only by the Windows operating system.

`Segment sector size:`
Gives the alignment set by the /ALIGN option or the default of 512. This field equals the sector size on disk.

`Heap allocation:`
Gives the size of the heap. This field is displayed only if a **HEAPSIZE** statement appeared in the module-definition file.

`Application type:`
Gives the type as specified in the **NAME** statement of the module-definition file used to create the file being examined, or as specified with LINK's /PM option, or assumed by default. For a DLL, a 0 is always displayed.

`Other module flags:`
Gives other attributes of the file; if none, this field is not displayed.

At the end of the list of fields, EXEHDR displays any module flags that were set for every segment in the module. For example, `PROTMODE` may appear.

Tables

Following the header fields, EXEHDR displays the segment table with complete attributes, not just the nondefault attributes. Attributes that are meaningful only to the Windows operating system are displayed in lowercase and in parentheses. In addition to the attributes specified in the module-definition file (described in "CODE, DATA, and SEGMENTS Attributes" on page 503) or assumed by default, the verbose output includes the following two attributes:

- The `relocs` attribute is displayed for each segment that has address relocations. Relocations occur in each segment that references objects in other segments or makes dynamic-link references.
- The `iterated` attribute is displayed for each segment that has iterated data. Iterated data consist of a special code that packs repeated bytes.

EXEHDR then displays the Exports table if exports exist.

Relocations

Following the tables, EXEHDR displays descriptions of relocations. Each has a heading in the following form:

```
1 type   offset target
  BASE    eff4  seg   1 offset 0000
  BASE    f204  seg   2 offset 0000
  OFFSET  eff1  seg   1 offset e968
  OFFSET  314e  seg   1 offset 32ea
  BASE    c0f1  seg   3 offset 0000
  OFFSET  d397  seg   1 offset cf70
  PTR     cd3e  imp DOSCALLS.137
  OFFSET  b1a8  seg   1 offset ae7c
  PTR     f57c  imp KBDCALLS.13
```

The following list describes each heading:

number
The segment number, as given earlier in the segments table.

`type`
Relocation type, which gives the kind of address information requested.

`offset`
The location of the requested address change in the source segment.

`target`
The requested relocation address.

Each relocation table ends by stating the total number of relocations.

P A R T 4

Utilities

Chapter 16 Managing Projects with NMAKE. 527
Chapter 17 Managing Libraries with LIB. 581
Chapter 18 Creating Help Files with HELPMAKE 593
Chapter 19 Browser Utilities. 615
Chapter 20 Using Other Utilities 631

CHAPTER 16

Managing Projects with NMAKE

This chapter describes the Microsoft Program Maintenance Utility (NMAKE) version 1.20. NMAKE is a sophisticated command processor that saves time and simplifies project management. Once you specify which project files depend on others, NMAKE automatically builds your project without recompiling files that haven't changed since the last build.

If you are using the Programmer's Workbench (PWB) to build your project, PWB automatically creates a makefile and calls NMAKE to run the file. You may want to read this chapter if you intend to build your program outside of PWB, if you want to understand or modify a makefile created by PWB, or if you want to use a foreign makefile in PWB.

NMAKE can swap itself to expanded memory, extended memory, or disk to leave room for the commands it spawns. (For more information, see the description of the /M option on page 531.)

New Features

NMAKE version 1.20 offers the following new features. For details of each feature, see the reference part of this chapter.

- New options: /B, /K, /M, /V
- The **!MESSAGE** directive
- Two preprocessing operators: **DEFINED**, **EXIST**
- Three keywords for use with the **!ELSE** directive: **IF**, **IFDEF**, **IFNDEF**
- New directives: **!ELSEIF**, **!ELSEIFDEF**, **!ELSEIFNDEF**
- Addition of .CPP and .CXX to the **.SUFFIXES** list
- Predefined macros for C++ programs: **CPP**, **CXX**, **CPPFLAGS**, **CXXFLAGS**
- Predefined inference rules for C++ programs

Overview

NMAKE works by looking at the “time stamp” of a file. A time stamp is the time and date the file was last modified. Time stamps are assigned by most operating systems in 2-second intervals. NMAKE compares the time stamps of a “target” file and its “dependent” files. A target is usually a file you want to create, such as an executable file, though it could be a label for a set of commands you wish to execute. 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 a later time stamp than the target or if the target does not exist. (For information on how the 2-second interval affects your build, see the description of the /B option on page 530.)

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 “makefile” that you supply. A makefile (sometimes called a description file) is a text file containing a set of instructions that NMAKE uses to build your project. The instructions consist of description blocks, macros, directives, and inference rules. Each description block typically lists a target (or targets), the target’s dependents, and the commands that build the target. NMAKE compares the time stamp on the target file with the time stamp on the dependent files. If the time stamp of any dependent is the same as or later than the time stamp of the target, NMAKE updates the target by executing the commands listed in the description block.

It is possible to run NMAKE without a makefile. In this case, NMAKE uses predefined macros and inference rules along with instructions given on the command line or in TOOLS.INI. (For information on the TOOLS.INI file, see page 534.)

NMAKE’s main purpose is to help you build programs quickly and easily. However, it is not limited to compiling and linking; NMAKE can run other types of programs and can execute operating system commands. You can use NMAKE to prepare backups, move files, and perform other project-management tasks that you ordinarily do at the operating-system prompt.

This chapter uses the term “build,” as in building a target, to mean evaluating the time stamps of a target and its dependent and, if the target is out of date, executing the commands associated with the target. The term “build” has this meaning whether or not the commands actually create or change the target file.

Running NMAKE

You invoke NMAKE with the following syntax:

NMAKE [[*options*]] [[*macros*]] [[*targets*]]

The *options* field lists NMAKE options, which are described in the following section, “Command-Line Options.”

The *macros* field lists macro definitions, which allow you to change text in the makefile. The syntax for macros is described in “User-Defined Macros” on page 551.

The *targets* field lists targets to build. NMAKE builds only the targets listed on the command line. If you don’t specify a target, NMAKE builds only the first target in the first dependency in the makefile. (You can use a pseudotarget to tell NMAKE to build more than one target. See “Pseudotargets” on page 540.)

NMAKE uses the following priorities to determine how to conduct the build:

1. If the /F option is used, NMAKE searches the current or specified directory for the specified makefile. NMAKE halts and displays an error message if the file does not exist.
2. If you do not use the /F option, NMAKE searches the current directory for a file named MAKEFILE.
3. If MAKEFILE does not exist, NMAKE checks the command line for target files and tries to build them using inference rules (either defined in TOOLS.INI or predefined). This feature lets you use NMAKE without a makefile as long as NMAKE has an inference rule for the target.
4. If a makefile is not used and the command line does not specify a target, NMAKE halts and displays an error message.

Example

The following command specifies an option (/S) and a macro definition (`"program=sample"`) and tells NMAKE to build two targets (`sort.exe` and `search.exe`). The command does not specify a makefile, so NMAKE looks for MAKEFILE or uses inference rules.

```
NMAKE /S "program=sample" sort.exe search.exe
```

For information on NMAKE macros, see page 550.

Command-Line Options

NMAKE accepts options for controlling the NMAKE session. Options are not case sensitive and can be preceded by either a slash (/) or a dash (–).

You can specify some options in the makefile or in TOOLS.INI.

/A
Forces NMAKE to build all evaluated targets, even if the targets are not out-of-date with respect to their dependents. This option does not force NMAKE to build unrelated targets.

/B
Tells NMAKE to execute a dependency even if time stamps are equal. Most operating systems assign time stamps with a resolution of 2 seconds. If your commands execute quickly, NMAKE may conclude that a file is up to date when in fact it is not. This option may result in some unnecessary build steps but is recommended when running NMAKE on very fast systems.

/C
Suppresses default NMAKE output, including nonfatal NMAKE error or warning messages, time stamps, and the NMAKE copyright message. If both /C and /K are specified, /C suppresses the warnings issued by /K.

/D
Displays information during the NMAKE session. The information is interspersed in NMAKE's default output to the screen. NMAKE displays the time stamp of each target and dependent evaluated in the build and issues a message when a target does not exist. Dependents for a target precede the target and are indented. The /D and /P options are useful for debugging a makefile.

To set or clear /D for part of a makefile, use the **!CMDSWITCHES** directive; see "Preprocessing Directives" on page 572.

/E
Causes environment variables to override macro definitions in the makefile. See "Macros" on page 550.

/F *filename*
Specifies *filename* as the name of the makefile. Zero or more spaces or tabs precede *filename*. If you supply a dash (–) instead of a filename, NMAKE gets makefile input from the standard input device. (End keyboard input with either F6 or CTRL+Z.) NMAKE accepts more than one makefile; use a separate /F specification for each makefile input.

If you omit /F, NMAKE searches the current directory for a file called MAKEFILE (with no extension) and uses it as the makefile. 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.

/I

Ignores exit codes from all commands listed in the makefile. NMAKE processes the whole makefile even if errors occur. To ignore exit codes for part of a makefile, use the dash (–) command modifier or the **.IGNORE** directive; see "Command Modifiers" on page 544 and "Dot Directives" on page 570. To set or clear /I for part of a makefile, use the **!CMDSWITCHES** directive; see "Preprocessing Directives" on page 572. To ignore errors for unrelated parts of the build, use the /K option; /I overrides /K if both are specified.

/K

Continues the build for unrelated parts of the dependency tree if a command terminates with an error. By default, NMAKE halts if any command returns a nonzero exit code. If this option is specified and a command returns a nonzero exit code, NMAKE ceases to execute the block containing the command. It does not attempt to build the targets that depend on the results of that command; instead, it issues a warning and builds other targets. When /K is specified and the build is not complete, NMAKE returns an exit code of 1. This differs from the /I option, which ignores exit codes entirely; /I overrides /K if both are specified. The /C option suppresses the warnings issued by /K.

/M

Swaps NMAKE to disk to conserve extended or expanded memory under MS-DOS. By default, NMAKE leaves room for commands to be executed in low memory by swapping itself to extended memory (if enough space exists there) or to expanded memory (if there is not sufficient extended memory but there is enough expanded memory) or to disk. The /M option tells NMAKE to ignore any extended or expanded memory and to swap only to disk.

/N

Displays but does not execute the commands that would be executed by the build. Preprocessing commands are executed. This option is useful for debugging makefiles and checking which targets are out-of-date. To set or clear /N for part of a makefile, use the **!CMDSWITCHES** directive; see "Preprocessing Directives" on page 572.

/NOLOGO

Suppresses the NMAKE copyright message.

/P

Displays NMAKE information to the standard output device, including all macro definitions, inference rules, target descriptions, and the **.SUFFIXES** list, before running the NMAKE session. If /P is specified without a makefile or command-line target, NMAKE displays the information and does not issue an error. The /P and /D options are useful for debugging a makefile.

/Q
Checks time stamps of targets that would be updated by the build but does not run the build. NMAKE returns a zero exit code if the targets are up-to-date and a nonzero exit code if any target is out-of-date. Only preprocessing commands in the makefile are executed. This option is useful when running NMAKE from a batch file.

/R
Clears the **.SUFFIXES** list and ignores inference rules and macros that are defined in the TOOLS.INI file or that are predefined.

/S
Suppresses the display of all executed commands. To suppress the display of commands in part of a makefile, use the **@** command modifier or the **.SILENT** directive; see "Command Modifiers" on page 544 and "Dot Directives" on page 570. To set or clear /S for part of a makefile, use the **!CMDSWITCHES** directive; see "Preprocessing Directives" on page 572.

/T
Changes time stamps of command-line targets (or first target in the makefile if no command-line targets are specified) to the current time and executes preprocessing commands but does not run the build. Contents of target files are not modified.

/V
Causes all macros to be inherited when recursing. By default, only macros defined on the command line and environment-variable macros are inherited when NMAKE is called recursively. This option makes all macros available to the recursively called NMAKE session. See "Inherited Macros" on page 563.

/X *filename*
Sends all NMAKE error output to *filename*, which can be a file or a device. Zero or more spaces or tabs can precede *filename*. If you supply a dash (–) instead of a filename, NMAKE sends its error output to standard output. By default, NMAKE sends errors to standard error. This option does not affect output that is sent to standard error by commands in the makefile.

/?
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 makefile 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.

NMAKE Command File

You can place a sequence of command-line arguments 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. You can use a command file to overcome the limit on the length of a command line in the operating system (128 characters in MS-DOS).

To provide input to NMAKE with a command file, type

NMAKE **@***commandfile*

The *commandfile* is the name of a text file containing the information NMAKE expects on the command line. Precede the name of the command file with an at sign (**@**). You can specify a path with the filename.

NMAKE treats the file as if it were a single set of arguments. It replaces each line break with a space. Macro definitions that contain spaces must be enclosed in quotation marks; see "Where to Define Macros" on page 552.

You can split input between the command line and a command file. Specify **@***commandfile* on the command line at the place where the file's information is expected. Command-line input can precede and/or follow the command file. You can specify more than one command file.

Example 1

If a file named UPDATE contains the line

```
/S "program = sample" sort.exe search.exe
```

you can start NMAKE with the command

```
NMAKE @update
```

The effect is the same as if you entered the following command line:

```
NMAKE /S "program = sample" sort.exe search.exe
```

Example 2

The following is another version of the UPDATE file:

```
/S "program \
= sample" sort.exe search.exe
```

The backslash (\) allows the macro definition to span two lines.

Example 3

If the command file called UPDATE contains the line

```
/S "program = sample" sort.exe
```

you can start NMAKE with the command

```
NMAKE @update search.exe
```

The TOOLS.INI File

You can customize NMAKE by placing commonly used information in the TOOLS.INI initialization file. Settings for NMAKE must follow a line that begins with the tag `[NMAKE]`. The tag is not case sensitive. This section of the initialization file can contain any makefile information. NMAKE uses this information in every session, unless you run NMAKE with the /R option. NMAKE looks for TOOLS.INI first in the current directory and then in the directory specified by the INIT environment variable.

You can use the **!CMDSWITCHES** directive to specify command-line options in TOOLS.INI. An option specified this way is in effect for every NMAKE session. This serves the same purpose as does an environment variable, which is a feature available in other utilities. For more information on **!CMDSWITCHES**, see page 572.

Macros and inference rules appearing in TOOLS.INI can be overridden. See "Precedence Among Macro Definitions" on page 563 and "Precedence Among Inference Rules" on page 570.

NMAKE reads information in TOOLS.INI before it reads makefile information. Thus, for example, a description block appearing in TOOLS.INI acts as the first description block in the makefile; this is true for every NMAKE session, unless /R is specified.

To place a comment in TOOLS.INI, specify the comment on a separate line beginning with a semicolon (;). You can also specify comments with a number sign (#) as you can in a makefile; for more information, see "Comments" on page 536.

Example

The following is an example of text in a TOOLS.INI file:

```
[NMAKE]
; macros
AS     = masm
AFLAGS = /FR /LA /ML /MX /W2
; inference rule
.asm.obj:
    $(AS) /ZD ZI $(AFLAGS) $.asm
```

NMAKE reads and applies the lines following `[NMAKE]`. The example redefines the macro **AS** to invoke the Microsoft Macro Assembler MASM.EXE utility., redefines the macro **AFLAGS**, and redefines the inference rule for making .OBJ files from .ASM sources. These NMAKE features are explained throughout this chapter.

Contents of a Makefile

An NMAKE makefile contains description blocks, macros, inference rules, and directives. This section presents general information about writing makefiles. The rest of the chapter describes each of the elements of makefiles in detail.

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 without its special meaning, place a caret (^) in front of it. NMAKE ignores carets that precede characters other than the special characters listed previously. A caret within a quoted string is treated as a literal caret character.

You can also use a caret at the end of a line to insert a literal newline character in a string or macro. 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 in a macro definition. A newline character that follows a backslash is replaced with a space. For more information, see “User-Defined Macros” on page 551.

In a command, a percent symbol (**%**) can represent the beginning of a file specifier. (See "Filename-Parts Syntax" on page 546.) NMAKE interprets %s as a filename, and it interprets the character sequence of %| followed by d, e, f, p, or F as part or all of a filename or path. If you need to represent these characters literally in a command, specify a double percent sign (**%%**) in place of a single one. In all other situations, NMAKE interprets a single % literally. However, NMAKE always interprets a double %% as a single %. Therefore, to represent a literal %%, you can specify either three percent signs, %%%, or four percent signs, %%%%.

To use the dollar sign (**$**) as a literal character in a command, you must specify two dollar signs (**$$**); this method can also be used in other situations where **^$** also works.

For information on literal characters in macro definitions, see "Special Characters in Macros" on page 551.

Wildcards

You can use MS-DOS wildcards (***** and **?**) to specify target and dependent names. NMAKE expands wildcards that appear on dependency lines. A wildcard specified in a command is passed to the command; NMAKE does not expand it.

Example

In the following description block, the wildcard `*` is used twice:

```
project.exe : *.asm
    ml *.asm /Feproject.exe
```

NMAKE expands the `*.asm` in the dependency line and looks at all files having the .ASM extension in the current directory. If any .ASM file is out-of-date, the ML command expands the `*.c` and compiles and links all files.

Comments

To place a comment in a makefile, precede it with a number sign (**#**). If the entire line is a comment, the **#** must appear at the beginning of the line. Otherwise, the **#** follows the item being commented. NMAKE ignores all text from the number sign to the next newline character.

Command lines cannot contain comments; this is true even for a command that is specified on the same line as a dependency line or inference rule. This is because NMAKE does not parse a command; instead, it passes the entire command to the operating system. However, a comment can appear between lines in a commands block. To change a command to a comment, insert a **#** at the beginning of the command line.

You can use comments in the following situations:

```
# Comment on line by itself

OPTIONS = /MAP  # Comment on macro definition line

all.exe : one.obj two.obj  # Comment on dependency line
    link one.obj two.obj;
# Comment in commands block
    copy one.exe \release
# Command turned into comment:
#   copy *.obj \objects

.obj.exe:  # Comment in inference rule
```

To specify a literal #, precede it with a caret (^), as in the following:

```
DEF = ^#define  #Macro representing a C preprocessing directive
```

Comments can also appear in a TOOLS.INI file. TOOLS.INI allows an additional form of comment using a semicolon (;). See “The TOOLS.INI File” on page 534.

Long Filenames

You can use long filenames if they are supported by your file system. However, you must enclose the name in double quotation marks, as in the following dependency line:

```
all : "VeryLongFileName.exe"
```

Description Blocks

Description blocks form the heart of the makefile. The following is a typical NMAKE description block:

Figure 16.1 NMAKE Description Block

The first line in a description block is the “dependency line.” In this example, the the dependency contains one “target” and three “dependents.” The dependency is followed by a commands block that lists one or more commands.

The following sections discuss dependencies, targets, and dependents. The contents of a commands block are described in “Commands” on page 543.

Dependency Line

A description block begins with a “dependency line.” A dependency line specifies one or more “target” files and then lists zero or more “dependent” files. If a target does not exist, or if its time stamp is earlier than that of any dependent, NMAKE executes the commands block for that target. The following is an example of a dependency line:

```
myapp.exe : myapp.obj another.obj myapp.def
```

This dependency line tells NMAKE to rebuild the MYAPP.EXE target whenever MYAPP.OBJ, ANOTHER.OBJ, or MYAPP.DEF has changed more recently than MYAPP.EXE.

The dependency line must not be indented (it cannot start with a space or tab). The first target must be specified at the beginning of the line. Targets are separated from dependents by a single colon, except as described in “Using Targets in Multiple Description Blocks” on page 539. The colon can be preceded or followed by zero or more spaces or tabs. The entire dependency must appear on one line; however, you can extend the line by placing a backslash (\) after a target or dependent and continuing the dependency on the next line.

Before executing any commands, NMAKE moves through all dependencies and applicable inference rules to build a “dependency tree” that specifies all the steps required to fully update the target. NMAKE checks to see if dependents themselves are targets in other dependency lists, if any dependents in those lists are targets elsewhere, and so on. After it builds the dependency tree, NMAKE checks time stamps. If it finds any dependents in the tree that are newer than the target, NMAKE builds the target.

Targets

The targets section of the dependency line lists one or more target names. At least one target must be specified. Separate multiple target names with one or more spaces or tabs. You can specify a path with the filename. Targets are not case sensitive. A target (including path) cannot exceed 256 characters.

If the name of the last target before the colon (**:**) is a single character, you must put a space between the name and the colon; otherwise, NMAKE interprets the letter-colon combination as a drive specifier.

Usually a target is the name of a file to be built using the commands in the description block. However, a target can be any valid filename, or it can be a pseudotarget. (For more information, see “Pseudotargets” on page 540.)

NMAKE builds targets specified on the NMAKE command line. If a command-line target is not specified, NMAKE builds the first target in the first dependency in the makefile.

The example in the previous section tells NMAKE how to build a single target file called MYAPP.EXE if it is missing or out-of-date.

Using Targets in Multiple Description Blocks

A target can appear in only one description block when specified using the single-colon (**:**) syntax to separate the target from the dependent. To update a target using more than one description block, specify two consecutive colons (**::**) between targets and dependents. One use for this feature is for building a complex target that contains components created with different commands.

Example

The following makefile updates a library:

```
target.lib :: one.asm two.asm three.asm
    ML one.asm two.asm three.asm
    LIB target -+one.obj -+two.obj -+three.obj;
target.lib :: four.c five.c
    CL /c four.c five.c
    LIB target -+four.obj -+five.obj;
```

If any of the assembly-language files have changed more recently than the library, NMAKE assembles the source files and updates the library. Similarly, if any of the C-language files have changed, NMAKE compiles the C files and updates the library.

Accumulating Targets in Dependencies

Dependency lines are cumulative when the same target appears more than once in a single description block. For example,

```
bounce.exe : jump.obj
bounce.exe : up.obj
    echo Building bounce.exe...
```

is evaluated by NMAKE as

```
bounce.exe : jump.obj up.obj
    echo Building bounce.exe...
```

This evaluation has several effects. Since NMAKE builds the dependency tree based on one target at a time, the lines can contain other targets, as in:

```
bounce.exe leap.exe : jump.obj
bounce.exe climb.exe : up.obj
    echo Building bounce.exe...
```

The preceding example is evaluated by NMAKE as

```
bounce.exe : jump.obj
leap.exe : jump.obj
bounce.exe : up.obj
climb.exe : up.obj...
    echo Building bounce.exe...
```

NMAKE evaluates a dependency for each of the three targets as if each were specified in a separate description block. If `bounce.exe` or `climb.exe` is out-of-date, NMAKE runs the given command. If `leap.exe` is out-of-date, the given command does not apply, and NMAKE tries to use an inference rule.

If the same target is specified in two single-colon dependency lines in different locations in the makefile, and if commands appear after only one of the lines, NMAKE interprets the dependency lines as if they were adjacent or combined. This can cause an unwanted side effect: NMAKE does not invoke an inference rule for the dependency that has no commands (see "Inference Rules" on page 563). Rather, it assumes that the dependencies belong to one description block and executes the commands specified with the other dependency.

The following makefile is interpreted in the same way as the preceding examples:

```
bounce.exe : jump.obj
    echo Building bounce.exe...
.
.
.
bounce.exe : up.obj
```

This effect does not occur if the colons are doubled (**::**) after the duplicate targets. A double-colon dependency with no commands block invokes an inference rule, even if another double-colon dependency containing the same target is followed by a commands block.

Pseudotargets

A "pseudotarget" is a target that doesn't specify a file but instead names a label for use in executing a group of commands. NMAKE interprets the pseudotarget as a file that does not exist and thus is always out-of-date. When NMAKE evaluates a pseudotarget, it always executes its commands block. Be sure that the current directory does not contain a file with a name that matches the pseudotarget.

A pseudotarget name must follow the syntax rules for filenames. Like a filename target, a pseudotarget name is not case sensitive. However, if the name does not have an extension (that is, it does not contain a period), it can exceed the 8-character limit for filenames and can be up to 256 characters long.

A pseudotarget can be listed as a dependent. A pseudotarget used this way must appear as a target in another dependency; however, that dependency does not need to have a commands block.

A pseudotarget used as a target has an assumed time stamp that is the most recent time stamp of all its dependents. If a pseudotarget has no dependents, the assumed time stamp is the current time. NMAKE uses the assumed time stamp if the pseudotarget appears as a dependent elsewhere in the makefile.

Pseudotargets are useful when you want NMAKE to build more than one target automatically. NMAKE builds only those targets specified on the NMAKE command line, or, when no command-line target is specified, it builds only the first target in the first dependency in the makefile. To tell NMAKE to build multiple targets without having to list them on the command line, write a description block with a dependency containing a pseudotarget and list as its dependents the targets you want to build. Either place this description block first in the makefile or specify the pseudotarget on the NMAKE command line.

Example 1

In the following example, `UPDATE` is a pseudotarget.

```
UPDATE : *.*
    !COPY $** a:\product
```

If `UPDATE` is evaluated, NMAKE copies all files in the current directory to the specified drive and directory.

Example 2

In the following makefile, the pseudotarget `all` builds both PROJECT1.EXE and PROJECT2.EXE if either `all` or no target is specified on the command line. The pseudotarget `setenv` changes the LIB environment variable before the .EXE files are updated:

```
all : setenv project1.exe project2.exe

project1.exe : project1.obj
    LINK project1;

project2.exe : project2.obj
    LINK project2;

setenv :
    set LIB=\project\lib
```

Dependents

The dependents section of the dependency line lists zero or more dependent names. Usually a dependent is a file used to build the target. However, a dependent can be any valid filename, or it can be a pseudotarget. You can specify a path with the filename. Dependents are not case sensitive. Separate each dependent name with one or more spaces or tabs. A single or double colon (**:** or **::**) separates it from the targets section.

Along with dependents you explicitly list in the dependency line, NMAKE can assume an "inferred dependent." An inferred dependent is derived from an inference rule. (For more information, see "Inference Rules" on page 563.) NMAKE considers an inferred dependent to appear earlier in a dependents list than explicit dependents. It builds inferred dependents into the dependency tree. It is important to note that when an inferred dependent in a dependency is out-of-date with respect to a target, NMAKE invokes the commands block associated with the dependency, just as it does with an explicit dependent.

NMAKE uses the dependency tree to make sure that dependents themselves are updated before it updates their targets. If a dependent file doesn't exist, NMAKE looks for a way to build it; if it already exists, NMAKE looks for a way to make sure it is up-to-date. If the dependent is listed as a target in another dependency, or if it is implied as a target in an inference rule, NMAKE checks that the dependent is up-to-date with respect to its own dependents; if the dependent file is out-of-date or doesn't exist, NMAKE executes the commands block for that dependency.

The following example lists three dependents after MYAPP.EXE:

```
myapp.exe : myapp.obj another.obj myapp.def
```

Specifying Search Paths for Dependents

You can specify the directories in which NMAKE should search for a dependent. The syntax for a directory specification is:

{*directory*[[**;***directory*...]]**}***dependent*

Enclose one or more directory names in braces (**{ }**). Separate multiple directories with a semicolon (**;**). No spaces are allowed. You can use a macro to specify part or all of a search path. NMAKE searches the current directory first, then the directories in the order specified. A search path applies only to a single dependent.

Example

The following dependency line contains a directory specification:

```
forward.exe : {\src\alpha;d:\proj}pass.obj
```

The target FORWARD.EXE has one dependent, PASS.OBJ. The directory list specifies two directories. 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.

Commands

The commands section of a description block or inference rule lists the commands that NMAKE must run if the dependency is out-of-date. You can specify any command or program that can be executed from an MS-DOS command line (with a few exceptions, such as PATH). Multiple commands can appear in a commands block. Each appears on its own line (except as noted in the next section). If a description block doesn't contain any commands, NMAKE looks for an inference rule that matches the dependency. (See "Inference Rules" on page 563.) The following example shows two commands following a dependency line:

```
myapp.exe : myapp.obj another.obj myapp.def
    link myapp another, , NUL, mylib, myapp
    copy myapp.exe c:\project
```

NMAKE displays each command line before it executes it, unless you specify the /S option, the **.SILENT** directive, the **!CMDSWITCHES** directive, or the **@** modifier.

Command Syntax

A command line must begin with one or more spaces or tabs. NMAKE uses this indentation to distinguish between a dependency line and a command line.

Blank lines cannot appear between the dependency line and the commands block. However, a line containing only spaces or tabs can appear; this line is interpreted as a null command, and no error occurs. Blank lines can appear between command lines.

A long command can span several lines if each line ends with a backslash (\ ). A backslash at the end of a line is interpreted as a space on the command line. For example, the LINK command shown in previous examples in this chapter can be expressed as:

```
    link myapp\
another, , NUL, mylib, myapp
```

NMAKE passes the continued lines to the operating system as one long command. A command continued with a backslash must still be within the

operating system's limit on the length of a command line. If any other character, such as a space or tab, follows the backslash, NMAKE interprets the backslash and the trailing characters literally.

You can also place a single command at the end of a dependency line, whether or not other commands follow in the indented commands block. Use a semicolon (**;**) to separate the command from the rightmost dependent, as in:

```
project.obj : project.c project.h ; cl /c project.c
```

Command Modifiers

Command modifiers provide extra control over the commands in a description block. You can use more than one modifier for a single command. Specify a command modifier preceding the command being modified, optionally separated by spaces or tabs. Like a command, a modifier cannot appear at the beginning of a line. It must be preceded by one or more spaces or tabs.

The following describes the three NMAKE command modifiers.

@*command*
: Prevents NMAKE from displaying the command. Any results displayed by commands are not suppressed. Spaces and tabs can appear before the command. By default, NMAKE echoes all makefile commands that it executes. The /S option suppresses display for the entire makefile; the **.SILENT** directive suppresses display for part of the makefile.

–[*number*]*command*
: Turns off error checking for the command. Spaces and tabs can appear before the command. By default, NMAKE halts when any command returns an error in the form of a nonzero exit code. This modifier tells NMAKE to ignore errors from the specified command. If the dash is followed by a number, NMAKE stops if the exit code returned by the command is greater than that number. No spaces or tabs can appear between the dash and the number; they must appear between the number and the command. (For more information on using this number, see "Exit Codes from Commands" on page 545.) The /I option turns off error checking for the entire makefile; the **.IGNORE** directive turns off error checking for part of the makefile.

!*command*
: Executes the command for each dependent file if the command preceded by the exclamation point uses the predefined macros **$**** or **$?**. (See "Filename Macros" on page 555.) Spaces and tabs can appear before the command. The **$**** macro represents all dependent files in the dependency line. The **$?** macro refers to all dependent files in the dependency line that have a later time stamp than the target.

Example 1

In the following example, 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.

Example 2

In the following description block, if the program `sample` returns a nonzero exit code, NMAKE does not halt; if `sort` returns an exit code that is greater than 5, NMAKE stops:

```
light.lst : light.txt
    -sample light.txt
    -5 sort light.txt
```

Example 3

The description block

```
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:
```

Exit Codes from Commands

NMAKE stops execution if a command or program executed in the makefile encounters an error and returns a nonzero exit code. The exit code is displayed in an NMAKE error message.

You can control how NMAKE behaves when a nonzero exit code occurs by using the /I or /K option, the **.IGNORE** directive, the **!CMDSWITCHES** directive, or the dash (**–**) command modifier.

Another way to use exit codes is during preprocessing. You can run a command or program and test its exit code using the **!IF** preprocessing directive. For more information, see “Executing a Program in Preprocessing” on page 575.

Filename-Parts Syntax

NMAKE provides a syntax that you can use in commands to represent components of the name of the first dependent file. This file is generally the first file listed to the right of the colon in a dependency line. However, if a dependent is implied from an inference rule, NMAKE considers the inferred dependent to be the first dependent file, ahead of any explicit dependents. If more than one inference rule applies, the **.SUFFIXES** list determines which dependent is first. The filename components are the file's drive, path, base name, and extension as you have specified it, not as it exists on disk.

You can represent the complete filename with the following syntax:

%s

For example, if a description block contains

```
sample.exe : c:\project\sample.obj
    LINK %s;
```

NMAKE interprets the command as

```
LINK c:\project\sample.obj;
```

You can represent parts of the complete filename with the following syntax:

%|[[*parts*]]**F**

where *parts* can be zero or more of the following letters, in any order:

Letter	Description
No letter	Complete name
d	Drive
p	Path
f	File base name
e	File extension

Using this syntax, you can represent the full filename specification by `%|F` or by `%|dpfeF`, as well as by `%s`.

Example

The following description block uses filename-parts syntax:

```
sample.exe : c:\project\sample.obj
    LINK %s, a:%|pfF.exe;
```

NMAKE interprets the first representation as the complete filename of the depen-dent. It interprets the second representation as a filename with the same path and base name as the dependent but on the specified drive and with the specified extension. It executes the following command:

```
LINK c:\project\sample.obj, a:\project\sample.exe;
```

Note For another way to represent components of a filename, see "Modifying Filename Macros" on page 556.

Inline Files

NMAKE can create "inline files" in the commands section of a description block or inference rule. An inline file is created on disk by NMAKE and contains text you specify in the makefile. The name of the inline file can be used in commands in the same way as any filename. NMAKE creates the inline file only when it executes the command in which the file is created.

One way to use an inline file is as a response file for another utility such as LINK or LIB. Response files avoid the operating system limit on the maximum length of a command line and automate the specification of input to a utility. Inline files eliminate the need to maintain a separate response file. They can also be used to pass a list of commands to the operating system.

Specifying an Inline File

The syntax for specifying an inline file in a command is:

<<[[*filename*]]

Specify the double angle brackets (<<) on the command line at the location where you want a filename to appear. Because command lines must be indented, the angle brackets cannot appear at the beginning of a line. The angle bracket syntax must be specified literally; it cannot be represented by a macro expansion.

When NMAKE executes the description block, it replaces the inline file specification with the name of the inline file being created. The effect is the same as if a filename was literally specified in the commands section.

The *filename* supplies a name for the inline file. It must immediately follow the angle brackets; no space is permitted. You can specify a path with the filename. No extension is required or assumed. If a file by the same name already exists, NMAKE overwrites it; such a file is deleted if the inline file is temporary. (Temporary inline files are discussed in the next section.)

A name is optional; if you don't specify *filename*, NMAKE gives the inline file a unique name. If *filename* is specified, NMAKE places the file in the directory specified with the name or in the current directory if no path is specified. If *filename* is not specified, NMAKE places the inline file in the directory specified by the TMP environment variable or in the current directory if TMP is not defined. You can reuse a previous inline *filename*; NMAKE overwrites the previous file.

Creating an Inline File

The instructions for creating the inline file begin on the first line after the <<[[*filename*]] specification. The syntax to create the inline file is:

<<[[*filename*]]
inlinetext
.
.
.
<<[[**KEEP** | **NOKEEP**]]

The set of angle brackets marking the end of the inline file must appear at the beginning of a separate line in the makefile. All *inlinetext* before the delimiting angle brackets is placed in the inline file. The text can contain macro expansions and substitutions. Directives and comments are not permitted in an inline file; NMAKE treats them as literal text. Spaces, tabs, and newline characters are treated literally.

The inline file can be temporary or permanent. To retain the file after the end of the NMAKE session, specify **KEEP** immediately after the closing set of angle brackets. If you don't specify a preference, or if you specify **NOKEEP** (the default), the file is temporary. **KEEP** and **NOKEEP** are not case sensitive. The temporary file exists for the duration of the NMAKE session.

It is possible to specify **KEEP** for a file that you do not name; in this case, the NMAKE-generated filename appears in the appropriate directory after the NMAKE session.

Example

The following makefile uses a temporary inline file to clear the screen and then display the contents of the current directory:

```
COMMANDS = cls ^
dir
showdir :
    <<showdir.bat
$(COMMANDS)
<<
```

In this example, the name of the inline file serves as the only command in the description block. This command has the same effect as running a batch file named SHOWDIR.BAT that contains the same commands as those listed in the macro definition.

Reusing an Inline File

After an inline file is created, you can use it more than once. To reuse an inline file in the command in which it is created, you must supply a *filename* for the file where it is defined and first used. You can then reuse the name later in the same command.

You can also reuse an inline file in subsequent commands in the same description block or elsewhere in the makefile. Be sure that the command that creates the inline file executes before all commands that use the file. Regardless of whether you specify **KEEP** or **NOKEEP**, NMAKE keeps the file for the duration of the NMAKE session.

Example

The following makefile creates a temporary LIB response file named LIB.LRF:

```
OBJECTS = add.obj sub.obj mul.obj div.obj
math.lib : $(OBJECTS)
    LIB math.lib @<<lib.lrf
-+$(?: = &^
-+)
listing;
<<
    copy lib.lrf \projinfo\lib.lrf
```

The resulting response file tells LIB which library to use, the commands to execute, and the name of the listing file to produce:

```
-+add.obj &
-+sub.obj &
-+mul.obj &
-+div.obj
listing;
```

The second command in the descripton block tells NMAKE to copy the response file to another directory.

Using Multiple Inline Files

You can specify more than one inline file in a single command line. For each inline specification, specify one or more lines of inline text followed by a closing line containing the delimiter. Begin the second file's text on the line following the delimiting line for the first file.

Example

The following example creates two inline files:

```
target.abc : depend.xyz
    copy <<file1 + <<file2 both.txt
I am the contents of file1.
<<
I am the contents of file2.
<<KEEP
```

This is equivalent to specifying

```
copy file1 + file2 both.txt
```

to concatenate two files, where FILE1 contains

```
I am the contents of file1.
```

and FILE2 contains

```
I am the contents of file2.
```

The **KEEP** keyword tells NMAKE not to delete FILE2. After the NMAKE session, the files FILE2 and BOTH.TXT exist in the current directory.

Macros

Macros offer a convenient way to replace a particular string in the makefile with another string. You can define your own macros or use predefined macros. Macros are useful for a variety of tasks, such as:

- Creating a single makefile that works for several projects. You can define a macro that replaces a dummy filename in the makefile 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 makefile in a single step.
- Specifying paths in an inference rule. (For an example, see Example 3 in "User-Defined Inference Rules" on page 567.)

This section describes user-defined macros, shows how to use a macro, and discusses the macros that have special meaning for NMAKE. It ends by discussing macro substitutions, inherited macros, and precedence rules.

User-Defined Macros

To define a macro, use the following syntax:

macroname=string

The *macroname* can be any combination of letters, digits, and the underscore (_) character, up to 1024 characters. Macro names are case sensitive; NMAKE interprets `MyMacro` and `MYMACRO` as different macro names. The *macroname* can contain a macro invocation. If *macroname* consists entirely of an invoked macro, the macro being invoked cannot be null or undefined.

The *string* can be any sequence of zero or more characters up to 64K–25 (65,510 bytes). 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.

Other syntax rules, such as the use of spaces, apply depending on where you specify the macro; see "Where to Define Macros" on page 552. The *string* can contain a macro invocation.

Example

The following specification defines a macro named `DIR` and assigns to it a string that represents a directory.

```
DIR=c:\objects
```

Special Characters in Macros

Certain characters have special meaning within a macro definition. You use these characters to perform specific tasks. If you want one of these characters to have a literal meaning, you must specify it using a special syntax.

- To specify a comment with a macro definition, place a number sign (#) and the comment after the definition, as in:

  ```
  LINKCMD = link /CO  # Prepare for debugging
  ```

  NMAKE ignores the number sign and all characters up to the next newline character. To specify a literal number sign in a macro, use a caret (^), as in ^#.

- To extend a macro definition to a new line, end the line with a backslash (\ ). The newline character that follows the backslash is replaced with a space when the macro is expanded, as in the following example:

  ```
  LINKCMD = link myapp\
  another, , NUL, mylib, myapp
  ```

  When this macro is expanded, a space separates `myapp` and `another`.

To specify a literal backslash at the end of the line, precede it with a caret (^), as in:

```
exepath = c:\bin^\
```

You can also make a backslash literal by following it with a comment specifier (#). NMAKE interprets a backslash as literal if it is followed by any other character.

- To insert a literal newline character into a macro, end the line with a caret (^). The caret tells NMAKE to interpret the newline character as part of the macro, not as a line break ending the macro definition. The following example defines a macro composed of two operating-system commands separated by a newline character:

```
CMDS = cls^
dir
```

For an illustration of how this macro can be used, see the first example under "Inline Files" on page 548.

- To specify a literal dollar sign (**$**) in a macro definition, use two dollar signs (**$$**). NMAKE interprets a single dollar sign as the specifier for invoking a macro; see "Using Macros" on page 554.

For information on how to handle other special characters literally, regardless of whether they appear in a macro, see "Using Special Characters as Literals" on page 535.

Where to Define Macros

You can define macros in the makefile, on the command line, in a command file, or in TOOLS.INI. (For more information, see "Precedence Among Macro Definitions" on page 563.) Each macro defined in the makefile or in TOOLS.INI must appear on a separate line. The line cannot start with a space or tab.

When you define a macro in the makefile or in TOOLS.INI, NMAKE ignores any spaces or tabs 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). The macro name being defined must appear at the beginning of the line. Only one macro can be defined per line. For example, the following macro definition can appear in a makefile or TOOLS.INI:

```
LINKCMD = LINK /MAP
```

Slightly different rules apply when you define a macro on the NMAKE command line or in a command file. The command-line parser treats spaces and tabs as argument delimiters. Therefore, spaces must not precede or follow the equal sign. If *string* contains embedded spaces or tabs, either the string itself or the entire macro must be enclosed in double quotation marks ("). For example, 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 permitted. It contains spaces that are not enclosed by quotation marks:

```
NMAKE LINKCMD = "LINK /MAP"
```

Null Macros and Undefined Macros

An undefined macro is not the same thing as a macro defined to be null. Both kinds of macros expand to a null string. However, a macro defined to be null is still considered to be defined when used with preprocessing directives such as **!IFDEF**. A macro name can be "undefined" in a makefile by using the **!UNDEF** preprocessing directive. (For information on **!IFDEF** and **!UNDEF**, see "Preprocessing Directives" on page 572).

To define a macro to be null:

- In a makefile or TOOLS.INI, specify zero or more spaces between the equal sign (=) and the end of the line, as in the following:

  ```
  LINKOPTIONS =
  ```

- On the command line or in a command file, specify zero or more spaces enclosed in double quotation marks (" "), or specify the entire null definition enclosed in double quotation marks, as in either of the following:

  ```
  LINKOPTIONS=""
  "LINKOPTIONS ="
  ```

To undefine a macro in a makefile or in TOOLS.INF, use the **!UNDEF** preprocessing directive, as in:

```
!UNDEF LINKOPTIONS
```

Using Macros

To use a macro (defined or not), enclose its name in parentheses preceded by a dollar sign (**$**), as follows:

$(*macroname*)

No spaces are allowed. For example, you can use the `LINKCMD` macro defined as

```
LINKCMD = LINK /map
```

by specifying

```
$(LINKCMD)
```

NMAKE replaces the specification `$(LINKCMD)` with `LINK /map`.

If the name you use as a macro has never been defined, or was previously defined but is now undefined, NMAKE treats that name as a null string. No error occurs.

The parentheses are optional if *macroname* is a single character. For example, `$L` is equivalent to `$(L)`. However, parentheses are recommended for consistency and to avoid possible errors.

Example

The following makefile 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.

Special Macros

NMAKE provides several special macros to represent various filenames and commands. One use for these macros is in the predefined inference rules. (For

more information, see “Predefined Inference Rules” on page 567.) Like user-defined macro names, special macro names are case sensitive. For example, NMAKE interprets `CC` and `cc` as different macro names.

The following sections describe the four categories of special macros. The filename macros offer a convenient representation of filenames from a dependency line. The recursion macros allow you to call NMAKE from within your makefile. The command macros and options macros make it convenient for you to invoke the Microsoft language compilers.

Filename Macros

NMAKE provides macros that are predefined to represent filenames. The filenames are as you have specified them in the dependency line and not the full specification of the filenames as they exist on disk. 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.)

$@
: The current target’s full name (path, base name, and extension), as currently specified.

$$@
: The current target’s full name (path, base name, and extension), as currently specified. This macro is valid only for specifying a dependent in a dependency line.

$*
: The current target’s path and base name minus the file extension.

$**
: All dependents of the current target.

$?
: All dependents that have a later time stamp than the current target.

$<
: The dependent file that has a later time stamp than the current target. You can use this macro only in commands in inference rules.

Example 1

The following example uses the **$?** macro, which represents all dependents that have changed more recently than the target. The **!** command modifier causes NMAKE to execute a command once for each dependent in the list. 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 -+$?;
```

Example 2

In the next example, NMAKE updates a file in another directory by replacing it with a file of the same name from the current directory. The **$@** macro is used to represent the current target's full name.

```
# File in objects directory depends on version in current directory
DIR = c:\objects
$(DIR)\a.obj : a.obj
    COPY a.obj $@
```

Modifying Filename Macros

You can append one of the modifiers in the following table 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

Example 1

Assume that **$@** represents the target C:\SOURCE\PROG\SORT.OBJ. The following table shows the effect of combining each modifier with **$@**:

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 SORT, and the value of **$(@D)** is a period (**.**) to represent the current directory.

Example 2

The following example uses the **F** modifier to specify a file of the same name in the current directory:

```
# Files in objects directory depend on versions in current directory
DIR = c:\objects
$(DIR)\a.obj $(DIR)\b.obj $(DIR)\c.obj : $$(@F)
    COPY $(@F) $@
```

Note For another way to represent components of a filename, see "Filename-Parts Syntax" on page 546.

Recursion Macros

There are three macros that you can use when you want to call NMAKE recursively from within a makefile. These macros can make recursion more efficient.

MAKE
Defined as the name which you specified to the operating system when you ran NMAKE; this name is `NMAKE` unless you have renamed the NMAKE.EXE utility file. Use this macro to call NMAKE recursively. The /N command-line option to prevent execution of commands does not prevent this command from executing. It is recommended that you do not redefine **MAKE**.

MAKEDIR
Defined as the current directory when NMAKE was called.

MAKEFLAGS
Defined as the NMAKE options currently in effect. This macro is passed automatically when you call NMAKE recursively. However, specification of this macro when invoking recursion is harmless; thus, you can use older makefiles that specify this macro. You cannot redefine **MAKEFLAGS**. To change the /D, /I, /N, and /S options within a makefile, use the preprocessing directive **!CMDSWITCHES**. (See "Preprocessing Directives" on page 572.) To add other options to the ones already in effect for NMAKE when recursing, specify them as part of the recursion command.

Calling NMAKE Recursively

In a commands block, you can specify a call to NMAKE itself. Either invoke the MAKE macro or specify NMAKE literally. The following NMAKE information is available to the called NMAKE session during recursion:

- Environment-variable macros (see "Inherited Macros" on page 563). To cause all macros to be inherited, specify the /V option.
- The **MAKEFLAGS** macro. If **.IGNORE** (or **!CMDSWITCHES** +I) is set, **MAKEFLAGS** contains an I when it is passed to the recursive call. Likewise, if **.SILENT** (or **!CMDSWITCHES** +S) is set, **MAKEFLAGS** contains an S when passed to the call.
- Macros specified on the command line for the recursive call.
- All information in TOOLS.INI.

Inference rules defined in the makefile are not passed to the called NMAKE session. Settings for **.SUFFIXES** and **.PRECIOUS** are also not inherited. However, you can make **.SUFFIXES**, **.PRECIOUS**, and all inference rules available to the recursive call either by specifying them in TOOLS.INI or by placing them in a file that is specified in an **!INCLUDE** directive in the makefile for each NMAKE session.

Example

The **MAKE** macro is useful for building different versions of a program. The following makefile calls NMAKE recursively to build targets in the \VERS1 and \VERS2 directories.

```
all : vers1 vers2

vers1 :
    cd \vers1
    $(MAKE)
    cd ..

vers2 :
    cd \vers2
    $(MAKE) /F vers2.mak
    cd ..
```

If the dependency containing `vers1` as a target is executed, NMAKE performs the commands to change to the \VERS1 directory and call itself recursively using the MAKEFILE in that directory. If the dependency containing `vers2` as a target is executed, NMAKE changes to the \VERS2 directory and calls itself using the file VERS2.MAK in that directory.

Command Macros

NMAKE predefines several macros to represent commands for Microsoft products. You can use these macros as commands in either a description block or an inference rule; they are automatically used in NMAKE's predefined inference rules. (See "Inference Rules" on page 563.) You can redefine these macros to represent part or all of a command line, including options.

AS
Defined as `ml`, the command to run the Microsoft Macro Assembler

BC
Defined as `bc`, the command to run the Microsoft Basic Compiler

CC
Defined as `cl`, the command to run the Microsoft C Compiler

COBOL
Defined as `cobol`, the command to run the Microsoft COBOL Compiler

CPP
Defined as `cl`, the command to run the Microsoft C++ Compiler

CXX
Defined as `cl`, the command to run the Microsoft C++ Compiler

FOR
Defined as `fl`, the command to run the Microsoft FORTRAN Compiler

PASCAL
Defined as `pl`, the command to run the Microsoft Pascal Compiler

RC
Defined as `rc`, the command to run the Microsoft Resource Compiler

Options Macros

The following macros represent options to be passed to the commands for invoking the Microsoft language compilers. These macros are used automatically in the predefined inference rules. (See "Predefined Inference Rules" on page 567.) By default, these macros are undefined. You can define them to mean the options you want to pass to the compilers, and you can use these macros in commands in description blocks and inference rules. As with all macros, the options macros can be used even if they are undefined; a macro that is undefined or defined to be a null string generates a null string where it is used.

AFLAGS
Passes options to the Microsoft Macro Assembler

BFLAGS
Passes options to the Microsoft Basic Compiler

CFLAGS
Passes options to the Microsoft C Compiler

COBFLAGS
Passes options to the Microsoft COBOL Compiler

CPPFLAGS
Passes options to the Microsoft C++ Compiler

CXXFLAGS
Passes options to the Microsoft C++ Compiler

FFLAGS
Passes options to the Microsoft FORTRAN Compiler

PFLAGS
Passes options to the Microsoft Pascal Compiler

RFLAGS
Passes options to the Microsoft Resource Compiler

Substitution Within Macros

Just as macros allow you to substitute text in a makefile, you can also substitute text within a macro itself. The substitution applies only to the current use of the macro and does not modify the original macro definition. To substitute text within a macro, use the following syntax:

$(*macroname***:***string1***=***string2***)**

Every occurrence of *string1* is replaced by *string2* in the macro *macroname*. Do not put any spaces or tabs before the colon. Spaces that appear after the colon are interpreted as part of the string in which they occur. If *string2* is a null string, all occurrences of *string1* are deleted from the *macroname* macro.

Macro substitution is literal and case sensitive. This means that the case as well as the characters in *string1* must match the target string in the macro exactly, or the substitution is not performed. This also means that *string2* is substituted exactly as it is specified. Because substitution is literal, the strings cannot contain macro expansions.

Example 1

The following makefile illustrates macro substitution:

```
SOURCES = project.c one.c two.c

project.exe : $(SOURCES:.c=.obj)
    LINK $**;
```

The predefined macro **$**** stands for the names of all the dependent files (See "Filename Macros" on page 555.) When this makefile is run, NMAKE executes the following command:

```
LINK project.obj one.obj two.obj;
```

The macro substitution does not alter the `SOURCES` macro definition; if it is used again elsewhere in the makefile, `SOURCES` has its original value as it was defined.

Example 2

If the macro OBJS is defined as

```
OBJS = ONE.OBJ TWO.OBJ THREE.OBJ
```

you can replace each space in the defined value of `OBJS` with a space, followed by a plus sign, followed by a newline character, by using

```
$(OBJS: = +^
)
```

The caret (^) tells NMAKE to treat the end of the line as a literal newline character. The expanded macro after substitution is:

```
ONE.OBJ +
TWO.OBJ +
THREE.OBJ
```

This example is useful for creating response files.

Substitution Within Predefined Macros

You can also substitute text in any predefined macro (except **$$@**) using the same syntax as for other macros.

The command in the following description block makes a substitution within the predefined macro **$@**, which represents the full name of the current target. Note that although **$@** is a single-character macro, when it is used in a substitution, it must be enclosed in parentheses.

```
target.abc : depend.xyz
    echo $(@:targ=blank)
```

NMAKE substitutes `blank` for `targ` in the target, resulting in the string `blanket.abc`. If dependent `depend.xyz` has a later time stamp than target `target.abc`, then NMAKE executes the command

```
echo blanket.abc
```

Environment-Variable Macros

When NMAKE executes, it inherits macro definitions equivalent to every environment variable that existed before the start of the NMAKE session. If a variable such as LIB or INCLUDE has been set in the operating-system environment, you can use its value as if you had specified an NMAKE macro with the same name and value. The inherited macro names are converted to uppercase. Inheritance occurs before preprocessing. The /E option causes macros inherited from environment variables to override any macros with the same name in the makefile.

You can redefine environment-variable macros the same way that you define or redefine other macros. Changing a macro does not change the corresponding environment variable; to change the variable, use a SET command. Also, using the SET command to change an environment variable in an NMAKE session does not change the corresponding macro; to change the macro, use a macro definition.

If an environment variable has not been set in the operating-system environment, it cannot be set using a macro definition. However, you can use a SET command in the NMAKE session to set the variable. The variable is then in effect for the rest of the NMAKE session unless redefined or cleared by a later SET command. A SET definition that appears in a makefile does not create a corresponding macro for that variable name; if you want a macro for an environment variable that is created during an NMAKE session, you must explicitly define the macro in addition to setting the variable.

If an environment variable is defined as a string that would be syntactically incorrect in a makefile, NMAKE does not create a macro from that variable. No warning is generated.

Warning If an environment variable contains a dollar sign (**$**), NMAKE interprets it as the beginning of a macro invocation. The resulting macro expansion can cause unexpected behavior and possibly an error.

Example

The following makefile redefines the environment-variable macro called `LIB`:

```
LIB = c:\tools\lib

sample.exe : sample.obj
    LINK sample;
```

No matter what value the environment variable LIB had before, it has the value `c:\tools\lib` when NMAKE executes the LINK command in this description block. Redefining the inherited macro does not affect the original environment variable; when NMAKE terminates, LIB still has its original value.

If LIB is not defined before the NMAKE session, the LIB macro definition in the preceding example does not set a LIB environment variable for the LINK command. To do this, use the following makefile:

```
sample.exe : sample.obj
    SET LIB=c:\tools.lib
    LINK sample;
```

Inherited Macros

When NMAKE is called recursively, the only macros that are inherited by the called NMAKE are those defined on the command line or in environment variables. Macros defined in the makefile are not inherited when NMAKE is called recursively. There are several ways to pass macros to a recursive NMAKE session:

- Run NMAKE with the /V option. This option causes all macros to be inherited by the recursively called NMAKE. You can use this option on the NMAKE command for the entire session, or you can specify it in a command for a recursive NMAKE call to affect just the specified recursive session.
- Use the SET command before the recursive call to set an environment variable before the called NMAKE session.
- Define a macro on the command line for the recursive call.
- Define a macro in the TOOLS.INI file. Each time NMAKE is recursively called, it reads TOOLS.INI.

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 makefile or include file
3. An inherited environment-variable macro
4. A macro defined in the TOOLS.INI file
5. A predefined macro, such as **AS** or **CC**

The /E option causes macros inherited from environment variables to override any macros with the same name in the makefile. The **!UNDEF** directive in a makefile overrides a macro defined on the command line.

Inference Rules

Inference rules are templates that define how a file with one extension is created from a file with another extension. NMAKE uses inference rules to supply commands for updating targets and to infer dependents for targets. In the dependency tree, inference rules cause targets to have inferred dependents as well as explicitly specified dependents; see “Inferred Dependents” on page 569.

The **.SUFFIXES** list determines priorities for applying inference rules; see "Dot Directives" on page 570.

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. Inference rules can be specified in the makefile or in TOOLS.INI.

Inference rules can be used in the following situations:

- If NMAKE encounters a description block that has no commands, it checks the **.SUFFIXES** list and the files in the current or specified directory and then searches for an inference rule that matches the extensions of the target and an existing dependent file with the highest possible **.SUFFIXES** priority.
- If a dependent file doesn't exist and is not listed as a target in another description block, NMAKE looks for an inference rule that shows how to create the missing dependent from another file with the same base name.
- If a target has no dependents and its description block has no commands, NMAKE can use an inference rule to create the target.
- If a target is specified on the command line and there is no makefile (or no mention of the target in the makefile), inference rules are used to build the target.

If a target is used in more than one single-colon dependency, an inference rule might not be applied as expected; see "Accumulating Targets in Dependencies" on page 539.

Inference Rule Syntax

To define an inference rule, use the following syntax:

.*fromext***.***toext***:**
    commands

The first line lists two extensions: *fromext* represents the extension of a dependent file, and *toext* represents the extension of a target file. Extensions are not case sensitive. Macros can be invoked to represent *fromext* and *toext*; the macros are expanded during preprocessing.

The period (**.**) preceding *fromext* must appear at the beginning of the line. The colon (**:**) can be preceded by zero or more spaces or tabs; it can be followed only by spaces or tabs, a semicolon (**;**) to specify a command, a number sign (**#**) to specify a comment, or a newline character. No other spaces are allowed.

The rest of the inference rule gives the commands to be run if the dependency is out-of-date. Use the same rules for commands in inference rules as in description blocks. (See “Commands” on page 543.)

An inference rule can be used only when a target and dependent have the same base name. You cannot use a rule to match multiple targets or dependents. For example, you cannot define an inference rule that replaces several modules in a library because all but one of the modules must have a different base name from the target library.

Inference rules can exist only for dependents with extensions that are listed in the **.SUFFIXES** directive. (For information on **.SUFFIXES**, see “Dot Directives” on page 570.) If an out-of-date dependency does not have a commands block, and if the **.SUFFIXES** list contains the extension of the dependent, NMAKE looks for an inference rule matching the extensions of the target and of an existing file in the current or specified directory. If more than one rule matches existing dependent files, NMAKE uses the order of the **.SUFFIXES** list to determine which rule to invoke. Priority in the list descends from left to right. NMAKE may invoke a rule for an inferred dependent even if an explicit dependent is specified; for more information, see “Inferred Dependents” on page 569.

Inference rules tell NMAKE how to build a target specified on the command line if no makefile is provided or if the makefile does not have a dependency containing the specified target. When a target is specified on the command line and NMAKE cannot find a description block to run, it looks for an inference rule to tell it how to build the target. You can run NMAKE without a makefile if the inference rules that are predefined or defined in TOOLS.INI are all you need for your build.

Inference Rule Search Paths

The inference-rule syntax described previously 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. An inference rule that specifies paths has the following syntax:

{*frompath***}.***fromext***{***topath***}.***toext***:**
    commands

No spaces are allowed. The *frompath* directory must match the directory specified for the dependent file; similarly, *topath* must match the target’s directory specification. For NMAKE to apply an inference rule to a dependency, the paths in the dependency line must match the paths specified in the inference rule exactly. For example, if the current directory is called PROJ, the inference rule

```
{..\proj}.exe{..\proj}.obj:
```

does not apply to the dependency

```
project1.exe : project1.obj
```

If you use a path on one extension in the inference rule, you must use paths on both. You can specify the current directory by either a period (**.**) or an empty pair of braces (**{}**).

You can specify only one path for each extension in an inference rule. To specify more than one path, you must create a separate inference rule for each path.

Macros can be invoked to represent *frompath* and *topath*; the macros are expanded during preprocessing.

User-Defined Inference Rules

The following examples illustrate several ways to write inference rules.

Example 1

The following makefile contains an inference rule and a minimal description block:

```
.c.obj:
    cl /c $<

sample.obj :
```

The inference rule tells NMAKE how to build a .OBJ file from a .C file. The predefined macro **$<** represents the name of a dependent that has a later time stamp than the target. The description block lists only a target, SAMPLE.OBJ; there is no dependent or command. However, given the target's base name and extension, plus the inference rule, NMAKE has enough information to build the target.

After checking to be sure that `.c` is one of the extensions in the **.SUFFIXES** list, NMAKE looks for a file with the same base name as the target and with the .C extension. 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 period. 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 for .c.obj when building the target.

Example 3

The following inference rule uses macros to specify paths in an inference rule:

```
C_DIR = proj1src
OBJ_DIR = proj1obj
{$(C_DIR)}.c{$(OBJ_DIR)}.obj:
    cl /c $
```

If the macros are redefined, NMAKE uses the definition that is current at that point during preprocessing. To reuse an inference rule with different macro definitions, you must repeat the rule after the new definition:

```
C_DIR = proj1src
OBJ_DIR = proj1obj
{$(C_DIR)}.c{$(OBJ_DIR)}.obj:
    cl /c $<
C_DIR = proj2src
OBJ_DIR = proj2obj
{$(C_DIR)}.c{$(OBJ_DIR)}.obj:
    cl /c $<
```

Predefined Inference Rules

NMAKE provides predefined inference rules containing commands for creating object, executable, and resource files. Table 16.1 describes the predefined inference rules.

Table 16.1 Predefined Inference Rules

Rule	Command	Default Action
.asm.exe	$(AS) $(AFLAGS) $*.asm	ML $*.ASM
.asm.obj	$(AS) $(AFLAGS) /c $*.asm	ML /c $*.ASM
.c.exe	$(CC) $(CFLAGS) $*.c	CL $*.C
.c.obj	$(CC) $(CFLAGS) /c $*.c	CL /c $*.C
.cpp.exe	$(CPP) $(CPPFLAGS) $*.cpp	CL $*.CPP

Table 16.1 Predefined Inference Rules (*continued*)

Rule	Command	Default Action
.cpp.obj	$(CPP) $(CPPFLAGS) /c $*.cpp	CL /c $*.CPP
.cxx.exe	$(CXX) $(CXXFLAGS) $*.cxx	CL $*.CXX
.cxx.obj	$(CXX) $(CXXFLAGS) /c $*.cxx	CL /c $*.CXX
.bas.obj	$(BC) $(BFLAGS) $*.bas;	BC $*.BAS;
.cbl.exe	$(COBOL) $(COBFLAGS) $*.cbl, $*.exe;	COBOL $*.CBL, $*.EXE;
.cbl.obj	$(COBOL) $(COBFLAGS) $*.cbl;	COBOL $*.CBL;
.for.exe	$(FOR) $(FFLAGS) $*.for	FL $*.FOR
.for.obj	$(FOR) /c $(FFLAGS) $*.for	FL /c $*.FOR
.pas.exe	$(PASCAL) $(PFLAGS) $*.pas	PL $*.PAS
.pas.obj	$(PASCAL) /c $(PFLAGS) $*.pas	PL /c $*.PAS
.rc.res	$(RC) $(RFLAGS) /r $*	RC /r $*

For example, assume you have the following makefile:

```
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 16.1 shows that more than one inference rule exists for building an .EXE file. NMAKE uses the order of the extensions appearing in the **.SUFFIXES** list to determine which rule to invoke. It then looks in the current or specified directory for a file that has the same base name as the target `sample` and one of the extensions in the **.SUFFIXES** list; it checks the extensions one by one until it finds a matching dependent file in the directory.

For example, if a file called SAMPLE.ASM exists, NMAKE applies the `.asm.exe` inference rule. If both SAMPLE.C and SAMPLE.ASM exist, and if `.c` appears before `.asm` in the **.SUFFIXES** list, 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 (**AFLAGS**, **CFLAGS**, and so on) are undefined. As explained in "Using Macros" on page 554, 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.

Inferred Dependents

NMAKE can assume an "inferred dependent" for a target if there is an applicable inference rule. An inference rule is applicable if:

- The *toext* in the rule matches the extension of the target being evaluated.
- The *fromext* in the rule matches the extension of a file that has the same base name as the target and that exists in the current or specified directory.
- The *fromext* is in the **.SUFFIXES** list.
- No other *fromext* in a matching rule is listed in **.SUFFIXES** with a higher priority.
- No explicitly specified dependent has a higher priority extension.

If an existing dependent matches an inference rule and has an extension with a higher **.SUFFIXES** priority, NMAKE does not infer a dependent.

NMAKE does not necessarily execute the commands block in an inference rule for an inferred dependent. If the target's description block contains commands, NMAKE executes the description block's commands and not the commands in the inference rule. The effect of an inferred dependent is illustrated in the following example:

```
project.obj :
    cl /Zi /c project.c
```

If a makefile contains this description block and if the current directory contains a file named PROJECT.C and no other files, NMAKE uses the predefined inference rule for `.c.obj` to infer the dependent `project.c`. It does not execute the predefined rule's command, `cl /c project.c`. Instead, it runs the command specified in the makefile.

Inferred dependents can cause unexpected side effects. In the following examples, assume that both PROJECT.ASM and PROJECT.C exist and that **.SUFFIXES** contains the default setting. If the makefile contains

```
project.obj : project.c
```

NMAKE infers the dependent `project.asm` ahead of `project.c` because **.SUFFIXES** lists `.asm` before `.c` and because a rule for `.asm.obj` exists. If either PROJECT.ASM or PROJECT.C is out-of-date, NMAKE executes the commands in the rule for `.asm.obj`.

However, if the dependency in the preceding example is followed by a commands block, NMAKE executes those commands and not the commands in the inference rule for the inferred dependent.

Another side effect occurs because NMAKE builds a target if it is out-of-date with respect to any of its dependents, whether explicitly specified or inferred. For example, if PROJECT.OBJ is up-to-date with respect to PROJECT.C but not with respect to PROJECT.ASM, and if the makefile contains

```
project.obj : project.c
    cl /Zi /c project.c
```

NMAKE infers the dependent `project.asm` and updates the target using the command specified in this description block.

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 as follows:

1. An inference rule defined in the makefile. If more than one rule is defined, the last rule applies.
2. An inference rule defined in the TOOLS.INI file. If more than one rule is defined, the last rule 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 match a target's extension and a dependent is not specified, NMAKE uses the inference rule whose dependent's extension appears first in the **.SUFFIXES** list.

Directives

NMAKE provides several ways to control the NMAKE session through dot directives and preprocessing directives. Directives are instructions to NMAKE that are placed in the makefile or in TOOLS.INI. NMAKE interprets dot directives and preprocessing directives and applies the results to the makefile before processing dependencies and commands.

Dot Directives

Dot directives must appear outside a description block and must appear at the beginning of a line. Dot directives begin with a period (**.**) and are followed by a colon (**:**). Spaces and tabs can precede and follow the colon. These directive names are case sensitive and must be uppercase.

.IGNORE :

Ignores nonzero exit codes returned by programs called from the makefile. By default, NMAKE halts if a command returns a nonzero exit code. This directive affects the makefile from the place it is specified to the end of the file. To turn it off again, use the **!CMDSWITCHES** preprocessing directive. To ignore the exit code for a single command, use the dash (**–**) command modifier. To ignore exit codes for an entire file, invoke NMAKE with the /I option.

.PRECIOUS : *targets*

Tells NMAKE not to delete *targets* if the commands that build them are interrupted. This directive has no effect if a command is interrupted and handles the interrupt by deleting the file. Separate the target names with one or more spaces or tabs. By default, NMAKE deletes the target if building was interrupted by CTRL+C or CTRL+BREAK. Multiple specifications are cumulative; each use of **.PRECIOUS** applies to the entire makefile.

.SILENT :

Suppresses display of the command lines as they are executed. By default, NMAKE displays the commands it invokes. This directive affects the makefile from the place it is specified to the end of the file. To turn it off again, use the **!CMDSWITCHES** preprocessing directive. To suppress display of a single command line, use the **@** command modifier. To suppress the command display for an entire file, invoke NMAKE with the /S option.

.SUFFIXES : *list*

Lists file suffixes (extensions) for NMAKE to try to match when it attempts to apply an inference rule. (For details about using **.SUFFIXES**, see "Inference Rules" on page 563.) The list is predefined as follows:

```
.SUFFIXES : .exe .obj .asm .c .cpp .cxx .bas .cbl .for .pas .res .rc
```

To add additional suffixes to the end of the list, specify

.SUFFIXES : *suffixlist*

where *suffixlist* is a list of the additional suffixes, separated by one or more spaces or tabs. To clear the list, specify

```
.SUFFIXES :
```

without extensions. To change the list order or to specify an entirely new list, you must clear the list and specify a new setting. To see the current setting, run NMAKE with the /P option.

Preprocessing Directives

NMAKE preprocessing directives are similar to compiler preprocessing directives. You can use several of the directives to conditionally process the makefile. 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 makefile as a whole.

Preprocessing directives begin with an exclamation point (**!**), which must appear at the beginning of the line. Zero or more spaces or tabs can appear between the exclamation point and the directive keyword; this allows indentation for readability. These directives (and their keywords and operators) are not case sensitive.

!CMDSWITCHES {+|–}*opt*...

Turns on or off one or more options. (For descriptions of options, see page 529.) Specify an operator, either a plus sign (+) to turn options on or a minus sign (–) to turn options off, followed by one or more letters representing options. Letters are not case sensitive. Do not specify the slash (/). Separate the directive from the operator by one or more spaces or tabs; no space can appear between the operator and the options. To turn on some options and turn off other options, use separate specifications of the **!CMDSWITCHES** directives.

All options with the exception of /F, /HELP, /NOLOGO, /X, and /? can appear in **!CMDSWITCHES** specifications in TOOLS.INI. In a makefile, only the letters D, I, N, and S can be specified. If **!CMDSWITCHES** is specified within a description block, the changes do not take effect until the next description block. This directive updates the **MAKEFLAGS** macro; the changes are inherited during recursion.

!ERROR *text*

Displays *text* to standard error in the message for error U1050, then stops the NMAKE session. This directive stops the build even if /K, /I, **.IGNORE**, **!CMDSWITCHES**, or the dash (–) command modifier is used. Spaces or tabs before *text* are ignored.

!MESSAGE *text*

Displays *text* to standard output, then continues the NMAKE session. Spaces or tabs before *text* are ignored.

!INCLUDE [[<]]*filename*[[>]]
Reads and evaluates the file *filename* as a makefile before continuing with the current makefile. NMAKE first looks for *filename* in the current directory if *filename* is specified without a path; if a path is specified, NMAKE looks in the specified directory. Next, if the **!INCLUDE** directive is itself contained in a file that is included, NMAKE looks for *filename* in the parent file's directory; this search is recursive, ending with the original makefile's directory. Finally, if *filename* is enclosed by angle brackets (< >), NMAKE searches in the directories specified by the **INCLUDE** macro. The **INCLUDE** macro is initially set to the value of the INCLUDE environment variable.

!IF *constantexpression*
Processes the statements between the **!IF** and the next **!ELSE** or **!ENDIF** if *constantexpression* evaluates to a nonzero value.

!IFDEF *macroname*
Processes the statements between the **!IFDEF** and the next **!ELSE** or **!ENDIF** if *macroname* is defined. NMAKE considers a macro with a null value to be defined.

!IFNDEF *macroname*
Processes the statements between the **!IFNDEF** and the next **!ELSE** or **!ENDIF** if *macroname* is not defined.

!ELSE [[**IF** *constantexpression*|**IFDEF** *macroname*|**IFNDEF** *macroname*]]
Processes the statements between the **!ELSE** and the next **!ENDIF** if the preceding **!IF**, **!IFDEF**, or **!IFNDEF** statement evaluated to zero. The optional keywords give further control of preprocessing.

!ELSEIF
Synonym for **!ELSE IF**.

!ELSEIFDEF
Synonym for **!ELSE IFDEF**.

!ELSEIFNDEF
Synonym for **!ELSE IFNDEF**.

!ENDIF
Marks the end of an **!IF**, **!IFDEF**, or **!IFNDEF** block. Anything following **!ENDIF** on the same line is ignored.

!UNDEF *macroname*
Undefines a macro by removing *macroname* from NMAKE's symbol table. (For more information, see "Null Macros and Undefined Macros" on page 553.)

Example

The following set of directives

```
!IF
!ELSE
!   IF
!   ENDIF
!ENDIF
```

is equivalent to the set of directives

```
!IF
!ELSE IF
!ENDIF
```

Expressions in Preprocessing

The *constantexpression* used with the **!IF** or **!ELSE IF** directives can consist of integer constants, string constants, or program invocations. You can group expressions by enclosing them in parentheses. NMAKE treats numbers as decimals unless they start with 0 (octal) or 0x (hexadecimal).

Expressions in NMAKE use C-style signed long integer arithmetic; numbers are represented in 32-bit two's-complement form and are in the range –2147483648 to 2147483647.

Two unary operators evaluate a condition and return a logical value of true (1) or false (0):

DEFINED (*macroname*)
: Evaluates to true if *macroname* is defined. In combination with the **!IF** or **!ELSE IF** directives, this operator is equivalent to the **!IFDEF** or **!ELSE IFDEF** directives. However, unlike these directives, **DEFINED** can be used in complex expressions using binary logical operators.

EXIST (*path*)
: Evaluates to true if *path* exists. **EXIST** can be used in complex expressions using binary logical operators. If *path* contains spaces (allowed in some file systems), enclose it in double quotation marks.

Integer constants can use the unary operators for numerical negation (**–**), one's complement (**~**), and logical negation (**!**).

Constant expressions can use any binary operator listed in Table 16.2. To compare two strings, use the equality (**==**) operator and the inequality (**!=**) operator. Enclose strings in double quotation marks.

Table 16.2 Binary Operators for Preprocessing

Operator	Description
+	Addition
–	Subtraction

*	Multiplication
/	Division
%	Modulus
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
&&	Logical AND
||	Logical OR

Table 16.2 Binary Operators for Preprocessing (*continued*)

Operator	Description
<<	Left shift
>>	Right shift
==	Equality
!=	Inequality
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

Example

The following example shows how preprocessing directives can be used to control whether the linker inserts debugging information into the .EXE file:

```
!INCLUDE <infrules.txt>
!CMDSWITCHES +D
winner.exe : winner.obj
!IF DEFINED(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 makefile. The **!CMDSWITCHES** directive sets the /D option, which displays the time stamps of the files as they are checked. The **!IF** directive checks to see if the macro `debug` is defined. If it is defined, the next **!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.

Executing a Program in Preprocessing

You can invoke a program or command from within NMAKE and use its exit code during preprocessing. NMAKE executes the command during preprocessing, and it replaces the specification in the makefile with the command's exit code. A nonzero exit code usually indicates an error. You can use this value in an expression to control preprocessing.

Specify the command, including any arguments, within brackets (**[ ]**). You can use macros in the command specification; NMAKE expands the macro before executing the command.

Example

The following part of a makefile tests the space on disk before continuing the NMAKE session:

```
!IF [c:\util\checkdsk] != 0
!    ERROR Not enough disk space; NMAKE terminating.
!ENDIF
```

Sequence of NMAKE Operations

When you write a complex makefile, it can be helpful to know the sequence in which NMAKE performs operations. This section describes those operations and their order.

When you run NMAKE from the command line, NMAKE's first task is to find the makefile:

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.

NMAKE then assigns macro definitions with the following precedence (highest to lowest):

1. Macros defined on the command line
2. Macros defined in a makefile 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 first in order of priority and then 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 makefile can be redefined; a macro is valid from the point it is defined until it is redefined or undefined.

NMAKE also assigns inference rules, using the following precedence (highest to lowest):

1. Inference rules defined in a makefile or include file
2. Inference rules defined in the TOOLS.INI file
3. Predefined inference rules (such as .asm.obj)

You can use command-line options to change some of these priorities.

- The /E option allows macros inherited from the environment to override macros defined in the makefile.
- The /R option tells NMAKE to ignore macros and inference rules that are defined in TOOLS.INI or are predefined.

Next, NMAKE evaluates any preprocessing directives. If an expression for conditional preprocessing contains a program in 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 makefile. Preprocessing determines the final makefile that NMAKE reads.

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 only the first target in the makefile. 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.

NMAKE updates a target by comparing its time stamp to the time stamp of each dependent of that target. A target is out-of-date if any dependent has a later time stamp; if the /B option is specified, a target is out-of-date if any dependent has a later or equal time stamp.

If the dependents of the targets are themselves out-of-date or do not exist, NMAKE updates them first. If the target has no explicit dependent, NMAKE looks for an inference rule that matches the target. If a rule exists, NMAKE updates the target using the commands given with the inference rule. If more than one rule applies to the target, NMAKE uses the priority in the **.SUFFIXES** list to determine which inference rule to use.

NMAKE normally stops processing the makefile when a command returns a nonzero exit code. In addition, if NMAKE cannot tell whether the target was built successfully, it deletes the target. The /I command-line option, **.IGNORE** directive, **!CMDSWITCHES** directive, and dash (**–**) command modifier all tell NMAKE to ignore error codes and attempt to continue processing. The /K option tells NMAKE to continue processing unrelated parts of the build if an error occurs. The **.PRECIOUS** directive prevents NMAKE from deleting a partially created target if you interrupt the build with CTRL+C or CTRL+BREAK. You can document errors by using the **!ERROR** directive to print descriptive text. The directive causes NMAKE to print some text, then stop the build.

A Sample NMAKE Makefile

The following example illustrates many of NMAKE's features. The makefile creates an executable file from C-language source files:

```
# This makefile 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 debugging information

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 makefile is named SAMPLE.MAK. To invoke it, enter

```
NMAKE /F SAMPLE.MAK all clean
```

NMAKE builds SAMPLE.EXE and deletes intermediate files.

Here is how the makefile works. The `CFLAGS`, `CODEVIEW`, and `LFLAGS` macros define the default options for the compiler, linker, and inclusion of debugging 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 debugging information.

The `OBJS` macro specifies the object files that make up the executable file 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 "Substitution Within Macros" on page 560.)

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` because `all` is the first target in the makefile.

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 can 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 because 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.

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 also require recompilation.

The `clean` pseudotarget deletes unneeded files after a build. The dash (–) command 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.

NMAKE Exit Codes

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
1	Incomplete build (issued only when /K is used)
2	Program error, possibly due to one of the following: • A syntax error in the makefile • An error or exit code from a command • An interruption by the user
4	System error—out of memory
255	Target is not up-to-date (issued only when /Q is used)

CHAPTER 17

Managing Libraries with LIB

This chapter describes the Microsoft Library Manager (LIB) version 3.20. LIB creates and manages standard libraries, which are used to resolve references to external routines and data during static linking.

Overview

LIB creates, organizes, and maintains standard libraries. Standard libraries are collections of compiled or assembled object modules that provide a common set of useful routines and data. You use these libraries to provide your program with the routines and data at link time; this is called static linking. After you have linked a program to a library, the program can use a routine or data item exactly as if it were included in the program.

With LIB you can create a library file, add modules to a library, and delete or replace them. You can combine libraries into one library file and copy or move a module to a separate object file. You can also produce a listing of all public symbols in the library modules.

LIB works with the following kinds of files:

- Object files in the Microsoft Relocatable Object-Module Format (OMF), which is based on the Intel 8086 OMF
- Standard libraries in Microsoft library format
- Import libraries created by the Microsoft Import Library Manager (IMPLIB)
- 286 XENIX archives and Intel-style libraries

This chapter distinguishes between an “object file” and an “object module.” An object file is an independent file that can have a full path and extension (usually .OBJ). An object module is an object file that has been incorporated into a library. Object modules in the library have only base names. For example, SORT is an object-module name, while B:\RUN\SORT.OBJ is an object-file name.

Running LIB

To run LIB, type `LIB` at the operating system prompt and press ENTER. You can provide input to LIB in three ways, separately or in combination:

- Specify input on the command line.
- Respond to the prompts that LIB displays.
- Specify a response file that contains the expected input.

The LIB Command Line

You can run LIB and specify all the input it needs from the command line. The LIB command line has the following form:

LIB *oldlibrary* [[*options*]] [[*commands*]] [[**,** [[*listfile*]] [[**,** [[*newlibrary*]]]]]] [[**;**]]

Fields must appear in order but can be left blank (except for *oldlibrary*). A semicolon (**;**) after any field terminates the command; LIB assumes defaults for any remaining fields. The fields are described in "Specifying LIB Fields," which begins on page 583.

To terminate the session at any time, press CTRL+C.

The following example instructs LIB to combine the object files FIRST.OBJ and SECOND.OBJ and to name the combined library THIRD.LIB:

```
LIB FIRST +SECOND, , THIRD
```

For a more detailed example of running LIB from the command line, see page 591.

LIB Command Prompts

If you do not specify all expected input on the command line and do not end the line with a semicolon, LIB asks you for the missing input by displaying four prompts. LIB waits for you to respond to each prompt and then asks for the next input. The responses you give to the LIB command prompts correspond to the fields on the LIB command line. The following list shows these correspondences:

Library name: *oldlibrary* [[*options*]]
Operations: *commands*
List file: *listfile*
Output library: *newlibrary*

You can select default responses to the remaining prompts at any time by typing a single semicolon (**;**) followed immediately by a carriage return. The defaults

for prompts are the same as the defaults for the corresponding command-line fields.

The following example specifies `THIRD` as the output library-file name at the prompt:

```
Output library: THIRD
```

For a more detailed example of how to use the LIB prompts, see page 591.

The LIB Response File

To run LIB without typing the full command line or responses to prompts, you can use a response file. You must first create a response file, which is a text file containing the command-line information; you can write and edit this file in PWB or use another editor. Then invoke LIB using the following command:

LIB *@responsefile*

The *responsefile* is the name of a text file containing some or all of the input expected by LIB. You can specify a full path with the filename. Precede it with an at sign (**@**).

You can also enter the name of the response file at any position in a command line or after any of LIB's prompts. The input from the response file is treated exactly as if it had been entered in the command line or after prompts. When you run LIB with a response file, LIB displays prompts followed by the input from the response file. If the response file does not contain all expected input and does not end with a semicolon, LIB prompts for the remaining responses.

Each input field in the response file must appear on a separate line or must be separated from other fields on the same line by a comma. A carriage return and linefeed combination is equivalent to pressing ENTER in response to a prompt or to entering a comma in a command line. Input must appear in the same order as in the command-line fields or at the LIB prompts.

The following response file tells LIB to add the object files CURSOR.OBJ and HEAP.OBJ as the last two modules in LIBFOR.LIB:

```
LIBFOR
+CURSOR +HEAP;
```

Specifying LIB Fields

For all three methods of input, LIB expects information to be specified in a definite order and organized into fields. This section describes the input fields in

the order required by LIB. The fields are *oldlibrary*, *options*, *commands*, *listfile*, and *newlibrary*.

The Library File

The *oldlibrary* field specifies the name of an existing library or a library to be created. If you omit the extension, LIB assumes an extension of .LIB. You can specify a full path with the filename.

Important The path and filename cannot contain a dash character (–). LIB interprets the dash as the LIB "delete" operator.

Creating a Library File

To create a new library file, give the name of the library file you want to create in the *oldlibrary* field of the command line or at the `Library name:` prompt. LIB supplies the .LIB extension.

The name of the new library file must not be the name of an existing file. If it is, LIB assumes that you want to change the existing file. When you give the name of a library file that does not currently exist, LIB displays the following prompt:

```
Library file does not exist. Create?
```

Press Y to create the file or N to terminate the library session. If the library name is followed immediately by commands, a comma, or a semicolon, LIB suppresses the message and assumes Y.

Performing Consistency Checks

If *oldlibrary* is followed immediately by a semicolon (**;**), LIB performs a consistency check on the specified library to see if all the modules in the library are in usable form. LIB prints a message only if it finds an invalid object module; no message appears if all modules are intact. LIB puts the message in the listing file if one is created; otherwise, it writes the message to the standard output.

The following example causes LIB to perform a consistency check of the library file FOR.LIB if the library file exists.

```
LIB FOR;
```

No other action is performed. LIB displays any consistency errors it finds and ends the session. If FOR.LIB does not exist, LIB creates an empty library file with that name.

LIB Options

Options are not case sensitive and can appear only between the *oldlibrary* and *commands* fields on the command line or at the `Library Name:` prompt following the *oldlibrary* specification. The option name must be preceded by a forward slash
(/) as the option specifier. (Do not use a dash, –, as the option specifier. LIB interprets a dash as the "delete" operator.) Options can be abbreviated to the shortest unique name; the brackets show the optional part of the name. This chapter uses meaningful yet legal forms of the option names, which may be longer than the shortest unique names. LIB has the following options:

/H[[ELP]]
: Calls the QuickHelp utility. If LIB cannot find the Help file or QuickHelp, it displays a brief summary of LIB command-line syntax.

/I[[GNORECASE]]
: Tells LIB to ignore case when comparing symbols. LIB does this by default. Use the /NOI option to create a library that is marked as case sensitive.

  Use /IGN when combining a case-sensitive library with others that are not case sensitive to create a new library that is not case sensitive. (See the /NOI option for more information.)

/NOE[[XTDICTIONARY]]
: Prevents LIB from creating an extended dictionary of cross-references between modules. LINK uses the extended dictionary to speed up a library search. (LINK also has an option called /NOE, where /NOE means "do not read an extended dictionary.")

  Creating an extended dictionary requires more memory. If LIB reports the error message `no more virtual memory`, either use /NOE or build the library with fewer modules.

/NOI[[GNORECASE]]
: Tells LIB to preserve case when comparing symbols. By default, LIB ignores case. Use /NOI when you have symbols that are the same except for case. (When LINK uses the library, it ignores case unless LINK's /NOI option is specified.)

  If a library is built with /NOI, the library is internally marked to indicate that case sensitivity is in effect. (Libraries for case-sensitive languages such as C are built with /NOI.) If you combine multiple libraries and any one of them is case sensitive, LIB marks the output library as case sensitive. To override this, use the /IGN option.

/NOL[[OGO]]
: Suppresses the LIB copyright message.

/P[[AGESIZE]]:*number*

Specifies the page size of a new library or changes the page size of an existing library. The *number* specifies the new page size in bytes. It must be an integer power of 2 between 16 and 32,768. The default page size is 16 bytes for a new library or the current page size for an existing library. Combined libraries take the largest component page size.

The page size of a library sets the alignment of modules stored in the library. Modules start at locations that are a multiple of the page size from the beginning of the file. When creating a library, LIB builds a dictionary, which holds the locations of each name in each module. Each location value represents the number of pages in the file. Because of this addressing method, a library with a large page size can hold more modules than a library with a smaller page size.

The page size also determines the maximum possible size of the .LIB file. This limit is *number* * 64K. For example, `/PAGE:32` limits the .LIB file to 2 megabytes (32 * 65,536 bytes). However, for each module in the library, an average of *number*/2 bytes of storage space is wasted. In most cases, a small page size is advantageous; you should use a small page size unless you need to put a very large number of modules in a library.

/?

Displays a brief summary of LIB command-line syntax.

LIB Commands

The *commands* field specifies five operations for performing library-management tasks with LIB and manipulating modules: add, delete, replace, copy, and move. These commands can be used on the command line or in a response file in response to the `Operations:` prompt. To use this field, type a command operator followed immediately by a module name or an object-file name. You can specify more than one operation in this field in any order. If you leave the *commands* field blank, LIB does not make any changes to *oldlibrary*.

If you have many operations to perform during a library session, you can use an ampersand (**&**) to extend the operations line. Type the ampersand after a module name or filename; do not put the ampersand between an operator and a name. Immediately after the ampersand, press ENTER and then continue to type the rest of the command line. You can use this technique on the command line or in response to a prompt. When the ampersand is entered at a prompt, it tells LIB to repeat the `Operations:` prompt. In a response file, begin a new line of commands after the ampersand. See the examples at the end of this chapter for an illustration of the use of the ampersand.

You can perform one or more library-management functions during a LIB session. For each session, LIB determines whether a new library is being created

or an existing library is being examined or modified. It then processes commands in the following order:

1. Deletion and move commands. LIB does not actually delete modules from the existing library file. Instead, it marks the selected modules for deletion, creates a new library file, and copies only the modules not marked for deletion into the new library file. If there are no deletion or move commands, LIB creates the new file by copying the original library file. (The *newlibrary* field, described on page 590, controls what happens to the existing library.)
2. Addition commands. Like deletions, additions are not performed on the original library file. Instead, the additional modules are appended to the end of the new library file.

As LIB carries out these commands, it reads the object modules in the library and checks them for validity. It then builds a dictionary, an extended dictionary (unless /NOE is specified), and a listing file (if a *listfile* is specified). The listing file contains a list of all public symbols and the names of the modules in which they are defined.

Important Paths and filenames specified with these commands cannot contain a dash character (–). LIB interprets the dash as the LIB "delete" operator.

The Add Command (+)

Use the add command to create a library file, to add a module, or to combine libraries. The command has the form:

+*name*

where *name* is the name of the object file or library file. If no extension is specified, LIB assumes .OBJ. You can specify a path with the filename.

Creating a New Library

Use the add command to create a new library from one or more object files. Specify the name of the new library in the *oldlibrary* field, then specify each object file's name preceded by a plus sign. In the following example, LIB is instructed to create the library file FIRST.LIB containing the object module called MORE:

```
LIB FIRST +MORE;
```

Adding Library Modules

Use the add command to add an object module to a library. Give the name of the object file to be added immediately following the plus sign. LIB adds object modules to the end of a library file.

LIB strips the drive, path, and extension from the object-file name and leaves only the base name. This becomes the name of the object module in the library. For example, if the object file B:\CURSOR.OBJ is added to a library file, the name of the corresponding object module is CURSOR.

In the following example, LIB is instructed to add the module MORE to the already existing library file FIRST.LIB:

```
LIB FIRST +MORE;
```

Combining Libraries

To combine the contents of two libraries, supply the name of a library instead of an object file. In addition to standard libraries, LIB lets you combine import libraries (created by IMPLIB), 286 XENIX archives, and Intel-format libraries.

Specify the plus sign followed by the name of the library whose contents you wish to add to the original library. You must include the .LIB extension of the library name. Otherwise, LIB assumes that the file is an object file and looks for the file with an .OBJ extension.

LIB adds the modules of the new library to the end of the original library. Note that the added library still exists as an independent library. LIB copies the modules without deleting them.

Once you have added the contents of a library or libraries, you can save the new, combined library under a new name by giving a new name in the *newlibrary* field. If you omit this field, LIB saves the combined library under the name of the original library, that is, the name given in the *oldlibrary* field. The original library is saved with the same base name and the extension .BAK.

The following example combines DRAW.LIB and CHART.LIB into a library with the filename GRAPHICS.LIB:

```
LIB DRAW +CHART.LIB, ,GRAPHICS
```

The Delete Command (–)

Use the delete command to delete an object module from a library. The command has the form:

–*name*

where *name* is the name of the module to be deleted. A module name does not have a path or extension; it is simply a name, such as CURSOR.

The following example tells LIB to delete the FLOAT module from the MATH.LIB library:

```
LIB MATH –FLOAT;
```

The Replace Command (–+)

Use the replace command to replace a module in the library. The command has the form:

–+*name*

where *name* is the name of the module to be replaced. A module name has no path and no extension. LIB deletes the given module and then appends the object file having the same name as the module. The object file is assumed to have an .OBJ extension and to reside in the current directory.

The following three examples of command lines are equivalent. All three instruct LIB to replace the HEAP module in the library LANG.LIB. LIB deletes the HEAP module from the library and then appends the object file HEAP.OBJ as a new module in the library. Delete operations are always carried out before add operations, regardless of the order in which they are specified.

```
LIB LANG -+HEAP;
LIB LANG -HEAP +HEAP;
LIB LANG +HEAP -HEAP;
```

The Copy Command (*)

Use the copy command to copy a module from the library file into a newly created object file of the same name. The command has the following form:

name

where *name* is the name of the module to be copied. The module remains in the library file. LIB names the object file by using the base name of the module and adding an .OBJ extension. It then puts it in the current directory. You cannot override this filename or location; however, you can later rename the file and copy or move it to any location. LIB writes the full name of the object file (including drive, path to the current directory, base name, and extension) into the header of the object file.

The Move Command (–*)

Use the move command to move an object module from the library file to an object file. The command has the form:

–***name***

where *name* is the name of the module to be moved. This operation is equivalent to copying the module to an object file using the copy command (*****) and then deleting the module from the library using the delete command (–).

The Cross-reference Listing

A cross-reference listing contains two lists in the following order:

1. An alphabetical list of all public symbols in the library. Each symbol name is followed by the name of the module in which it is defined.
2. A list of the modules in the library with the location and size of each. Under each module name is an alphabetical listing of the public symbols defined in that module.

Create a cross-reference listing by giving a name for the listing file in the *listfile* field of the command line or at the `List file:` prompt. To create it in a directory other than the current one, specify a full path for the listing file. LIB does not supply a default extension if you omit the extension. When you do not specify a filename, the default is the special file named NUL, which tells LIB not to create a listing.

The following example creates a listing called LCROSS.PUB. It does nothing else except perform a consistency check of the library file LANG.LIB.

```
LIB LANG, LCROSS.PUB;
```

Note Source code symbols less than 127 characters long can exceed 127 characters as an internal or decorated name in an object file. LIB may not be able to successfully build a library from an object file if it contains symbols with names longer than 127 characters.

The Output Library

The *newlibrary* field specifies a name for a changed library file. You can specify a full path with the filename. LIB does not supply a default extension if you omit the extension.

You can change an existing library file by giving the name of the library file at the `Library name:` prompt. All operations you specify in the *commands* field of the command line or at the `Operations:` prompt are performed on that library.

LIB keeps both the unchanged library file and the newly changed version; it copies the library and makes changes to the copy. (This prevents the loss of your original file if you terminate LIB before the session is finished.) It names the two versions as follows:

- If you specify the name of a new library file in the *newlibrary* field, the modified library is stored under that name, and the original library is preserved under its own name.

- If you leave the field blank, LIB replaces the original library file with the changed version of the library and saves the original library file with the extension .BAK. Either way, at the end of a session you have two library files: the changed version and the original version.

Note You need enough space on disk for both the original library file and the copy.

Examples

All the following examples instruct LIB to:

- Suppress the creation of an extended dictionary of cross-references.
- Move the module STUFF from the library FIRST.LIB to an object file called STUFF.OBJ; the module STUFF is deleted from the library.
- Copy the module MORE from the library to an object file called MORE.OBJ; the module MORE remains in the library.
- Delete the module HEAP from the library.
- Create a cross-listing file called CROSSLST.
- Name the revised library SECOND.LIB. The new library contains all the modules in FIRST.LIB except STUFF and HEAP.
- Leave the original library, FIRST.LIB, unchanged.

Command-Line Example

```
LIB FIRST /NOE -*STUFF *MORE &
-HEAP, CROSSLST, SECOND
```

LIB Prompt Example

```
Library Name: FIRST /NOE
Operations: -*STUFF *MORE &
Operations: -HEAP
List File: CROSSLST
Output file: SECOND
```

Response-File Example

```
FIRST /NOE
-*STUFF *MORE &
-HEAP
CROSSLST
SECOND
```

LIB Exit Codes

LIB returns an exit code (also called return code or error code) to the operating system or the calling program. You can use the exit code 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 utility produced the error.
4	System error. The library manager encountered one of the following problems: • There was insufficient memory. • An internal error occurred. • The user interrupted the session.

CHAPTER 18

Creating Help Files With HELPMAKE

This chapter describes how to create and modify Help files using the Microsoft Help File Maintenance Utility (HELPMAKE) version 1.08. A “Help file” is a file that can be read by the Microsoft Advisor Help system and QuickHelp. If you’ve used the Programmer’s WorkBench (PWB) or one of the Microsoft Quick languages, you already know the advantages of the Microsoft Advisor. HELPMAKE extends these advantages by allowing you to customize the Microsoft Help files or create your own Help files.

HELPMAKE translates Help source files to a Help database accessible within the following 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
- MS-DOS EDIT version 5.0
- MS-DOS QBasic version 5.0

Warning The PWB editor breaks lines longer than about 250 characters. Some Help sources contain lines longer than this. To edit files that have 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.

Overview

HELPMAKE creates a Help file by encoding a source file. A Help file contains information that can be read by a Help reader (sometimes referred to in this chapter as an application). Examples of Help readers are the Microsoft Advisor or Microsoft QuickHelp. Help files have an .HLP extension.

Source files for HELPMAKE are text files that contain topic text along with attributes and commands that tell HELPMAKE how to process the file. HELPMAKE encodes text files written in the following formats: QuickHelp, rich text format (RTF), and minimally formatted ASCII.

Encoding compresses the text and translates the commands into information for the Help reader. You can control the amount of compression and other aspects of encoding.

HELPMAKE can also decode an existing Help file. Decoding decompresses the text into ASCII format. Attributes and commands can be preserved or omitted during decoding. You can modify an existing Help file by using HELPMAKE to decode the file and then rebuild it into a different Help file. You can even modify
a Microsoft help file by decompressing it and then encoding it with your changes. Regardless of the source format, HELPMAKE always decodes a Help file into the QuickHelp format.

The basic unit of Help is the database. A Help database is an individual file created by HELPMAKE. At the time it is created, it is given an internal name that is the same as the filename on disk. If the file is later renamed, the database retains this internal name as it is known by HELPMAKE and the Help reader.

A Help system consists of one or more physical Help files that are available to a Help reader. A physical Help file is a file on disk with an .HLP extension. It can contain a single database (with either the same or a different filename) or multiple databases. To create a physical Help file that contains several Help databases, use the MS-DOS COPY command. Specify the /b modifier to combine them as binary files. You can merge several databases into one physical Help file, combine two or more physical Help files, or append a Help database to an existing physical Help file. For example, the following command concatenates three individual Help databases into a new physical Help file:

```
COPY help1.hlp /b + help2.hlp /b + help3.hlp /b myhelp.hlp
```

The next example merges the database `yourhelp.hlp` with the existing Help file `utils.hlp`:

```
COPY utils.hlp /b + yourhelp.hlp /b
```

It is recommended that you back up existing Help files before running the COPY command. You may need to concatenate Help files if you reach a limit on physical files imposed by your system or the Help reader.

You can use HELPMAKE to deconcatenate, or split, a physical Help file that contains multiple databases. If you want to decompress such a Help file, you must first split it and then decompress each database.

When designing a Help system, it is important to know that a single database is more efficient to search than multiple databases or physical Help files.

Running HELPMAKE

The following sections describe HELPMAKE's syntax and options for encoding a Help file, decoding or deconcatenating a Help file, and getting Help on HELPMAKE. Some options apply only to encoding, others apply only to decoding, and a few apply to both.

The following are some general rules for syntax:

- Options are not case sensitive. Precede each option with either a forward slash
  (/) or a dash (–).
- You can specify a path with a filename. Separate multiple filenames with spaces or tabs. Where multiple files can be specified, you can use wildcard characters (***** and **?**).

Encoding

To create a Help file, use the following syntax:

HELPMAKE /E[[*n*]] /*Ooutfile options sourcefiles*

The /E option encodes a Help source file and creates a compressed Help database. The *n* is a decimal number that specifies the type of compression. If *n* is omitted, HELPMAKE compresses the file as much as possible (about 50 percent). The value of *n* is in a range from 0 through 15, which represents the following compression techniques:

Value	Technique
0	No compression
1	Run-length compression
2	Keyword compression
4	Extended keyword compression

8 Huffman compression

You can add these values to combine compression techniques. For example, specify /E3 to get run-length and keyword compression. Use /E0 to create the database quickly during the testing stages of database creation when you are not yet concerned with size.

The /O option specifies a filename for the database. This option is required when encoding.

Additional options are discussed in the next section and in “Other Options” on page 599.

The *sourcefiles* field specifies one or more text files that contain Help source information.

Options for Encoding

The following options control encoding:

/A*c*

Specifies *c* as a control character for the Help database. A control character marks a line that contains special information for internal use by the Help reader. Control characters differ for each Help reader. For example, the Microsoft Advisor uses a colon (**:**) to indicate a command, so you must specify /A: when building a Help file for use with the Advisor. HELPMAKE assumes /A: if the /T option is specified.

/C

Makes context strings case sensitive.

/K*filename*

Optimizes keyword compression by supplying a list of characters to act as word separators. The *filename* is a text file that contains a list of separator characters.

HELPMAKE can apply “keyword compression” to words that occur 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. For example, a number sign (**#**) is treated as a separator by default. However, in a Help file about the C language, you might want to have HELPMAKE treat each directive such as **#include** as a keyword instead of as a separator followed by a word. To encode **#include** and other directives as keywords, create a separator list that omits the number sign:

! ” & ‘ ’ ( ) * + – , / : ; < = > ? @ [\ ] ^ _ { | } ~

ASCII characters in the range from 0 through 31 are always separators, so you do not need to list them. However, a customized list must include all other separators, including the space (ASCII character 32). If you omit the space, HELPMAKE will not use spaces as word separators.

/L

Locks the Help file so that it cannot be decoded later.

/S*n*

Specifies the type of input file, according to the following *n* values:

Option	File Type
/S1	Rich text format (RTF)
/S2	QuickHelp (the default)
/S3	Minimally formatted ASCII

/T

Translates dot commands into internal format. If your source file contains dot commands other than **.context** and **.comment**, you must supply this option. The /T option is required if you want to use commands in the QuickHelp dot format. Dot commands are described on page 605. HELPMAKE assumes the /A: option if /T is specified.

/W*width*

Sets the fixed width of the resulting Help text in number of characters. The *width* is a decimal number in a range from 11 through 255. If /W is omitted, the default width is 76. When encoding an RTF source (/S1), HELPMAKE wraps the text to *width* characters. When encoding QuickHelp (/S2) or minimally formatted ASCII (/S3) files, HELPMAKE truncates lines to this width.

Example

The following example invokes HELPMAKE with the /V, /E, and /O options:

```
HELPMAKE /E /V /Omy.hlp my.txt > my.log
```

HELPMAKE reads input from the source file `my.txt` and creates the compressed Help database `my.hlp` (/O option). The /E option, without a compression specification, maximizes compression. The /V option enables verbose output; the MS-DOS redirection symbol (>) sends a log of HELPMAKE diagnostic information to the file `my.log`, because the verbose mode can generate a lengthy log.

Decoding

To decode a Help file, use the following syntax:

HELPMAKE /D[[*c*]] [[/O*outfile*]] *options sourcefiles*

The /D option decodes a Help file or splits a concatenated file into its component databases. The /D option can take a qualifying character *c*, which is either S or U.

Specify /D without a qualifying character to fully decode a database into a text file that is in QuickHelp format, with all links and formatting information intact. If the physical Help file contains concatenated databases, only the first database is decoded.

Specify /DU to decompress the database and remove all screen formatting and links. If the physical Help file contains concatenated databases, only the first database is decoded.

Specify /DS to split (deconcatenate) a physical Help file that contains one or more databases. HELPMAKE creates a physical Help file for each database in the original Help file. The Help file is not decompressed. HELPMAKE names the deconcatenated files using the names of the databases. The deconcatenated files are placed in the current directory. If a database in the file has a name that matches the name of the original physical Help file, HELPMAKE issues an error. In this case, rename the physical Help file, or run HELPMAKE in another directory and specify a path with the source file. Do not use the /O option with /DS.

The /O option specifies a filename for the decoded file. If /O is not specified, HELPMAKE sends the text to standard output. This option is not valid when using /DS.

There is one option available to control decoding. The /T option translates commands from internal format to dot-command format. This option applies only when using /D. It is recommended to always use this option to make the resulting source file more readable.

Additional options are discussed in “Other Options” on page 599.

The *sourcefiles* field specifies one or more physical Help files.

Example

The following example decodes the help file `my.hlp` into the source file `my.src`:

```
HELPMAKE /D /T /Omy.src my.hlp
```

Getting Help

To get help on HELPMAKE, use the following syntax:

HELPMAKE {/H[[ELP]] | /?}

The following are the options for help:

/?
Displays a brief summary of the HELPMAKE command-line syntax and exits without encoding or decoding any files. All other information on the command line is ignored.

/H[[ELP]]
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.

Other Options

The following options apply whether encoding or decoding.

The /NOLOGO option

The /NOLOGO option suppresses the HELPMAKE copyright message.

The /V option

The /V option controls the verbosity of diagnostic and informational output. HELPMAKE sends this information to standard output. The syntax for /V is:

/V[[*n*]]

Specify /V without *n* to get a full output. The decimal number *n* controls the amount of information produced. Numbers in a range from 0 through 3 are valid only for decoding. The values of *n* are:

Option	Output
/V	Maximum diagnostic output
/V0	No diagnostic output and no banner
/V1	HELPMAKE banner only
/V2	Pass names
/V3	Context strings encountered on first pass
/V4	Context strings encountered on each pass
/V5	Any intermediate steps within each pass
/V6	Statistics on Help file and compression

Source File Formats

You can create Help source files for HELPMAKE in any of three formats. The QuickHelp format is the default format for encoding. When Help databases are decoded, the resulting text files are always in QuickHelp format. The discussion that follows uses QuickHelp format to describe how to create a Help source file. Later sections describe the two other formats: rich text format (RTF) and minimally formatted ASCII.

Rich text format is a Microsoft word-processing format that is supported by several word processors, including Microsoft Word version 5.0 and later and Microsoft Word for Windows. For more information, see "Rich Text Format" on page 609.

Minimally formatted ASCII files define contexts and their topic text. They cannot contain formatting commands or explicit links. For more information, see "Minimally Formatted ASCII" on page 612.

In addition to these three formats, you can link to unformatted ASCII files from within a Help database. Unformatted ASCII files are text files with no formatting commands, context definitions, or special information. You do not process unformatted ASCII files with HELPMAKE. An unformatted ASCII file does not become a database or part of a physical Help file. The file's name is used as the object of a link. For example, you can create a link to an include file or a program example. Any word that is an implicit link in other Help files is also an implicit link in unformatted ASCII files.

A Help system can use any combination of files with different format types.

Elements of a Help Source File

The following sections describe how to create the fundamental elements of a Help file.

Defining a Topic

A Help source file is a text file that consists of a sequence of topics. A topic is the fundamental unit of Help information. It is usually a screen of information about a particular subject.

Each topic begins with one or more consecutive **.context** statements or definitions. The topic consists of all subsequent lines up to the next **.context** statement. A context definition associates the topic with a "context string," which is the word or phrase for which you want to be able to request Help. When Help is requested on a context string, the Help reader displays the topic. A context definition has the following form:

```
.context string
```

The **.context** command defines a context string for the topic that follows it. A context string can contain one word or several words depending on the Help reader and the delimiters it understands. For example, because Microsoft QuickBasic considers spaces to be delimiters, a context string in a QuickBasic Help file 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 a topic that is marked with the requested context string.

For example, the following line introduces Help for the **#include** directive:

```
.context #include
```

A topic can be associated with more than one context string. For example, the C-language functions **strtod**, **strtol**, **_strtold**, and **strtoul** are described in a single topic, and each is defined in a separate **.context** command for that topic, as follows:

```
.context strtod
.context _strtold
.context strtol
.context strtoul
```

Warning HELPMAKE warns you if it encounters a duplicate context definition within a given Help source file. Each context string must be unique within a database. You cannot associate a single context string with several topics in a single database.

A context string can be global or local. The *string* for a local context is preceded by an at sign (**@**). For more information, see "Local Contexts" on page 603.

Creating Links to Other Topics

A topic can contain a link to another topic. Links let you navigate a Help database. When a topic is displayed, you can ask for Help on links contained in the topic. These links can be associated with other contexts in the same Help database, contexts in other Help databases, or even ASCII files on disk. You can view the cross-referenced material immediately by activating the link without having to search the Help system's indexes and tables of contents for the topic.

The keystroke that activates a link depends on the application. Consult the documentation for each product for the various ways to get Help on a link. In Microsoft language products, use ENTER, SPACEBAR, or F1. If the file that

contains the link's destination is not already open, the Help reader finds it and opens it.

The topic text can present the link in various ways, depending on how you want to design your Help system. The link can appear as a "See:" cross-reference, for example, or as a button that contains a title surrounded by special characters. It can even be undistinguished from surrounding text.

A link is either explicit (coded) or implicit (available without coding). It is associated with either a global context (visible throughout the Help system) or a local context (visible only in one database). The following sections discuss these features of links.

Explicit Links

An explicit link is a word or phrase coded with invisible text that provides the context to which the link refers or the action which the Help reader is to take. Use the \v formatting attribute to delimit the invisible text. Format the explicit link in the source file using the following syntax:

*string***v***text***v**

If *string* consists of more than one word, you must anchor the string with the \ a formatting attribute as follows:

a*string***v***text***v**

An anchored link must be specified entirely on one line.

The \v attributes surround the invisible *text*, which is one of the following commands to the Help reader:

contextstring
: Display the topic associated with *contextstring* when the link is activated. The context string must be available either as a local context in the same Help database or as a global context anywhere in the Help system. For a discussion of global and local contexts, see "Local Contexts" on page 603.

*helpfile***!***contextstring*
: Search *helpfile* for *contextstring* and display the topic associated with it. Only the specified Help database or physical Help file is searched for the context. Since *helpfile* is not in the local database, *contextstring* must be a global context. Use this specification to confine the search to a single database if a context is contained in more than one database and you want only one of the topics to be found.

*filename***!**
: Display *filename* as a single topic. The specified file must be a text file no larger than 64K.

!*command*
Execute the command specified after the exclamation point (**!**). The command is case sensitive. Commands are application-specific. For example, in the
Microsoft Advisor and QuickHelp, the command !B represents the previously accessed topic.

In the following example, the word `Example` is an explicit link:

```
\bSee also:\p Example\vopen.ex\v
```

The \v formatting attribute marks the explicit link in the Help text. The \b and \p are formatting attributes that mark `See also:` as bold text. (Formatting attributes are described on page 605.) The link refers to `open.ex`. On the screen, this line appears as follows:

```
See also: Example
```

If you select any letter in `Example` and request Help, the Help reader displays the topic whose context is `open.ex`.

To create an explicit link that contains more than one word, you must use an anchor, as in the following example:

```
\bSee also:\p \aExample 1\vopen.ex1\v, \aExample 2\vopen.ex2\v
```

The \ a attribute creates an anchor for the explicit link. The phrase following the \ a attribute refers to the context specified in the invisible text. The first \v attribute marks both the end of the anchored string and the beginning of the invisible text. The second \v attribute ends the invisible text. The anchored link must fit on one line.

Implicit Links

An implicit link is a single word for which a global context exists somewhere in the Help system. Any word that appears as a global context is implicitly linked. You do not code the word to create the link. When you ask for Help on a word that exists as an implicit link, the Help reader displays the topic that has a **.context** string that matches the selected word.

For example, suppose that the Help database contains a screen that starts with:

```
.context open
```

If you ask for Help on the word “open” (using the features for requesting Help that are available in your Help reader), the topic that begins with `.context open` is displayed. An explicit link to the topic is not necessary. For example, in PWB you can place the cursor on the word “open” as it appears in your source

file or in a displayed Help topic, then click the right mouse button or press F1. Thus, every occurrence of “open” is a potential implicit link.

Local Contexts

A “local context” is a context string that begins with an at sign (**@**). Local contexts use less file space and speed access. However, a local context has meaning only within the database in which it appears.

HELPMAKE encodes a local context as an internally generated number rather than a context string. This saves space in the database. Unlike a global context (a context string that is specified without the preceding **@**), a local context is not stored as a string. Thus, topics headed by local contexts can only be accessed using explicit links and cannot be accessed from another database. Local contexts are not restored as strings when a database is decompressed.

The following source file contains two topics, one marked with a global context and one marked with a local context:

```
.context Global
     This is a topic that is marked with a global context.
     It is accessed using the context string "Global". It
     contains a link to a topic marked with a local context.
     See: \aA Local Topic\v@Local\v
.context @Local
     This topic can be reached only by the explicit link in
     the previous topic (or by sequentially browsing the file).
```

The text `A Local Topic` is explicitly linked to `@local`, which is a local context. If the user asks for Help on the text or scrolls through the Help file, the Help reader displays the topic text that follows the context definition for `@local`. This topic cannot be accessed any other way (except by sequentially browsing the database).

If you want a topic to be accessible in both local and global contexts, mark the topic text with both global and local **.context** statements:

```
.context Global
.context @Local
     This is a topic that is marked with a global context and
     a local context. It can be accessed using the context
     string "Global" (as an explicit or implicit link) or the
     context string "@Local" (as an explicit link only). (It
     can also be reached by sequentially browsing the file).
```

Both **.context** statements must appear together, immediately before the topic text they are to be associated with.

To create a context that begins with a literal @, precede it with a backslash (\ ).

Formatting Topic Text

You can use formatting attributes to control the appearance of the text on the screen. Using these attributes, you can make words appear in various colors, inverse video, and so forth, depending on the application and the capabilities of your display. This is useful, for example, to distinguish explicit links in the text.

Each formatting attribute consists of a backslash (\ ) followed by a character. Table 18.1 lists the formatting attributes.

Table 18.1 Formatting Attributes

Formatting Attribute	Action
\a	Anchors text for explicit links
\b, \B	Turns bold 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 explicit links)
\\	Inserts a single backslash in text

On color monitors, text labeled with the bold, italic, and underline attributes is translated by the application into suitable colors, depending on the user's default color selections. On monochrome monitors, the text's appearance depends on the application.

The \b, \i, \u, and \v options are toggles; they turn their respective attributes on or off. You can use several of these on the same text. Use the \p attribute to turn off all attributes except \v. Use the \v attribute to hide explicit links in the text. Explicit links are discussed on page 602.

Only visible characters count toward the character-width limit specified with the /W command-line option. Lines that begin with an application-specific control character are truncated to 255 characters regardless of the width specification. For more information on truncation and application-specific control characters, see "Options for Encoding" on page 596.

In the following example, \b initiates bold text for `Example 1`, and \p changes the remaining text to plain text:

```
\bExample 1\p  This is a bold head for the first example.
```

Dot and Colon Commands

Dot commands identify topics and convey other topic-related information to the Help reader.

The most important dot command is the **.context** command, described in "Defining a Topic" on page 600. Every topic begins with one or more **.context** commands. Each **.context** command defines a context string for the topic. You can define more than one context for a single topic, as long as you do not place any topic text between the context definitions.

Most dot commands have an equivalent colon command, which consists of a colon (**:**) followed by a character. If you decode a database without using /T, commands in the database are shown as colon commands. You can use both colon commands and dot commands in the same source file. If you use any dot commands other than **.context** or **.comment**, you must supply the /T option when encoding.

Table 18.2 lists the dot and colon commands. Some commands are not supported by all Help readers.

Table 18.2 Dot and Colon Commands

Dot Command	Colon Command	Action
.category *string*	:c	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 list of topics. Supported only by QuickHelp.
.command	:x	Indicates that the topic cannot be displayed. Use this command to hide command topics and other internal information.
.comment *string* **..** *string*	none	The *string* is a comment that appears only in the source file. Comments are not inserted in the database and are not restored during decoding.
.context *string*	none	The *string* defines a context.
.end	:e	Ends a paste section. See the **.paste** command. Supported only by QuickHelp.
.execute	:y	Executes the specified command. For example, `.execute P`*mark context* represents a jump to the specified context at the specified mark. See the **.mark** command.

.freeze *numlines*	:z	Locks the first *numlines* lines at the top of the screen. These lines do not move when the text is scrolled.
.length *topiclength*	:l	Sets the default window size for the topic in *topiclength* lines.
.line *number*	none	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. See **.source**. The **.line** command is not inserted in the Help database and is not restored during decoding.

Table 18.3 Dot and Colon Commands (*continued*)

Dot Command	Colon Command	Action
.list	:i	Indicates that the current topic contains a list of topics. Help displays a highlighted line; you can choose a topic by moving the highlighted line over the desired topic and pressing ENTER. If the line contains a coded link, Help looks up that link. If it does not contain a link, Help looks within the line for a string terminated by two spaces or a newline character and looks up that string. Otherwise, Help looks up the first word.
.mark *name* [[*column*]]	*:m*	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.
.next *context*	:>	Tells the Help reader 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*	:p	Begins a paste section. The *pastename* appears in the QuickHelp Paste menu. Supported only by QuickHelp.
.popup	:g	Tells the Help reader to display the current topic as a popup window instead of as a normal, scrollable topic. Supported only by QuickHelp.
.previous *context*	:<	Tells the Help reader 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	:u	Turns off special processing of certain characters by the Help reader.
.ref *topic*[[**,** *topic*]]	:r	Tells the Help reader to display the *topic* in the Reference menu. You can list multiple topics; separate each additional *topic* with a comma. A **.ref** command is not affected by the /W option. 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 word on the line. Supported only by QuickHelp.

Table 18.4 Dot and Colon Commands (*continued*)

Dot Command	Colon Command	Action
.source *filename*	(none)	Tells HELPMAKE that subsequent topics come from *filename*. HELPMAKE error messages contain the name and line number of the input file. The **.source** command tells HELPMAKE to use *filename* in the 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. See **.line**. The **.source** command is not inserted in the Help database and is not restored during decoding.
.topic *text*	:n	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.

Example

The following example is in QuickHelp format:

```
.context Sample
.context @Sample
.topic Sample Help Topic
.length 20
.freeze 3
                              \i\p\aBack\v!B\v\i\p
-----------------------------------------------------

  Help can contain text with three attributes:

  \bAttribute\p    \bQuickHelp Code\p

  \iItalic\p       \\i
  \bBold\p         \\b
  \uUnderline\p    \\u

  The visual appearance of each attribute
  or combination of attributes is determined
  by the application that displays the help.

  \bSee:\p
```

```
Coding, Expressions, Grammar, Keywords, Syntax
\i\p\aFlow Control\v@flow\v\i\p
\i\p\aRelease Notes\v$DOC:README.DOC!\v\i\p
.context @flow
.topic Sample Help: Flow Control
.length 8
.freeze 3
                              \i\p\aBack\v!B\v\i\p
---------------------------------------------------

  Here's another sample help screen.

  This is an explicit link: \i\p\aSample\v@Sample\v\i\p
  This is an implicit link: Sample
```

Other Help Text Formats

There are two other Help text formats you can use to create a Help database: rich text format (RTF) and minimally formatted ASCII. These formats are described in the next two sections.

Rich Text Format

Rich text format (RTF) is a Microsoft word-processing format supported by several word processors, including Microsoft Word version 5.0 and later and Microsoft Word for Windows. RTF is an intermediate format that allows documents to be transferred between applications without loss of formatting. You can use RTF to simplify the transfer of help files from one format to another. Like QuickHelp files, RTF files can contain formatting attributes and links.

As with the other text formats, each topic in an RTF source file consists of one or more context strings followed by topic text. 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. You can define any number of contexts for one topic. The topic text comprises all subsequent paragraphs up to the next paragraph that begins with the Help delimiter.

All QuickHelp dot commands, except **.context** and **.length**, can be used in RTF files. Each command must appear in a separate paragraph.

There are two ways to create an RTF file. The easiest way is to use a RTF word processor. RTF files usually contain additional information that is not visible to the user; HELPMAKE ignores this extra information.

You can also use an ordinary text editor to insert RTF codes manually. Utility programs exist that convert text files in other formats to RTF format. For more information on converting to and from RTF, see the Microsoft Word for Windows *User's Guide*.

Using a Word Processor

In an RTF-compatible word processor, enter the text and format it as you want it to appear: bold, underlined, hidden, and italic. You can also format paragraphs by selecting body and first-line indenting. Choose a monospace font and set the margin to the /W value you plan to encode the database with. The only item you need to insert into an RTF file manually is the Help delimiter (>>) followed by the context string that starts each entry. If you use dot commands, place each in its own paragraph.

When you have entered and formatted the text, save it in RTF format. In Microsoft Word version 5.5, for example, choose Save As from the File menu, then select RTF under Format.

You cannot see the RTF formatting codes when you load an RTF file into a compatible word processor. The word processor displays the text with the specified attributes. However, you can view these codes by loading an RTF file into a text editor or word processor.

Manually Inserting RTF Formatting Codes

RTF uses curly braces (**{ }**) for nesting. Thus, the entire file is enclosed in curly braces, as is each specially formatted text item.

When you manually insert RTF codes, you must delimit each dot command with the \par code. (An RTF editor or word processor inserts " \par" at the beginning and end of a paragraph.) For example, to use the **.popup** command, write:

```
\par.popup\par
```

HELPMAKE recognizes the subset of RTF codes listed in Table 18.3.

Table 18.5 RTF Formatting Codes

RTF code	Action
\b	Bold. The Help reader decides how to display this; often it is intensified text.
\fi*n*	Paragraph first-line indent, *n* twips.*
\i	Italic. The application decides how to display this; often it is reverse video.
\li*n*	Paragraph indent from left margin, *n* twips.*
\line	New line (not new paragraph).
\par	End of paragraph.

Table 18.6 RTF Formatting Codes (*continued*)

RTF code	Action
\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 attribute; some adapters that do not support underlining display it as blue text.
\v	Hidden text. Hidden text is used for explicit links; it is not displayed.

* A "twip" is 1/20 of a point or 1/1440 of an inch. One space is approximately 180 twips.

Encoding RTF with HELPMAKE

When HELPMAKE compresses an RTF file, it formats the text to the width given by the /W option and ignores the paragraph formats.

When HELPMAKE encodes RTF, any text between an RTF code and invisible text becomes an explicit link. This is illustrated in the following example:

```
{\b Formatting table}{\v prinf.ex}
```

The string `Formatting table` is displayed in bold and is part of an explicit link to `printf.ex`.

Example

The following example is in RTF format:

```
{\rtf1
\pard\plain >>Sample
\par >@Sample
\par .topic Sample Help Topic
\par .freeze 3
\par \pard \li8000 {\i }{\b Back}{\v !B}{\i }
\par \pard ----------------------------------------------------
\par
\par \pard \li360 Help can contain text with three attributes:
\par \pard
\par \pard \li360 {\b Attribute}    {\b QuickHelp Code}
\par \pard
\par \pard \li360 {\i Italic}       \\i
\par \pard \li360 {\b Bold}         \\b
\par \pard \li360 {\ul Underline}    \\u
\par \pard
\par \pard \li360\ri720 The visual appearance of each attribute
or combination of attributes is determined
by the application that displays the help.
```

```
\par \pard
\par \pard \li360 {\b See:}
\par \pard
\par \pard \li360 Coding, Expressions, Grammar, Keywords, Syntax
\par \pard \li360 {\i }{\b Flow Control}{\v @flow}{\i }
\par {\i }{\b Release Notes}{\v $DOC:README.DOC!}{\i }
\par \pard >@flow
\par .topic Sample Help: Flow Control
\par .freeze 3
\par \pard \li8000 {\i }{\b Back}{\v !B}{\i }
\par \pard ----------------------------------------------------
\par
\par \pard \li360 Here's another sample help screen.
\par
\par \pard \li360 This is an explicit link: {\i }{\b Sample}{\v
@Sample}{\i }
\par \pard \li360 This is an implicit link: Sample
\par
}
```

Minimally Formatted ASCII

Minimally formatted ASCII files define contexts and their topic text. The Help information is displayed exactly as it appears in the file. A minimally formatted ASCII file cannot contain screen-formatting commands or explicit links. Any formatting codes are treated as ASCII text. Minimally formatted ASCII files have a fixed width.

A minimally formatted ASCII file contains a sequence of topics, each preceded by one or more context definitions. Each context definition must be on a separate line that begins with a help delimiter (>>). The topic consists of all subsequent lines up to the next context definition.

Implicit links work the same way they do in the other formats. A word in the Help text is an implicit link if it exists as a context somewhere in the Help system.

There are two ways to use a minimally formatted ASCII file. You can compress it with HELPMAKE and create a Help database, or a Help reader can access the uncompressed file directly. A Help reader can search a minimally formatted ASCII file faster if it has been compressed.

The following example coded in minimally formatted ASCII shows the same sample help topic as the QuickHelp and RTF examples presented elsewhere in this chapter:

```
>>Sample
---------------------[ Sample Help Topic ]-------------------
  Help can contain text with three attributes:

  Attribute    QuickHelp Code
  Italic       \i
  Bold         \b
  Underline    \u

  The visual appearance of each attribute
  or combination of attributes is determined
  by the application that displays the help.

  See:

  Coding, Expressions, Grammar, Keywords, Syntax

>>Coding
--------------------[ Sample Help: Coding ]------------------

  Here's another sample help screen.
```

The last three lines of this example differ from the ending lines in the other two examples because minimally formatted ASCII files cannot contain explicit links, and the help information is displayed exactly as it appears in the file.

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 and has a predefined meaning. You may see these contexts in the resulting text file when you decode a Microsoft help database.

The context prefixes shown in Table 18.4 are used by Microsoft to mark product-specific features. They appear in decompressed databases. However, you do not need to add them to the files you write.

Table 18.7 Microsoft Product Context Prefixes

Prefix	Purpose
d.	Dialog box. The context string for the Help on a dialog box is `d.` followed by the number assigned to that dialog box.
e.	Error number. If a product supports the error numbering used by Microsoft languages, it displays Help for each error using this prefix.

Table 18.8 Microsoft Product Context Prefixes (*continued*)

Prefix	Purpose
h.	Help item. The context string for miscellaneous Help is `h.` followed by an assigned string. These strings are described in Table 18.5. For example, most Help readers look for the context string `h.contents` when Contents is chosen from the Help menu.
m.	Menu item. The strings that can follow `h.` are defined by the access keys for the product's menu items. For example, the Exit command on the File menu is accessed by ALT+F, X. The context string for Help on the command is `m.f.x`.
n.	Message number. The context string for the Help on a message box is `n.` followed by the number assigned to that message box.

You can use the h. prefix, shown in Table 18.5, to identify standard Help-file contexts. For instance, **h.default** identifies the default Help screen (the screen that usually appears when you select top-level Help).

Table 18.9 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 to get 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.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.

CHAPTER 19

Browser Utilities

This chapter describes three utilities:

- Microsoft Browser Database Maintenance Utility (BSCMAKE) version 2.00
- Microsoft Browse Information Compactor (SBRPACK) version 2.00
- Microsoft Cross-Reference Utility (CREF) version 6.00

These utilities build a browser database for use with the Microsoft Source Browser, a feature of the Microsoft Programmer's Workbench (PWB). As a navigation tool, the browser gives you the means to move around quickly in a large project and find pieces of code in your source and include files. As an interactive program database, the browser can answer questions about where functions are invoked or where variables and types are used. The browser can also generate useful outlines, call trees, and cross-reference tables.

When you tell PWB to create a browser database (.BSC file) for the program you are building, PWB automatically calls BSCMAKE. You do not need to know how to run BSCMAKE to create your database in PWB. However, you may want to read this chapter to understand the PWB options available to modify the database. For information on how to create and use a browser database in PWB, see "Using the Source Browser" in Chapter 5.

If you build your program outside of PWB, you can still create a custom browser database that you can examine with the Browser in PWB. Run the BSCMAKE utility to build the database from .SBR files created during compilation. You might need to run SBRPACK to provide more efficiency during the build. This chapter describes how to use both these utilities to create your browser database. For further information, see "Building Databases for Non-PWB Projects" on page 94.

Note BSCMAKE is the successor to the Microsoft PWBRMAKE Utility. To

allow existing makefiles to remain compatible, a file called PWBRMAKE.EXE is provided with BSCMAKE. This version of PWBRMAKE calls BSCMAKE using the arguments and options specified on the PWBRMAKE command line.

Overview of Database Building

BSCMAKE can build a new database from newly created .SBR files. It can also maintain an existing database using .SBR files for object files that have changed since the last build. The following sections describe how .SBR files are created, what you need to know to build a database, and how you can make the database-building process more efficient.

Preparing to Build a Database

The input files for BSCMAKE are .SBR files that you create when you compile or assemble your source files. When you build or update your browser database, all .SBR files for your project must be available on disk. To create an .SBR file, specify the appropriate command-line option to the compiler or assembler (shown in parentheses below). The following products generate .SBR files:

- Microsoft MASM versions 6.0 and later (/FR or /Fr)
- Microsoft C Compiler versions 6.0 and later (/FR or /Fr)
- Microsoft FORTRAN versions 5.1 and later (/FR or /Fr)
- Microsoft Basic versions 7.1 and later (/FBr or /FBx)
- Microsoft COBOL versions 4.0 and later (BROWSE)

The above options /FR, /FBx, and BROWSE put all possible information into the .SBR file. The options /Fr and /FBr omit local symbols from the .SBR file. (If the .SBR file was created with all possible information, you can still omit local symbols by using BSCMAKE's /El option; see page 620.)

Database building can be more efficient if the .SBR files are first packed by SBRPACK. The Microsoft C Compiler (CL) versions 7.0 and later automatically calls SBRPACK when it creates an .SBR file. (If you want to prevent packing, specify CL's /Zn option.) Other Microsoft language products do not call SBRPACK. Before you run BSCMAKE, you may want to run SBRPACK on any .SBR files that were not previously packed. See "SBRPACK" on page 623.

You can create an .SBR file without performing a full compile. For example, you can specify the ML or CL /Zs option to perform a syntax check and still generate an .SBR file if you specify /FR or /Fr.

How BSCMAKE Builds a Database

BSCMAKE builds or rebuilds a database in the most efficient way it can. To avoid some potential problems, it is important to understand the database-building process.

When BSCMAKE builds a database, it truncates the .SBR files to zero length. During a subsequent build of the same database, a zero-length (or empty) .SBR file tells BSCMAKE that the .SBR file has no new contribution to make. It lets BSCMAKE know that an update of that part of the database is not required and an incremental build will be sufficient. During every build (unless the /n option is specified), BSCMAKE first attempts to update the database incrementally by using only those .SBR files that have changed.

BSCMAKE looks for a .BSC file that has the name specified with the /o option; if /o is not specified, BSCMAKE looks for a file that has the base name of the first .SBR file and a .BSC extension. If the database exists, BSCMAKE performs an incremental build of the database using only the contributing .SBR files. If the database does not exist, BSCMAKE performs a full build using all .SBR files.

Requirements for a Full Build

For a full build to succeed, all specified .SBR files must exist and must not be truncated. If any .SBR file is truncated, you must first rebuild it (by recompiling or assembling) before running BSCMAKE.

Requirements for an Incremental Build

For an incremental build to succeed, the .BSC file must exist. All contributing .SBR files, even empty files, must exist and must be specified on the BSCMAKE command line. If you omit an .SBR file from the command line, BSCMAKE removes its contribution from the database.

Methods for Increasing Efficiency

The database-building process can require large amounts of time, memory, and disk space. However, there are several ways to reduce these requirements.

Managing Memory Under MS-DOS

Building a database uses a lot of memory. Large projects benefit the most from use of the Source Browser, but under MS-DOS their large size can cause BSCMAKE to run out of memory. There are several ways to run BSCMAKE under MS-DOS that make use of virtual memory and extended memory. The commands to run these forms of BSCMAKE are described in “System Requirements for BSCMAKE” on page 618.

Making a Smaller Database

Smaller databases take less time to build, use up less space on disk, have a lower risk of causing BSCMAKE to run out of memory, and run faster in the browser. The following list gives some methods of making a smaller database:

- Use BSCMAKE options to exclude information from the database.
- Omit local symbols in one or more .SBR files when compiling or assembling.
- If an object file does not contain information that you need for your current stage of debugging, omit its .SBR file when rebuilding the database.

Saving Build Time and Disk Space

Unreferenced definitions cause .SBR files to take up more disk space and cause BSCMAKE to run less efficiently. The SBRPACK utility removes unreferenced definitions from .SBR files. For more information, see “SBRPACK” on page 623.

Using C/C++ Precompiled Headers

Using precompiled headers speeds up BSCMAKE because the browser information for the precompiled code is generated only when the .PCH file is created. The browser information is not replicated in each source file’s browser file, as it is when you do not use precompiled headers. This reduces the size of .SBR files for object files that use precompiled headers and makes database building faster. Also, less disk space is used.

BSCMAKE

The Microsoft Browser Database Maintenance Utility (BSCMAKE) converts .SBR files created by a compiler or assembler into database files that can be read by the PWB Source Browser. The filename of the resulting browser database has the extension .BSC. For more information on the browser, see “Using the Source Browser” in Chapter 5.

System Requirements for BSCMAKE

BSCMAKE version 2.00 exists as two executable files. The form of BSCMAKE that you run is determined by your computer’s memory. The following executable files are discussed in this section:

- BSCMAKE.EXE for MS-DOS; can use only extended memory
- BSCMAKEV.EXE for MS-DOS; can use virtual and extended memory

BSCMAKE can use either virtual memory or extended memory (or both) to avoid running out of memory. BSCMAKE.EXE uses extended memory if

available. If extended memory is unavailable, BSCMAKE runs under MS-DOS in real mode. The command to invoke this version of BSCMAKE.EXE is:

```
BSCMAKE
```

followed by the rest of the command line. For best results, the sum of available conventional and extended memory should be half the size of the disk space occupied by the finished database.

If your computer does not have extended memory or if it is insufficient for your database, you can use virtual memory. BSCMAKEV.EXE uses extended memory if it is available. If extended memory is unavailable or insufficient, BSCMAKEV uses virtual memory, copying information to your disk as needed during the database build. Swapping to disk is slower but can overcome a shortage of memory. The command to invoke this form of BSCMAKE is:

```
BSCMAKEV
```

followed by the rest of the command line.

To prevent BSCMAKE or BSCMAKEV from using extended memory, specify the /r option as the first option on the command line.

The BSCMAKE Command Line

To run BSCMAKE, use the following command line:

BSCMAKE [[*options*]] *sbrfiles*

This syntax applies to all forms of BSCMAKE. Specify either BSCMAKE or BSCMAKEV in the first position on the command line.

Options can appear only in the *options* field on the command line. If the /r option is used, it must be first.

The *sbrfiles* field specifies one or more .SBR files created by a compiler or assembler. If you specify more than one file, separate the names with spaces or tabs. You must specify the extension; there is no default. You can specify a path with the filename, and you can use operating-system wildcards (***** and **?**).

During an incremental build, you can specify new .SBR files that were not part of the original build. If you want all contributions to remain in the database, you must specify all .SBR files (including truncated files) that were originally used to create the database. If you omit an .SBR file, that file's contribution to the database is removed.

Do not specify a truncated .SBR file for a full build. A full build requires contributions from all specified .SBR files. Before you perform a full build, recompile and create a new .SBR file for each empty file.

Example

The following command runs BSCMAKE to build a file called MAIN.BSC from three .SBR files:

```
BSCMAKE main.sbr file1.sbr file2.sbr
```

BSCMAKE Options

This section describes the options available for controlling BSCMAKE. Several options control the content of the database by telling BSCMAKE to exclude or include certain information. The exclusion options can allow BSCMAKE to run faster and may result in a smaller .BSC file. Option names are case sensitive (except for /HELP and /NOLOGO).

/Ei *filename*
/Ei (*filename*...)
: Excludes the contents of the specified include files from the database. To specify multiple files, separate the names with spaces and enclose the list in parentheses. Use /Ei along with the /Es option to exclude files not excluded by /Es.

/El
: Excludes local symbols. The default is to include local symbols in the database. For more information about local symbols, see "Preparing to Build a Database" on page 616.

/Em
: Excludes symbols in the body of macros. Use /Em to include only the names of macros in the database. The default is to include both the macro names and the result of the macro expansions.

/Er *symbol*
/Er (*symbol*...)
: Excludes the specified symbols from the database. To specify multiple symbol names, separate the names with spaces and enclose the list in parentheses.

/Es
: Excludes from the database every include file specified with an absolute path or found in an absolute path specified in the INCLUDE environment variable. (Usually, these are the system include files, which contain a lot of information that you may not need in your database.) This option does not exclude files specified without a path or with relative paths or found in a relative path in INCLUDE. You can use the /Ei option along with /Es to exclude files that /Es does not exclude. If you want to exclude only some of the files that /Es excludes, use /Ei instead of /Es and list the files you want to exclude.

/HELP
Calls the QuickHelp utility. If BSCMAKE cannot find the Help file or QuickHelp, it displays a brief summary of BSCMAKE command-line syntax.

/Iu
Includes unreferenced symbols. By default, BSCMAKE does not record any symbols that are defined but not referenced. If an .SBR file has been processed by SBRPACK, this option has no effect for that input file because SBRPACK has already removed the unreferenced symbols.

/n
Forces a nonincremental build. Use /n to force a full build of the database whether or not a .BSC file exists and to prevent .SBR files from being truncated. See "Requirements for a Full Build" on page 617.

/NOLOGO
Suppresses the BSCMAKE copyright message.

/o *filename*
Specifies a name for the database file. By default, BSCMAKE assumes that the database file has the base name of the first .SBR file and a .BSC extension.

/r
Prevents BSCMAKE from using extended memory under MS-DOS. The /r option must appear first in the options field on the command line and cannot appear in a response file. BSCMAKE.EXE and BSCMAKEV.EXE are extender-ready and use extended memory if it exists. This option forces BSCMAKE to use only conventional memory and forces BSCMAKEV to use conventional memory and virtual memory. For more information, see "System Requirements for BSCMAKE" on page 618.

/S *filename*
/S (*filename*...)
Tells BSCMAKE to process the specified include file the first time it is encountered and to exclude it otherwise. Use this option to save processing time when a file (such as a header, or .H, file for a .C source file) is included in several source files but is unchanged by preprocessing directives each time. You may also want to use this option if a file is changed in ways that are unimportant for the database you are creating. To specify multiple files, separate the names with spaces and enclose the list in parentheses. If you want to exclude the file every time it is included, use the /Ei or /Es option.

/v
Provides verbose output, which includes the name of each .SBR file being processed and information about the complete BSCMAKE run.

/?
Displays a brief summary of BSCMAKE command-line syntax.

Example

The following command line tells BSCMAKE to use virtual memory and conventional memory (but not extended memory) to do a full build of MAIN.BSC from three .SBR files. It also tells BSCMAKE to exclude duplicate instances of TOOLBOX.H:

```
BSCMAKEV /r /n /S toolbox.h /o main.bsc file1.sbr file2.sbr file3.sbr
```

Using a Response File

You can provide part or all of the command-line input in a response file, which is a text file that contains options and/or filenames.

Specify the response file using the following syntax:

BSCMAKE **@***responsefile*

This syntax applies to all forms of BSCMAKE; you can specify BSCMAKE or BSCMAKEV in the first position on the command line. Only one response file is allowed. You can specify a path with *responsefile*. Precede the filename with an at sign (**@**). BSCMAKE does not assume an extension. You can specify additional *sbrfiles* on the command line after *responsefile*. If you use /r, you must specify it on the command line before the response file.

In the response file, specify the input to BSCMAKE in the same order as you would on the command line. Separate the command-line arguments with one or more spaces, tabs, or newline characters.

Example

The following command calls BSCMAKE using the prog1.txt response file:

```
BSCMAKE @prog1.txt
```

Example

The following is a listing of prog1.txt:

```
/n /v /o main.bsc /El
/S (
toolbox.h
verdate.h c:\src\inc\screen.h
)
/Er ( HWND HpOfSbIb
LONG LPSTR
NEAR NULL
PASCAL
VOID
WORD
)
file1.sbr file2.sbr file3.sbr file4.sbr
```

The prog1.txt response file instructs BSCMAKE to:

- Perform a nonincremental build (/n, /o) of MAIN.BSC from the FILE1, FILE2, FILE3, and FILE4 *.SBR files.
- Report full information about the complete BCSMAKE run (/v).
- Exclude local symbols (/El), and process the include files TOOLBOX.H, VERDATE.H, and SCREEN.H files only once (/S).
- Exclude the HWND, HpOfSbIb, LONG, LPSTR, NEAR, NULL, PASCAL, VOID, and WORD symbols (/Er).

BSCMAKE Exit Codes

BSCMAKE returns an exit code (also called return code or error code) to the operating system or the calling program. You can use the exit code to control the operation of batch files or makefiles.

Code	Meaning
0	No error
1	Command-line error
4	Fatal error during database build

SBRPACK

The Microsoft Browse Information Compactor (SBRPACK) removes unreferenced symbols from .SBR files before they are processed by BSCMAKE. This can result in smaller .SBR files, which allow BSCMAKE to run faster. Smaller .SBR files also save space on disk.

Packing .SBR files is optional. The Microsoft C Compiler versions 7.0 (CL) and later automatically call SBRPACK when you specify either /FR or /Fr to create an .SBR file. If you specify /Zn in addition to one of these options, CL does not call SBRPACK to pack the .SBR file. Other compilers and assemblers do not pack .SBR files. You may want to use SBRPACK to pack an .SBR file that was created without packing.

SBRPACK.EXE version 2.00 runs under real-mode MS-DOS. It does not use virtual memory, expanded memory, or extended memory.

Overview of SBRPACK

When symbols such as functions or data are defined but not referenced, you can use SBRPACK to remove them from the .SBR files before the files are processed by BSCMAKE. A common source of unreferenced symbols is an include, or header, file. When a source file includes a header file, it often brings in a large number of unreferenced definitions. Therefore, the .SBR file that results from compiling this source file can contain a large amount of unneeded information. The time or disk space saved by SBRPACK is directly related to the number of unreferenced symbols in the .SBR files.

If SBRPACK is not used, BSCMAKE will remove the same information (unless you specify BSCMAKE's /Iu option to preserve this information). However, BSCMAKE can run more efficiently if the .SBR files are first processed by SBRPACK. The time it takes to run both utilities can be less than if BSCMAKE is used alone, especially under real-mode MS-DOS or under extended MS-DOS using virtual memory.

You can run SBRPACK every time you create an .SBR file, or you can run it just once before running BSCMAKE. If you need to save room on your disk, run SBRPACK after every compilation. The .SBR files will then be stored in a more compact form. If you need to accelerate your program-build process, run SBRPACK only as needed, just before running BSCMAKE. The example in the following section shows how to run SBRPACK to perform each kind of efficiency.

The SBRPACK Command Line

To run SBRPACK, use the following command line:

SBRPACK [[*option*]] *sbrfiles*

Option names are not case sensitive. Only the /NOLOGO option applies to a packing session; the other options provide help and then halt SBRPACK.

The *sbrfiles* field specifies one or more .SBR files created by a compiler or assembler. If you specify more than one file, separate the names with spaces or

tabs. You must specify the extension; there is no default. You can specify a path with the filename, and you can use operating-system wildcards (***** and **?**).

You do not specify a name for the resulting files; SBRPACK saves the changed files under their original name. If you want to preserve the original files, copy them to another name before running SBRPACK.

SBRPACK has the following options:

/HELP
: Calls the QuickHelp utility. If SBRPACK cannot find the Help file or QuickHelp, it displays a brief summary of SBRPACK command-line syntax.

/NOLOGO
: Suppresses the SBRPACK copyright message.

/?
: Displays a brief summary of SBRPACK command-line syntax.

Example

The following commands assemble a file using the Microsoft Macro Assembler (ML), compile a file using the Microsoft C Optimizing Compiler (CL), and build a browser database using both SBRPACK and BSCMAKE:

```
ML /FR /c prog1.asm
CL /FR /c prog2.c
SBRPACK prog2.sbr
.
.
.
BSCMAKE *.sbr
```

These commands run SBRPACK every time an .SBR file is created. A separate SBRPACK command isn't needed for PROG2.SBR because CL calls SBRPACK automatically. Later in the program-building session, BSCMAKE builds a database and names it PROG1.BSC. This combination of commands saves space on disk during the program-building session.

The same commands can be configured to create the same database but save running time. In the following example, SBRPACK is called only when BSCMAKE is about to run. If these commands are in a makefile, time is saved if the program-building sequence stops before a database is built.

```
ML /FR /c prog1.asm
CL /FR /Zn /c prog2.c
.
.
.
SBRPACK *.sbr
BSCMAKE *.sbr
```

SBRPACK Exit Codes

SBRPACK returns an exit code (also called return code or error code) to the operating system or the calling program. You can use the exit code to control the operation of batch files or makefiles.

Code	Meaning
0	No error
1-4	Fatal SBRPACK error

Each fatal error generates a specific exit code. For individual exit codes associated with each error, see Appendix A.

CREF

This section contains information on the purpose and use of the Microsoft Cross-Reference Utility (CREF) Version 6.00.

The Microsoft Cross-Reference Utility (CREF) creates a cross-reference listing of all symbols in an assembly-language program. A cross-reference listing is an alphabetical list of symbols followed by references to where a symbol appears in the source code.

CREF is intended for use as a debugging aid to speed up the search for symbols encountered during a debugging session. The cross-reference listing, together with the symbol table created by the assembler, can make debugging and correcting a program easier.

Using CREF

CREF creates a cross-reference listing for a program by converting PWB Source Browser database files (those ending with a .BSC or an .SBR extension) into a readable ASCII file.

Command-Line Syntax

CREF *crossreferencefile* [[, *crossreferencelisting*]] [[;]]

crossreferencefile
Source Browser database file. Default filename extension is .BSC.

crossreferencelisting
Readable ASCII cross-reference listing. Default filename extension is REF. If this filename is not supplied on the command line, the base filename of the database file is used and the REF extension is added.

Using Prompts to Create a Cross-reference Listing

You can direct CREF to prompt you for the files it needs by starting CREF with just the command name (CREF). CREF prompts you for the input it needs by displaying the following lines, one at a time:

```
Cross-Reference [.BSC]:
Listing [filename.REF]:
```

The prompts correspond to the fields of the CREF command lines. CREF waits for you to respond to each prompt before printing the next one. You must type a cross-reference filename (extension is optional) at the first prompt. For the second prompt, you can either type a filename or press the ENTER key to accept the default displayed in brackets after the prompt.

Use of BSCMAKE with CREF

Note that the CREF utility can create reports only from the final form of the Source Browser database files. These files end in .BSC (Browser Source Cache), and are created by the utility BSCMAKE. Because ML creates an intermediate Source Browser file (with the filename extension .SBR), CREF invokes BSCMAKE to create a .BSC file from a single .SBR file input. BSCMAKE takes as input the .SBR files created by ML, merges them with other .SBR files created from other assembly-language source modules (if any exist), and creates a .BSC file as the result. The .BSC file contains all of the information (across multiple modules, if necessary) to provide a comprehensive cross-listing file for an entire project.

CREF will invoke BSCMAKE only when a single .SBR file is given as input to the utility; therefore, you must create a .BSC database file from the assembler-generated .SBR files before invoking CREF. An example of running BSCMAKE is:

```
BSCMAKE FILENAME.SBR FILE2.SBR FILE3.SBR ... FILEN.SBR
```

This example creates a .BSC file using the base name of the first .SBR file on the command line. In this case, the resultant file would be named FILENAME.BSC.

By using the .BSC Source Browser database files, CREF is able to provide cross-reference files for either a single assembly-language module or for an

entire multimodule assembly-language project. Below are the steps necessary for both scenarios.

For more information on BSCMAKE options, see page 620, or type BSCMAKE /? at the command line to get a quick-reference screen.

Creating Single-Module Cross-reference Listings

Using ML with the /FR switch, create an .SBR file for the assembly-language module. An example is:

```
ML /c /FR filename.ASM
```

This creates a file FILENAME.SBR (as well as FILENAME.OBJ).

Invoke CREF using the database file created above (FILENAME.SBR) as the input cross-reference file. An example of this is:

```
CREF FILENAME.SBR, FILENAME.REF
```

This creates the cross-reference listing FILENAME.REF.

Note that because CREF is capable only of reading Source Browser .BSC database files, the BSCMAKE utility is automatically invoked to convert the .SBR on the command line into a .BSC file.

Creating Multimodule Cross-reference Listings

Using ML with the /FR switch, create an .SBR file for each of the assembly-language modules. An example is:

```
ML /c /FR *.ASM
```

This creates a file FILENAME.SBR (as well as FILENAME.OBJ).

Invoke CREF using the database file created above (FILENAME.BSC) as the input cross-reference file. An example of this is:

```
CREF FILENAME.BSC, FILENAME.REF
```

The cross-reference listing FILENAME.REF is created. This particular example will have all of the symbols from all of the modules in one cross-reference listing file.

Reading Cross-reference Listings

The cross-reference listing contains the name of each symbol defined in your program. Each name is followed by a list of source filenames in which the symbol appears. Each source name is then followed by a list of line numbers representing the line or lines in the source file in which a symbol is referenced.

Line numbers in which a symbol is defined are marked with the number sign (#).

Below is a sample assembly-language source-code module, followed by the resulting cross-reference listing file.

Example

```
.DOSSEG .MODEL small, pascal
.STACK 100h
.DATA PUBLIC message, lmessage
message BYTE "Hello World!!!!" lmessage EQU $-message
.CODE
.STARTUP
EXTERN display:NEAR
call display
mov ax, 4C00h
int 21h
.EXIT END

Example Reference Listing (Created from above source-code module) ------
-------------------------------------------------------------
Microsoft Cross-Reference Version 6.00 Wed Nov 18 15:47:26 1992
Symbol Cross-Reference (# definition) Cref-1
@code ..\hello.asm . . . . . 2
# @CodeSize ..\hello.asm . . . . . 2
# @data ..\hello.asm . . . . . 2
# @DataSize ..\hello.asm . . . . . 2
# @fardata ..\hello.asm . . . . . 2
# @fardata? ..\hello.asm . . . . . 2
# @Interface ..\hello.asm . . . . . 2
# @Model ..\hello.asm . . . . . 2
# @stack ..\hello.asm . . . . . 2
# @Startup ..\hello.asm . . . . . 14
# _DATA ..\hello.asm . . . . . 2
# _TEXT ..\hello.asm . . . . . 2# 12
DGROUP ..\hello.asm . . . . . 2# 2 14
display ..\hello.asm . . . . . 16 17 <Unknown>. . . . . .
# hello.asm ..\hello.asm . . . . . 1
# lmessage ..\hello.asm . . . . . 10# 7
message ..\hello.asm . . . . . 9# 10 7
STACK ..\hello.asm . . . . . 4
```

Difference from Previous Releases

Use Differences

ML/MASM no longer generates a .CRF assembler-specific binary file. Instead, CREF 6.0 uses the generic Source Browser database file (.BSC).

Because of this, CREF can actually be used with any source-code module or project that is compiled/assembled by a Microsoft product that creates .SBR files.

Creating multimodule cross-reference listing files is another advantage of using .BSC Browser database files. Previous releases of CREF were limited to single modules.

Listing Differences

All line numbers that are reported are relative to the actual source-code module, not the assembler-generated listing file. This is a direct result of using the Source Browser database files (.BSC). These files are not directly related to MASM listings and are generic across Microsoft language products.

CHAPTER 20

Using Other Utilities

This chapter explains how to use the following utilities:

- CVPACK (Microsoft Debugging Information Compactor) version 4.05—Prepares executable files for use with the CodeView debugger by reducing the size of debugging information within the files.
- H2INC (Microsoft C Header Translation Utility) version 1.01—Translates C header files into MASM-compatible include files.
- IMPLIB (Microsoft Import Library Manager) version 1.40—Creates an import library for use in resolving external references from a Windows-based program to a dynamic-link library (DLL).
- RM (Microsoft File Removal Utility) version 2.00
  UNDEL (Microsoft File Undelete Utility) version 2.00
  EXP (Microsoft File Expunge Utility) version 2.00
- WX/WXServer Utility version 1.50—Runs a Windows-based program in an MS-DOS session.

CVPACK

This section describes the Microsoft Debugging Information Compactor (CVPACK) version 4.05. CVPACK 4.05 prepares an executable file for use with the Microsoft CodeView debugger version 4.05.

You should always use matching versions of CVPACK and CodeView. Earlier formats of debugging information and CVPACK-packing are not compatible with CodeView 4.05. If an executable file contains debugging information in an earlier format but has not been packed, packing with CVPACK 4.05 is all that is needed for the file to run in CodeView 4.05. However, if the executable file has been packed with an earlier version of CVPACK, you must relink the file.

Also, executable files packed using CVPACK 4.05 are not compatible with earlier versions of CodeView. The debugging information produced by Microsoft compilers and packed by CVPACK 4.05 is for use with CodeView 4.05 and is not compatible with earlier versions of CodeView.

Overview of CVPACK

An executable file to be run under CodeView 4.05 must first be packed by CVPACK 4.05. The debugging information in the file must be in the form given in the Microsoft Symbolic Debugging Information specification. This is the format supported by current Microsoft compilers and linkers.

LINK versions 5.30 and later automatically call CVPACK when you specify LINK's /CO option. You do not need to run CVPACK as a separate step. However, if you want to use CodeView to debug a file that was built by another linker (either an earlier Microsoft linker or a third-party linker), you must run CVPACK to convert the executable file to the current CodeView format. Be sure that the executable file has not been packed by an earlier version of CVPACK; if it has, you must relink the file.

CVPACK compresses debugging information by removing duplicate type definitions. To be removed by CVPACK, the definitions must be absolutely identical. For example, if a structure defined in two modules contains a pointer to another structure, but the second structure is defined in only one module, the pointer size is unknown in the other module. In this case, CVPACK cannot pack the duplicate structure definitions in the same way, which causes less efficient compression.

CVPACK can pack an executable that is in .COM format. The linker puts debugging information for a .COM file into a file with the same base name as the executable file and with a .DBG extension. When you specify a .COM file to be packed, CVPACK looks for a .DBG file with the same base name and in the same location as the .COM file.

The CVPACK Command Line

To run CVPACK, use the following command line:

CVPACK [[*option*]] *exefile*

The *exefile* specifies a single executable file to be packed. You can specify a path with the filename. If you do not specify an extension, CVPACK assumes the default extension .EXE.

CVPACK Options

CVPACK has the following options; the option names are not case sensitive:

/H[[ELP]]
: Calls the QuickHelp utility. If CVPACK cannot find the help file or QuickHelp, it displays a brief summary of CVPACK command-line syntax.

/M[[INIMUM]]
: Preserves only public symbols and line numbers. All other debugging information is removed from the executable file.

/N[[OLOGO]]
: Suppresses the CVPACK copyright message.

/?
: Displays a brief summary of CVPACK command-line syntax.

Note The /P option is not a valid option for the current version of CVPACK. Using this option causes an error.

Example

The following command packs the file PROJECT.EXE, located in the directory \TEST on the current drive:

```
CVPACK \TEST\PROJECT.EXE
```

CVPACK Exit Codes

CVPACK returns an exit code (also called return code or error code) to the operating system or the calling program. You can use the exit code to control the operation of batch files or makefiles.

Code	Meaning
0	No error
1	Program error caused by commands or files given as input to CVPACK

H2INC

This version (1.01) of H2INC now includes the following:

- The number of C keywords, preprocessor keywords and preprocessor pragmas has been expanded to maintain compatibility with Microsoft C/C++ Version 7.0.
- Invalid command-line options generate a warning, rather than a fatal error.

- OPTION CASEMAP: NONE is automatically added in the generated *.INC.
- A new warning (HI4010) is generated whenever an attempt is made to redefine a MASM keyword, or whenever a typedef statement converts to a type with the same name as the type.

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 that is compatible with Microsoft C versions 6.0 and later, and creates include files suitable for MASM versions 6.0 and later. These include files will not work with versions of MASM earlier than version 6.0.

H2INC is designed to translate project header files that you have written specifically for translation to MASM versions 6.0 and later include files. It is not designed to translate header files such as PM.H and WINDOWS.H.

This section explains how H2INC performs the C code translation and how the command-line options control the conversions.

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.

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.

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 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 list describes the options that indirectly affect the H2INC output:

Option	Action
/AC	Specifies compact memory model.
/AH	Specifies huge memory model.
/AL	Specifies large memory model.
/AM	Specifies medium memory model.
/AS	Specifies small memory model, the default.
/AT	Specifies tiny memory model (.COM).
/D[[*const*[[=*value*]]]]	Defines a constant or macro.
/G0	Enables 8086/8088 instructions (default).
/G1	Enables 80186/80188 instructions.
/G2	Enables 80286 instructions.

Option	Action
/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.
/Tc [[*filename*]]	Enables the processing of file whose name does not end in .H.
/u*ident*	"Undefines" one of the predefined identifiers.
/U	"Undefines" all predefined identifiers.
/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.

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.

The previous version of H2INC required you to specify **OPTION CASEMAP:NONE** in any MASM files that included .INC files generated with H2INC. H2INC now automatically specifies this option in the *.INC file it generates.

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. For more information on integer constants in MASM, see the *Programmer's Guide.*

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, nor does it translate type-cast conversions. For example, although the following C definitions are valid, H2INC creates invalid equates without generating an error.

These C statements

```
#define INT 6
#define FOREVER for(;;)
#define LONG_5  (long)  5
```

generate these MASM statements:

```
INT EQU 6t
FOREVER TEXTEQU <for(;;)>
LONG_5  TEXTEQU <(long) 5>
```

The first **#define** statement is invalid because **INT** is a MASM instruction; in MASM versions 6.0 and later, instructions are reserved and cannot be used as identifiers. Any attempt to redefine a MASM keyword will result in the warning:

```
HI4010: identifier: identifier is a MASM keyword.
```

The **for** loop definition is invalid because MASM cannot assemble C code.

The long type-cast conversion is invalid because a type cannot be assigned to a numerical equate. To resolve this in the above example, the C statement

```
#define LONG_5  (long)  5
```

could be changed to:

```
#define LONG_5 5.
```

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 610 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.

This also means that if a negative number is evaluated in an expression, its value can only be used as a double word (or longer) operand.

These C statements:

```
#define MINUS_1     (-1)
#define MINUS_2     -2
```

generate these MASM statements:

```
MINUS_1 EQU 0ffffffffh
MINUS_2 EQU -2t
```

In most cases, the second behavior is more desirable, as the decimal representation can be cast within MASM to the appropriate size (with a `word ptr` operator, for example).

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, wchar_t	**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 always assumes that a variable is external. For example, the C declaration

```
long big_data;
```

is converted to this MASM declaration:

```
EXTERNDEF big_data:SDWORD
```

For more information on MASM data types, see the *Programmer's Guide*.

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 variable values, since the scope of these variables extends only to the file where they are declared. Instead, H2INC substitutes EXTERNDEF declarations for all static variables. (This includes initialized static variables.)

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.

For information about MASM pointers, see the *Programmer's Guide*.

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. For information on alignment values, see the *Programmer's Guide*.

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
```

For information on MASM structures and unions, see the *Programmer’s Guide*.

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 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
```

For information about MASM records, see the *Programmer's Guide*.

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, then 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
```

For information on MASM integer constants, see the *Programmer's Guide*.

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 will generate the warning:

```
HI4010: identifier: identifier is a MASM keyword.
```

and 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
```

For information on using **TYPEDEF** in MASM 6.1, see the *Programmer's Guide.*

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.

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 4 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
```

For more information, see the *Programmer's Guide*.

Summary of H2INC-Recognized Keywords and Pragmas

The four lists below comprise a summary of all the special keywords recognized by H2INC. Items or lists marked with an asterisk (*) are recognized by H2INC version 1.01 or later.

C Keywords (Non-Scored)

auto	else	int	static
break	enum	interrupt	struct
case	extern	long	switch
cdecl	far	near	typedef
char	float	pascal	union
const	for	register	unsigned
continue	fortran	return	void
default	goto	short	volatile
do	huge	signed	wchar_t *
double	if	sizeof	while

C Keywords (Single-Scored) (Double-Scored)

_api	_interrupt	__api	__interrupt
_asm	_loadds	__asm	__loadds
_based	_near	__based	__near
_cdecl	_pascal	__cdecl	__pascal
_export	_saveregs	__export	__saveregs
_far	_segment	__far	__segment
_far16	_segname	__far16	__segname
_fastcall	_self	__fastcall	__self
_fortran	_stdcall	__fortran	__stdcall
_huge	_syscall	__huge	__syscall

Preprocessor Keywords

define	ifdef
elif	ifndef
else	include
endif	line
error	pragma
ident	undef
if	

Preprocessor Pragmas

alloc_text	inline_recursion *	plmn
auto_inline *	intrinsic	same_seg
check_pointer	linesize	search_lib
check_stack	loop_opt	segment
code_seg *	message	setlocale *
comment	native_caller *	skip
data_seg	optimize	subtitle
function	pack	switch_check
hdrstop *	page	title
init_seg *	pagesize	vtordisp *
inline_depth *	plmf	warning *

IMPLIB

This section describes the Microsoft Import Library Manager (IMPLIB) version 1.40. This utility creates an import library from one or more module-definition (.DEF) files and dynamic-link libraries (DLLs) for use in resolving external references from a Windows-based program to a DLL. IMPLIB version 1.40 is designed to use .DEF files and DLLs that work with the Microsoft Segmented-Executable Linker, versions 5.30 and later.

About Import Libraries

An “import library” is a static library (usually with a .LIB extension) that can be read by the LINK utility. You specify the import library to LINK in the same ways you specify standard libraries created by the LIB utility. You can use LIB to combine an import library with other static libraries, either standard or import. For more information on LINK, see Chapter 13. For more information on LIB, see Chapter 17.

Import libraries are recommended for resolving references from applications to DLLs. Without an import library, an external reference to a dynamic-link routine must be either declared in an **IMPORTS** statement in the application’s .DEF file or explicitly coded in your program.

This section assumes you are familiar with import libraries, dynamic linking, and module-definition files. For information on module-definition files, see Chapter 14. For information on dynamic linking and import libraries, see the *Programmer’s Guide.*

IMPLIB uses only the following statements from a module-definition file and ignores other text in the .DEF file:

- **LIBRARY**
- **EXPORTS**
- **INCLUDE**

The IMPLIB Command Line

To run IMPLIB, use the following command line:

IMPLIB [[*options*]] *implibname* {*dllfile*... | *deffile*...}

The *options* field specifies IMPLIB options, which are explained in the next section.

The *implibname* field specifies the name for the new import library.

The *dllfile* field specifies the name of a DLL. You can use the *deffile* field to specify a module-definition file for the DLL rather than the DLL itself. You can enter multiple *dllfile* and *deffile* specifications. When you specify a DLL, IMPLIB puts all exports from the DLL into the import library. To include only a subset of the DLL's exported items in the import library, specify a module-definition file that contains only those exports.

IMPLIB does not assume default extensions for any field. You must specify the full names of input and output files and include the file extensions. You can specify a path with a filename.

Example

```
IMPLIB mylib.lib mylib.dll
```

This command creates the import library named MYLIB.LIB from the dynamic-link library MYLIB.DLL.

Options

Options names are not case sensitive and can be abbreviated to the shortest unique name. IMPLIB has the following options:

/H[[ELP]]
: Calls the QuickHelp utility. If IMPLIB cannot find the help file or QuickHelp, it displays a brief summary of IMPLIB command-line syntax.

/NOI[[GNORECASE]]
: Preserves case sensitivity in exported and imported names.

/NOL[OGO]
Suppresses the IMPLIB copyright message.

/?
Displays a brief summary of IMPLIB command-line syntax.

RM, UNDEL, and EXP

This sections describes the following utilities:

- Microsoft File Removal Utility (RM) version 2.00
- Microsoft File Undelete Utility (UNDEL) version 2.00
- Microsoft File Expunge Utility (EXP) version 2.00

RM, UNDEL, and EXP run under real-mode MS-DOS. You can use these utilities to create hidden backup files, recover the files, and delete them when no longer needed. You can also use them to manage the backup files created by the Microsoft Programmer's WorkBench (PWB).

Be sure to use matching versions of the RM, EXP, and UNDEL utilities. You can check version numbers by running each utility with the /? option.

Overview of the Backup Utilities

The RM, UNDEL, and EXP utilities help you create backup files and manage those files. RM ("remove") moves a file into a hidden subdirectory named DELETED. UNDEL ("undelete") makes the file visible again by moving it into DELETED's parent directory. EXP ("expunge") deletes the DELETED directory and all files contained within; after being expunged, these files cannot be restored by UNDEL.

Use RM, UNDEL, and EXP to manage backup files created by PWB. PWB stores backup files in a DELETED directory when its **Backup** switch is set to **Undel**.

The RM Utility

The RM utility moves one or more files to a hidden directory named DELETED. DELETED is a subdirectory of the directory that contains the file being deleted. Thus RM may create many DELETED directories on your drives or floppy disks. RM creates a DELETED subdirectory of a given directory if one does not already exist. Run RM using the following command line:

RM [[*options*]] [[*files*]]

The *files* field specifies the files to be deleted. You can name more than one file, and you can use operating-system wildcards (***** and **?**). You can specify a path with the filename. RM prompts for permission before removing a read-only file unless /F is specified.

RM has the following options; the option names are not case sensitive:

/F
: Deletes read-only files without prompting for permission.

/HELP
: Calls the QuickHelp utility. If RM cannot find the help file or QuickHelp, it displays a brief summary of RM command-line syntax.

/I
: Inquires for permission before deleting any file.

/K
: Keeps read-only files without deleting or prompting.

/R *directory*
: Recurses into subdirectories of *directory* and moves all files into corresponding DELETED directories.

/?
: Displays a brief summary of RM command-line syntax.

Example

```
RM /R \PROJECT
```

This command line tells RM to delete all files in the directory tree whose root is the directory named PROJECT. The PROJECT directory lies in the root directory on the current drive. RM moves all files in this tree to hidden directories named DELETED, each of which is created as a subdirectory of a directory that contains the file to be deleted.

The UNDEL Utility

The utility restores one or more deleted files by moving them from a hidden DELETED subdirectory to the parent directory. Run UNDEL using the following command line:

UNDEL [[{*option* | *files*}]]

The *files* field specifies the files to be restored. If you specify more than one file, separate the names with spaces. You cannot use operating-system wildcards (***** and **?**). You can specify a path with the filename. If more than one file in DELETED has the specified name, UNDEL lists the versions and prompts for

which file to restore. If a file with the same name already exists in the parent directory, UNDEL moves it to the DELETED directory before restoring the specified file.

To list all files in the current directory's DELETED subdirectory, specify the UNDEL command alone. However, you cannot list files in a remote directory; UNDEL does not accept a path without a filename.

UNDEL has the following options; the option names are not case sensitive:

/HELP
: Calls the QuickHelp utility. If UNDEL cannot find the help file or QuickHelp, it displays a brief summary of UNDEL command-line syntax.

/?
: Displays a brief summary of UNDEL command-line syntax.

Example

```
UNDEL \PROJECT\WORK\REPORT.TXT
```

This command line tells UNDEL to restore the file called REPORT.TXT in the directory \PROJECT\WORK on the current drive. If a file called REPORT.TXT already exists in that directory, UNDEL changes the file to a backup file in \PROJECT\WORK\DELETED before restoring REPORT.TXT. If more than one file called REPORT.TXT exists in \PROJECT\WORK\DELETED, UNDEL prompts for which version to restore.

The EXP Utility

The EXP utility removes a hidden DELETED directory and all files contained within. To run EXP, use the following command line:

EXP [[*options*]] [[*directories*]]

The *directories* field specifies one or more directories containing DELETED directories to be expunged. If no directory is specified, EXP deletes the current directory's DELETED subdirectory.

EXP has the following options; the option names are not case sensitive:

/HELP
: Calls the QuickHelp utility. If EXP cannot find the help file or QuickHelp, it displays a brief summary of EXP command-line syntax.

/Q
: Suppresses display of the names of deleted files.

/R
: Recurses into subdirectories of the current or specified directory and expunges all DELETED directories and files.

/?
Displays a brief summary of EXP command-line syntax.

Example

```
EXP /R \PROJECT\WORK
```

This command line tells EXP to:

- Delete the hidden directory \PROJECT\WORK\DELETED along with any files in the directory.
- Recurse through the tree whose root is \PROJECT\WORK and delete any DELETED directories and associated files.

WX/WXServer

This section describes the Microsoft WX/WXServer Utility version 1.50. This utility runs a Windows-based program from an MS-DOS prompt within the Windows operating system. The utility has two parts:

- WX.EXE is a command-line utility that runs a Windows-based application from an MS-DOS prompt either in a full screen or in a window.
- WXSRVR.EXE is a Windows-based program that must be running when you use WX.

Microsoft Programmer's WorkBench (PWB) uses WXServer to run your Windows-based programs and Microsoft CodeView for the Windows operating system.

Running WX/WXServer

WX/WXServer requires the Windows operating system 386 enhanced mode. The [386Enh] section of SYSTEM.INI must contain the following line:

```
device=[path\]vmb.386
```

To use WX/WXServer, you start WXSRVR.EXE once and leave it running in the Windows operating system. You then can run a Windows-based application in an MS-DOS session, either in a window or full screen, by running WX.EXE.

Running WXSRVR.EXE

To run WXSRVR.EXE, start it as an application in the Windows operating system using one of the following methods:

- In the File Manager window, double-click the WXSRVR.EXE file (or select it and press ENTER).

- From the File menu in either the Program Manager or File Manager window, choose the Run command and type WXSRVR in the Command Line box.
- In the Program Manager window, double-click the WXServer icon. If you added the MASM.GRP to your windows desktop, the WXServer icon will be there. (For information on adding the MASM.GRP to your windows desktop, see *Getting Started*.)

When WXSRVR.EXE is running, it does not come up in a window or full screen. The only features you can see are the WXServer icon on the desktop and the About Microsoft WXServer dialog box. The About dialog box is displayed if WXSRVR.EXE is running and you do one of the following:

- Open the Control menu for WXServer and choose About WXServer.
- Switch to WXServer.
- Double-click the WXServer icon on the desktop.

The About dialog box contains four command buttons, described below:

Button Label	Action
OK	Accepts changes to the Timer Delay setting
Cancel	Closes the dialog box and ignores any changes
Hide	Removes the icon from the desktop and leaves WXServer running
Terminate	Closes WXServer

The dialog box also contains a box labeled Timer Delay. The setting in this box determines how often WXServer checks for requests by WX to run a program. The default setting of 100 milliseconds is appropriate for most situations. You can try increasing the timer delay if your system seems to be running too slowly. You can decrease the timer delay to get a quicker response to the WX command.

If you want to hide the WXServer icon, you can do so in one of two ways:

- Run WXSRVR.EXE using the /H option, as in the following command:

  ```
  wxsrvr /h
  ```

- Open the About dialog box and choose Hide.

To restore the WXServer icon to the desktop, start WXSRVR.EXE again.

To end WXSRVR.EXE, do one of the following:

- From the Control menu for WXServer, choose Close.
- Open the About dialog box and choose Terminate.

Running WX

To run WX, open an MS-DOS session and enter the following command at the MS-DOS prompt:

WX [[*options*]] *program* [[*arguments...*]]

The *program* is the filename of the Windows-based application you want to run. The *arguments* are any command-line options, filenames, or other arguments required to run your application. Specify *options* to WX before specifying the program name. Options names are not case sensitive and can be abbreviated to the shortest unique name. WX has the following options:

/A[[SYNC]]
: Runs the program asynchronously. By default, WX runs synchronously, which pauses the MS-DOS session until the program ends. The /A option lets other programs run in the same MS-DOS session while the Windows-based program is running.

  When running synchronously, WX requires that the MS-DOS prompt run in the background. (To do this, open the Control menu for the MS-DOS prompt and choose the Settings command, then turn on the Background option under Tasking Options.)

  WXServer can run only one program synchronously, but it can run additional programs asynchronously.

/B[[ATCH]]
: Suppresses dialog messages if errors occur. This option is useful for batch processing.

/H[[ELP]]
: Displays a brief summary of WX command-line syntax.

/N[[OLOGO]]
: Suppresses the WX copyright message.

/W[[INDOW]]
: Runs a program from an MS-DOS prompt that is in a window rather than full screen.

/?
: Displays a brief summary of WX command-line syntax.

Example

The following command uses WX to run an application in a window without the copyright display:

```
wx /w /nologo project
```

The WX Environment Variable

WX.EXE uses the environment variable WX if the variable has been set. For example, if you always want to run WX in a window without the copyright display, use the following command to set the environment variable:

```
SET WX=/W /NOLOGO
```

After the WX environment variable is set, use the following command to run the application:

```
wx project
```

P A R T 5

Using Help

Chapter 21 Using Help 663

CHAPTER 21

Using Help

MASM offers two systems for accessing Help:

- The Microsoft Advisor, found within the Programmer's WorkBench (PWB) and CodeView
- QuickHelp, the standalone Help program

Both systems provide the same information on important topics and utilities provided with the development system, which include the language, run-time libraries, PWB, and CodeView.

Structure of the Microsoft Advisor

The Microsoft Advisor can be compared to a librarian managing a collection of books. Each book, or Help file, has its own table of contents, index, and pages of information. The Advisor organizes the Help files with a global contents and index. All of the files are listed, and their specific tables of contents and indexes can be accessed through the global references. The global contents screen is shown in Figure 21.1.

Figure 21.1 Microsoft Advisor Global Contents Screen

You can access a variety of information from the Help system. Information is available on the languages, run-time libraries, errors, and the Help system itself.

Navigating Through the Microsoft Advisor

You request information about a topic in a window by moving the cursor to it and pressing F1 or by clicking it with the right mouse button. The Help system then searches through the Help files for information about the topic. If it finds the topic, the Help system displays information in the Help window. If Help cannot be found for a particular word or symbol, a message informs you that no information is associated with the topic.

Sometimes, a topic with the same name occurs in several Help files. When you request Help in PWB for one of these names, PWB displays a dialog box in which you can select the context of the topic. The Next command on the Help menu takes you to the next occurrence. When you are using QuickHelp, the first topic is displayed. You can then press E to go to the next occurrence.

Note CodeView does not use the right mouse button for Help in the Source window. Clicking the right mouse button on a line in the Source window executes the program to that line. However, the right mouse button activates Help in the other CodeView windows.

Using the Help Menu

The simplest method for accessing Help is by using the commands found in the PWB and CodeView Help menus. These commands present information in the Help window.

Command	Description
Index	Displays the global index of categories (see Figure 21.2).
Contents	Displays the global Help contents screen.
Topic: *topic*	Provides information about the topic at the cursor. If information about the topic is available, the topic's name is appended to the Topic command. Otherwise, this command is dimmed.
Help on Help	Displays information about using Help itself.

Figure 21.2 Microsoft Advisor Global Index Screen

PWB and QuickHelp provide additional commands to access Help. These commands are described in the program-specific sections at the end of this chapter.

Using the Mouse and the F1 Key

You can use the mouse and the F1 key to get information about any menu command or dialog box, as well as information on keywords, operators, and run-time library functions.

Help on Menu Commands

To view information about a menu item:

1. Open the menu.
2. Drag the mouse to the command and click the right mouse button.
   -or-
   Use the ARROW keys to select the command and press F1.

The information on the selected command is displayed in a Help dialog box.

Help on Dialog Boxes

To view information about a dialog box:

1. Open the dialog box.
2. Click the Help button.
   -or-
   Press F1.

The information on the dialog box is displayed in a Help dialog box.

Using Hyperlinks

Hyperlinks are cross-references that connect related information.

Hyperlinks enclosed by the < and > characters are called “buttons.” You can navigate through the Help system by using these buttons.

You can press TAB to move to the next hyperlink button within the Help window. Pressing SHIFT+TAB moves to the previous button. In PWB and CodeView, typing any letter moves the cursor to the next button that begins with that letter; holding down SHIFT and typing a letter moves the cursor backward.

The Microsoft Advisor also recognizes language keywords, library functions, constants, and similar identifiers as hyperlinks, but these are not marked. Unmarked hyperlinks are recognized by the Microsoft Advisor wherever they appear in the Help text or *in your source code*. However, an unmarked hyperlink is not delimited with the < and > characters, and you can’t move to it with the TAB key.

An unmarked hyperlink can be activated only by pointing to it with the mouse and clicking the right mouse button or by placing the cursor on it and pressing F1. In QuickHelp, press the S key and then type the text of the hyperlink in the dialog box. In CodeView, use the Help (**H**) Command-window command.

To activate a hyperlink with the mouse:

1. Move the mouse pointer to the hyperlink.
2. Click the right mouse button.
   -or-
   Click the left mouse button twice (double-click). Double-clicking works only in the Help window.

To activate a hyperlink with the keyboard:

1. Press TAB, SHIFT+TAB, or the ARROW keys to move the cursor to the hyperlink. When you move the cursor to a hyperlink button, the entire button is selected.
2. Press F1, ENTER, or SPACEBAR.

Any of these actions displays information about the topic at the cursor. If the topic isn't a hyperlink, a message informs you that no information on the topic could be found.

Note CodeView uses the right mouse button differently in the Source window. Clicking the right button in the Source window executes the program to the line where the mouse was clicked. However, once the Help window is displayed, the right mouse button can be used to activate hyperlinks.

Using Help Windows and Dialog Boxes

The Microsoft Advisor displays information in windows or dialog boxes. Help windows and dialog boxes function in the same way as other windows and dialog boxes found in PWB and CodeView. For a complete description of windows and dialog boxes, see Chapter 4, "User Interface Details."

Using the Help Window

The Help window displays various contents, indexes, and information about selected topics. Some screens of information are larger than the Help window; information beyond the window borders can be viewed by using the scroll bars or the cursor-movement keys. The -◆- symbol indicates the end of information in the Help window.

Navigating with Hyperlinks

At the top of most Help windows is a row of hyperlink buttons that are useful for moving through the Help system:

Button	Description
<Up>	Moves upward in the hierarchy of Help screens. Since information is ordered in a logical way, moving from the general to the specific, this command is useful for moving up the information tree.
<Contents>	Displays the global contents screen. This command is useful because it returns you to a known point in the Help hierarchy. For some Help databases, the Contents button goes to that database's contents.
<Index>	Displays the global index list. Selecting an item from the list displays the index for that category. When you are viewing an index for a particular category, the letters on the bar across the top of the screen are hyperlinks. For some Help databases, the Index button goes to that database's index.
<Back>	Moves you to the last Help you saw.

The Contents and Index commands on the Help menu always display the global Contents and Index screens.

Screens on a particular topic are frequently grouped together in a Help file. You can press CTRL+F1 to display information about the next physical topic in the Help file.

Viewing the Previous Help

The Microsoft Advisor remembers the last 20 Help screens you've accessed. To return to a previous screen, use the <Back> button or press ALT+F1 as many times as necessary to return to the screen you want to see. The Help screen that appears is active; you can ask for Help on any of its hyperlinks or topics.

You can always return to the global Contents screen by choosing Contents from the Help menu or by pressing SHIFT+F1.

Copying and Pasting Help

Any text that appears in the PWB Help window can easily be copied to another window. For example, to test an example program from the Help window, you only have to copy it to a new file and compile it. You select and copy text in the Help window just as you do for any other window in PWB.

If you are using QuickHelp, you cannot cut and paste directly into your text editor. However, you can use the commands in the QuickHelp Paste menu to extract predetermined portions of the Help screen to a file. To change the name of the paste file, choose Rename Paste File from the File menu.

If you paste example code from QuickHelp, you will need to delete the "Topic" line at the beginning and the -◆- line at the bottom of the topic before you can successfully assemble or compile the example.

Closing the Help Window

Once you're through working with the Help system, you can close the active Help window.

To close the Help window:

- Choose the Close button in the upper left corner of the window.
  -or-
  Press ESC.

Using Help Dialog Boxes

Help dialog boxes provide information about menu commands and dialog boxes. A Help dialog box appears over the windows on the desktop. Unlike the Help window, a Help dialog box must be closed before you can continue. The Cancel button closes the Help dialog box.

To view information about a dialog box:

- Choose the Help button in the dialog box.
  -or-
  Press F1.

To close a Help dialog box:

- Choose the Cancel button.
  -or-
  Press ESC.

Accessing Different Types of Information

This section presents some strategies for accessing the different types of information available within the Help system.

Keyword Information

The Help system contains information about all keywords, operators, symbolic constants, and library functions in the development system. If you know the exact name of a keyword, you can type it in a window and click it with the right mouse button or press F1. For operators that do not have an alphabetic name, you must select the operator before activating Help. You can also use the index for the appropriate category of Help.

To get Help using the index:

1. From the Help menu, choose Index.
   -or-
   Choose the Index button on any Help screen.
2. Choose the appropriate category of Help from the list of indexes.

   Each index has a row of letters across the top.
3. Choose the keyword's first letter from the row of letters. If you want Help for
   a nonalphabetic operator, choose the asterisk (*****).
4. Scroll down the list of entries and choose the topic's hyperlink.

In PWB, you can get Help on a keyword or operator by using the **Arg** function, typing the keyword in the Text Argument dialog box, then pressing F1. Assuming that **Arg** is assigned to ALT+A (the default assignment), the following procedure displays Help for the **mov** function.

To get Help using the Arg function in PWB:

1. Press ALT+A

   PWB displays the message `Arg[1]` on the status bar.
2. Type `mov`.

   When you type the first letter of the keyword, PWB displays the Text Argument dialog box. Continue typing the keyword.
3. Press F1.

   PWB displays the Help for the **mov** function.

To get Help on a topic in QuickHelp:

1. Choose Search from the View menu or press the S key.

   QuickHelp displays a dialog box where you can type the topic name.
2. Type the keyword.
3. Choose OK or press ENTER.

Figure 21.3 shows a PWB window with the information for the **mov** function.

Figure 21.3 Help for mov in a PWB Window

When information about a programming-language keyword or function is shown in the Help window, two additional hyperlink buttons are displayed.

<Description>
: Provides a detailed explanation of the function. When the description information is displayed, the button changes to <Summary>. Click this button to return to the summarized information about the function.

<Example>
: Displays source code that provides an example of how the function is used.

Topical Information

The Help system is useful when you want an overview of the available reference topics or when you only have a general idea of what information you need. Start with the global contents screen, and then select any hyperlinks that relate to the topic. You can traverse the hyperlinks until you locate the necessary information.

Menu and Dialog-Box Help

You can get information about any menu command or dialog box by pressing F1 when the menu command is highlighted or the dialog box is displayed. This is helpful when you are first learning to use the development system and you are not completely familiar with all of the features.

Error Help

The Microsoft Advisor provides information about compiler and linker error messages. Whenever a message is displayed on the bottom line of the window in PWB, press F1 to see Help on that error.

You can also get Help for any error in the Build Results window.

- **To find the meaning of an error message using the mouse:**
  1. Position the mouse pointer on the error number in the Build Results window.
  2. Click the right mouse button.

- **To find the meaning of an error message using the keyboard:**
  1. Move the cursor to the Build Results window.
  2. Position the cursor on the error number.
  3. Press F1.

Help on error messages is also available directly by executing the **Arg** function, typing the error number and its alphabetic prefix, and then pressing F1. Make sure that you type the number exactly—case is significant.

Using Different Help Screens

In addition to the global screens and the topic screens that have already been described in this chapter, the Microsoft Advisor contains some other types of screens that you use in special ways.

Using Index Screens

An index screen has a bar of letters at the top of the screen, below the row of hyperlink buttons. Each letter on the bar is a hyperlink to that letter's list of index entries. The asterisk (*****) at the end of the bar is also a hyperlink. This screen lists the nonalphabetic entries. Click the right mouse button on the letter to see that part of the index.

Figure 21.4 shows the PWB index screen for the A category. Below the row of alphabet hyperlinks is a list of index entries. Each entry is a hyperlink to the indicated topic.

Figure 21.4 PWB Index

Using Topic Lists

Some topics are not a screen of text with fixed hyperlink buttons at the top. Instead, they are a list of topics in which each line is a hyperlink. The entire line is highlighted. You can point to the line and click the right mouse button to activate the hyperlink. You can also use the UP ARROW and DOWN ARROW keys to select a topic, and then press F1 or ENTER to go to that topic.

Using Help in PWB

PWB provides additional Help features to help you find the information you need.

Opening a Help File

You can open Help files temporarily in PWB by using the **SetHelp** function. If you keep rarely used Help files in a directory that is not listed in the HELPFILES environment variable, you can still open the files when you need them.

To open another Help file in PWB:

1. Execute the **Arg** function (press ALT+A).
2. Type the name of the Help file to open. PWB displays the Text Argument dialog box when you type the first letter of the filename.
3. Execute the **SetHelp** function (press SHIFT+CTRL+S).

To close a Help file, execute **Arg Meta** *file* **SetHelp**. That is, press ALT+A, F9, type the filename, then press SHIFT+CTRL+S.

Global Search

The Global Search command on the Help menu in PWB lets you search all open Help files for a string of text or a regular expression. All text in the Help files is searched, not just the topic names. A global search results in a list of topics, each of which contains text that matches the search string. QuickHelp can also perform global Help searches, but does not offer regular-expression matching.

Searching all the Help can take a long time. Therefore, it is recommended that you use the Global Search command only after you have tried other methods of finding the information you need.

Running a Global Search

When you choose the Global Search command, PWB displays the Global Search dialog box where you can specify options for the search. Enter the string or pattern you want to locate in the Find Text box. If you want the search to be case sensitive, turn on the Match Case option. To match a regular expression rather than literal text, turn on the Regular Expression option.

Regular expressions allow you to specify general patterns of text or several alternative strings to match. The current regular-expression syntax is displayed in parentheses after the Regular Expression option. For more information about searching with regular expressions, see Chapter 5, “Advanced PWB Techniques,” and Appendix B.

When you choose OK, PWB starts searching for the specified string or regular expression. The search begins with the Help file that was opened most recently. Because the search can take a long time, it is recommended that you choose a likely category of Help from the global Contents screen before starting a global search.

When you start a global search, PWB displays a dialog box that shows the progress of the search. Choose the Stop Search button at any time to stop the search and view the partial results. When the search ends, PWB displays a list of matching topics.

Using Search Results

When the search is finished, or when you halt the search by choosing Stop Search, PWB displays a list of the topics that contain text that matches the specified string. Each topic is represented by its title if it has one, followed by the name of the database that contains the topic, and sometimes followed by the topic name.

To select a topic from the list:

- Click the right mouse button on the line.
  -or-
  Press the UP ARROW and DOWN ARROW keys until the topic is highlighted, and then press ENTER or F1.

PWB displays the selected topic. If that topic does not supply the information that you need, go back to the list and select another topic.

To go back to the list:

- Choose Search Results from the Help menu.
  -or-
  Press ALT+F1 until the list is displayed.

Restricting the Search

By default, PWB performs a global search in all open databases. There are several ways to control which databases are searched:

- Before the search, display Help from the database that is most likely to contain the information you want. When you run the search, choose Stop Search when the dialog box indicates that the first database has been searched.
- Close some databases by using the **Meta** form of the **SetHelp** function.
- Set the HELPFILES environment variable to the file or files to be searched by using the Environment Variables command on the Options menu. The list of files cannot exceed the MS-DOS limit of 128 characters.

  Note that the changes you make to HELPFILES may be restored the next time you start PWB or use the project, depending on the settings of the **Envcursave** and **Envprojsave** switches.
- Choose the Editor Settings command from the Options menu. Then select PWBHELP as the Switch Owner and Text as the switch type. Assign a value to the **Helpfiles** switch to open other Help files in addition to the ones listed in the HELPFILES environment variable.

To see a list of all open Help files and databases, execute the **Arg ? SetHelp** command. The default keystrokes for this are ALT+A, ?, SHIFT+CTRL+S. The resulting list of physical Help files and Help databases is displayed in the Help window.

Using QuickHelp

QuickHelp is a separate application that provides access to any Help file. It uses the same Help files as the Microsoft Advisor and presents information about topics in the same way. QuickHelp is designed for the developer who prefers using command-line utilities or another editor and doesn't have access to the Microsoft Advisor through PWB.

Major utilities that come with MASM invoke QuickHelp and display related information when you use the /Help option. You can also use QuickHelp from the command line, as explained in the following sections.

Using the /Help Option

You can get immediate information on the major MASM components by using the /Help option. The following procedures use the LIB utility as an example. However, you can use these methods for all command-line utilities in the development system.

To learn about the LIB utility:

- At the operating-system command line, type:

```
LIB /Help
```

  LIB starts QuickHelp which displays information about LIB.

Using the QH Command

You can also run QuickHelp from the MS-DOS command line or by double-clicking the MASM 6.1 Reference icon in your Windows operating system Program Manager MASM group. (For more information on adding the MASM.GRP program group to your windows desktop, see *Getting Started*.)

To get Help on the LIB utility:

- At the operating-system command line, type:

```
QH LIB.EXE
```

  You can type the name of any Microsoft utility instead of LIB.

To start QuickHelp to view the Advisor Contents screen:

- At the operating-system command line, type:

```
QH Advisor
```

In addition to information about programs, QuickHelp can also display information about compiler and run-time errors. Type `QH` and the error number with its alphabetic prefix on the command line.

Opening and Closing Help Files

When you run QuickHelp, it looks for the environment variable HELPFILES and opens all listed .HLP files. If the HELPFILES variable isn't defined, QuickHelp opens all .HLP files in directories specified by the PATH environment variable.

Warning Windows-based Help files are not compatible with QuickHelp. Make sure that Windows-based Help files are not listed in the HELPFILES environment variable.

Choose the List Database command on the File menu to view a list of all the open Help files.

To open additional Help files:

1. Choose the Open Database command from the File menu.
2. Type the name of the Help file to be opened in the dialog box that appears. You can specify all Help files in a directory by typing `*.HLP`.
3. Press ENTER or choose the OK button.

To close an open Help file:

1. Choose the Close Database command from the File menu.

   The File menu changes to a list of open Help files.

2. Choose the Help file to close.

Displaying a Topic

You can view information about a topic by using the Search command on the View menu. When topic information is displayed, it is shown in the same format as information presented by the Microsoft Advisor.

To display a topic from any of the open Help files:

1. Choose the Search command from the View menu.
2. Type the topic you want information about in the dialog box.
3. Click the OK button or press ENTER.

QuickHelp searches for the topic in the open Help files. If the topic cannot be found, a dialog box informs you that the search failed. If the search is successful, information about the topic is displayed in the QuickHelp window.

Navigating Through Topics

A series of commands on the View menu allow you to display selected topics. These commands include the following:

Command	Description
View History	Displays a list of all the topics that have recently been displayed. See "Using Topic Lists" on page 673 for information on using the list.
View Last	Displays the last topic you looked at.
View Next	Displays the next topic in the Help file.
View Back	Moves backward one topic in the Help file.

Using the QuickHelp Window

The QuickHelp window shown in Figure 21.5 is similar to the Microsoft Advisor Help window. Information that doesn't fully fit in a window can be scrolled, and hyperlinks are used to display additional information. The main difference is that information presented in QuickHelp cannot be copied selectively.

Figure 21.5 The QuickHelp Window

Copying and Pasting in QuickHelp

To transfer information from QuickHelp to another program, specify a file with the Rename Paste File command in the File menu. Once the file is specified, choose the Current Window or the Current Topic command in the Paste menu to transfer the text to that file. Be sure to specify a new file when you paste because QuickHelp overwrites the existing file by default. To append to an existing file, choose the Paste Mode command from the Options menu. The default filename is PASTE.QH in the directory specified by the TMP environment variable.

More About QuickHelp

In addition to the features mentioned previously, QuickHelp has a variety of other options such as changing the appearance of the Help window, searching for text within topics, and controlling the function of the right mouse button.

To learn more about QuickHelp's features:

1. Make sure the QH.HLP file is open.
2. To view QuickHelp's Help, press F1.
   -or-
   To get information about a menu command, click it with the right mouse button, or highlight the command and press F1.

Managing Help Files

When you run the MASM Version 6.10 SETUP program, you are given a choice of whether to install the Help files. If you choose to install Help, SETUP copies the Help files to the directory that you specify. By default, this is the C:\MASM\Help directory.

Several other Microsoft products contain a Microsoft Advisor Help system. If you have more than one of these products, you can use all the files as one system by copying all .HLP files to a common directory. However, make sure that Windows-based Help files are separate from the Advisor Help files.

Some Help files, such as UTILS.HLP, exist in other Microsoft language products. When an existing Help file has the same filename as a MASM Help file, use the most recent file. Note that the files RC.HLP and UTILERR.HLP are obsolete and should be deleted or moved to another directory.

The HELPFILES environment variable tells the Advisor where to find Help files. You usually set this variable in AUTOEXEC.BAT. If you move the Help files, make sure to change the SET command in AUTOEXEC.BAT to point HELPFILES to the new location.

Managing Many Help Files

If you have a large number of Help files, you may reach a limit on the number of physical Help files or Help databases that can be open at one time. QuickHelp, PWB, and CodeView display a message when you have too many Help files. If this is the case, you must do one or more of the following:

- Delete all obsolete Help files.
- Move rarely used Help files to another directory. You can then open these files as you need them.
- Concatenate some Help files.

It is recommended that you always keep ADVISOR.HLP. Moreover, for Help on error messages, you must use the Help file for the tool that issues the error. It is recommended that you save backup copies of all Help files before concatenating, splitting, or deleting any files.

To open and close Help files in PWB, use the **SetHelp** function. To open and close Help files in QuickHelp, choose the Open Database and Close Database commands from the File menu.

You can get a listing of the open Help files in PWB and QuickHelp. These lists show the open Help files, the Help databases contained in the files, and the title for each database if it has one. To get a list of open Help files in PWB, execute the function sequence **Arg ? SetHelp**. With the default keystrokes, press ALT+A, type a question mark (**?**), then press SHIFT+CTRL+S. To get a list of open Help files in QuickHelp, choose the List Databases command from the File menu. Once you have created the list of Help files, you can print it for later reference.

Concatenating Help Files

To concatenate two or more physical Help files, use the MS-DOS COPY command. (Before you concatenate a Help file, save a backup copy of the files you are going to concatenate.) The syntax for using the COPY command to combine Help files is:

COPY *file* **/b** [[**+** *file* **/b**]]... *newfile*

Use a plus sign (+) between the filenames of the original Help files. Specify the /b option to copy the files as binary files. If you don't specify a new filename, the resulting file takes the name of the first file and the original file is overwritten.

You can use this command to combine two Microsoft Advisor Help files. For example, to create a physical Help file named ADVISOR.HLP that contains ADVISOR.HLP and QH.HLP, use the following command:

```
COPY ADVISOR.HLP /b + QH.HLP /b
```

You can also combine your own Help file (created using HELPMAKE) with Microsoft Help files.

Splitting Help Files

To split a physical Help file into its component databases, use the HELPMAKE utility. (Before you split a Help file, save a backup copy of the file you are going to split.) The syntax for using HELPMAKE to split a Help file is:

HELPMAKE /DS *file*

Specify the /DS option when splitting a Help file. For more information on the /DS option, as well as other uses of HELPMAKE, see Chapter 18. HELPMAKE creates individual physical files with the name of the original Help database. The resulting files are created in the current directory.

For example, the following command extracts the component Help databases from the UTILS.HLP file:

```
HELPMAKE /DS UTILS.HLP
```

The UTILS.HLP file itself is not changed. You can delete the unneeded component files and then concatenate the remaining files to create a new version of UTILS.HLP.

Appendixes

Appendix A Error Messages 685
Appendix B Regular Expressions 845

APPENDIX A

Error Messages

The error messages generated by MASM components fall into three categories:

- Fatal errors. These indicate a severe problem that prevents the utility from completing its normal process.
- Nonfatal errors. The utility may complete its process. If it does, its result is not likely to be the one you want.
- Warnings. These messages indicate conditions that may prevent you from getting the results you want.

All error messages take the form:

```
Utility: Filename (Line) : [Error type} (Code): Message text
```

Utility is the program that sent the error message.

Filename is the file that contains the error-generating condition.

Line is the approximate line where the error condition exists.

Error type is Fatal Error, Error, or Warning.

Code is the unique 5- or 6-digit error code.

Message text is a short and general description of the error condition.

Error Message Lists

Messages for each utility are listed below in numerical order, with a brief explanation of each error. The following two tables list the messages by utility and error code, respectively.

Table A.1 Error Codes Listed by Utility

Utility Name	Error Type	Code	Page
BSCMAKE	Fatal	BK1500 to BK1515	688
	Warnings	BK4500 to BK4503	691
C/C++ Expression Evaluators	All	CAN0000 to CAN0063; CXX0000, CXX0064	692
CodeView	Nonfatal	CV0000 to CV5014	700
CVPACK	Fatal	CK1000 to CK1021	716
	Warnings	CK4000 to CK4003	720
EXEHDR	Fatal	U1100 to U1140	721
Math Coprocessor	All	M6101 to M6205	722
H2INC	Fatal	HI1003 to HI1801	724
	Nonfatal	HI2000 to HI2555	727
	Warnings	HI4000 to HI4820	745
HELPMAKE	Fatal	H1000 to H1990	761
	Nonfatal	H2000 to H2003	766
	Warnings	H4000 to H4003	766
IMPLIB	Fatal	IM1600 to IM1608	767
	Nonfatal	IM2601 to IM2603	768
	Warnings	IM4600 and IM4601	768
LIB	Fatal	U1150 to U1203	769
	Nonfatal	U2152 to U2159	772
	Warnings	U4150 to U4158	773
LINK	Fatal	L1001 to L1129	775
	Nonfatal	L2000 to L2064	786
	Warnings	L4000 to L4086	791
ML	Fatal	A1000 to A1901	798
	Nonfatal	A2000 to A2901	802
	Warnings	A4000 to A6005	825
NMAKE	Fatal	U1000 to U1099; U1450 to U1455	828
	Nonfatal	U2001	838
	Warnings	U4001 to U4009	838
PWB	All	PWB3089 TO PWB3912; PWB12078 TO PWB12086	840
SBRPACK	All	SB1000 to SB1006	842

Table A.2 Error Codes Listed by Error Code Range

Code	Utility Name	Error Type	Page
A1000 to A1901	ML	Fatal	798
A2000 to A2901	ML	Nonfatal	802
A4000 to A6005	ML	Warnings	825
BK1500 to BK1515	BSCMAKE	Fatal	688
BK4500 to BK4503	BSCMAKE	Warnings	691
CAN0000 to CAN0063; CXX0000, CXX0064	C/C++ Expression Evaluators	All	692
CK1000 to CK1021	CVPACK	Fatal	716
CK4000 to CK4003	CVPACK	Warnings	720
CV0000 to CV5014	CodeView	Nonfatal	700
H1000 to H1990	HELPMAKE	Fatal	761
H2000 to H2003	HELPMAKE	Nonfatal	766
H4000 to H4003	HELPMAKE	Warnings	766
HI1003 to HI1801	H2INC	Fatal	724
HI2000 to HI2555	H2INC	Nonfatal	727
HI4000 to HI4820	H2INC	Warnings	745
IM1600 to IM1608	IMPLIB	Fatal	767
IM2600 to IM2603	IMPLIB	Nonfatal	768
IM4600 and IM4601	IMPLIB	Warnings	768
L1001 to L1129	LINK	Fatal	775
L2000 to L2064	LINK	Nonfatal	786
L4000 to L4086	LINK	Warnings	791
M6101 to M6205	Math Coprocessor	All	722
PWB3089 to PWB3912; PWB12078 to PWB12086	PWB	All	840
SB1000 to SB1006	SBRPACK	All	842
U1000 to U1099	NMAKE	Fatal	828
U1100 to U1140	EXEHDR	Fatal	721
U1150 to U1203	LIB	Fatal	769
U1450 to U1455	NMAKE	Fatal	828
U2001	NMAKE	Nonfatal	838
U2152 to U2159	LIB	Nonfatal	772
U4001 to U4009	NMAKE	Warnings	838
U4150 to U4158	LIB	Warnings	773

BSCMAKE Error Messages

Microsoft Browser Database Maintenance Utility (BSCMAKE) generates the following error messages:

- Fatal errors (BK1*xxx*) cause BSCMAKE to stop execution.
- Warnings (BK4*xxx*) indicate possible problems in the database-building process.

BSCMAKE Fatal Error Messages

BK1500 UNKNOWN ERROR
Contact Microsoft Product Support Services

BSCMAKE detected an unknown error condition.

Note the circumstances of the error and notify Microsoft Corporation by following the instructions in the "Microsoft Support Services" section of the introduction to this book.

BK1501 unknown character *character* **in option** *option*

BSCMAKE did not recognize the given character specified for the given option.

BK1502 incomplete specification for option *option*

The given option did not contain the correct syntax.

BK1503 cannot write to file *filename*

BSCMAKE could not write to the given file.

One of the following may have occurred:

- The disk was full.
- A hardware error occurred.

BK1504 cannot position in file *filename*

BSCMAKE 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.

BK1505 **cannot read from file** *filename*

BSCMAKE 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.

BK1506 **cannot open file** *filename*

BSCMAKE could not open the given file.

One of the following may have occurred:

- No more file handles were available. Increase the number of file handles by changing the FILES setting in CONFIG.SYS to allow a larger number of open files. FILES=50 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.

BK1507 **cannot open temporary file** *filename*

BSCMAKE could not open one of its temporary files.

One of the following may have occurred:

- No more file handles were available. Increase the number of file handles by changing the FILES setting in CONFIG.SYS to allow a larger number of open files. FILES=50 is the recommended setting.
- The TMP environment variable was not set to a valid drive and directory.
- The disk was full.

BK1508 **cannot delete temporary file** *filename*

BSCMAKE 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.

BK1509 out of heap space

BSCMAKE ran out of memory.

One of the following may be a solution:

- Reduce the memory that BSCMAKE will require by using one or more options. Use /Ei or /Es to eliminate some input files. Use /Em to eliminate macro bodies.
- Run BSCMAKE (or PWB if you are building a database in PWB) in an MS-DOS session within Windows to use virtual memory provided under the Windows operating system.
- Free some memory by removing terminate-and-stay-resident (TSR) software.
- Reconfigure the EMM driver.
- Change CONFIG.SYS to specify fewer buffers (the BUFFERS command) and fewer drives (the LASTDRIVE command).
- Run BSCMAKEV.EXE instead of BSCMAKE.EXE.

BK1510 corrupt .SBR file *filename*

The given .SBR file is corrupt or does not have the expected format.

Recompile to regenerate the .SBR file.

BK1511 invalid response file specification

BSCMAKE 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:

```
bscmake @
```

BK1512 database capacity exceeded

BSCMAKE 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.

BK1513 **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 BK4502, which will give the name of the .SBR file that caused the error.

BSCMAKE 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 file was missing.
- The wrong database name was specified.
- The database was corrupted, and a full build was required.

BK1514 **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 BK4502, 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 was corrupted, and a full build was required.

BK1515 *bscfile* **: incompatible version; cannot incrementally update**

The given database (.BSC file) was not created with this version of BSCMAKE. A database can be incrementally built only by the same version of BSCMAKE as the one used to fully build the database.

BSCMAKE Warning Messages

BK4500 **UNKNOWN WARNING**
Contact Microsoft Product Support Services

An unknown error condition was detected by BSCMAKE.

Note the circumstances of the error and notify Microsoft Corporation by following the instructions in the “Microsoft Support Services” section of the introduction to this book.

BK4501 **ignoring unknown option** *option*

BSCMAKE did not recognize the given option and ignored it.

If the given option is /r, it must be specified first on the BSCMAKE command line.

BK4502 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 that is not part of the original build of the database is specified during a rebuild of that database, BSCMAKE issues this warning. One of the following may be a cause:

- The wrong database name was specified.
- The database was deleted (error BK1513 will result).
- The database file was corrupted, requiring a full build.

BK4503 minor error in .SBR file *filename* ignored

The given .SBR file contained an error that did not halt the build. However, the resulting .BSC file may not be correct.

Recompile to regenerate the .SBR file.

CodeView C/C++ Expression Evaluator Errors

CAN0000 no error condition

No error has occurred, and this message should not appear.

You can continue debugging normally.

Note the circumstances of the error and notify Microsoft Corporation by following the instructions in the “Microsoft Support Services” section of the introduction to this book.

CAN0001 exception executing user function

The code being executed caused a general protection fault.

CAN0002 error accessing user memory

The expression attempts to reference memory that is not allocated to the program being debugged.

CAN0003 internal error in expression evaluator

The CodeView expression evaluator encountered an internal error.

Note the circumstances of the error and notify Microsoft Corporation by following the instructions in the “Microsoft Support Services” section of the introduction to this book.

CAN0004 syntax error

The syntax of the expression is incorrect.

Retype the expression with the correct syntax.

CAN0005 operator not supported

An unsupported C operator was specified in an expression.

You can usually write an equivalent expression using the supported C operators.

CAN0006 missing left parenthesis

Unbalanced parentheses were found in the expression.

Retype the expression with balanced parentheses.

CAN0007 missing right parenthesis

Unbalanced parentheses were found in the expression.

Retype the expression with balanced parentheses.

CAN0008 missing \at end of string

The double quotation mark (") expected at the end of the string literal was missing.

Retype the expression, enclosing the string literal in double quotation marks.

CAN0009 missing ' after character constant

The single quotation mark (') expected at the end of the character constant was missing.

Retype the expression, enclosing the character constant in single quotation marks.

CAN0010 missing left bracket

The expression contains unbalanced square brackets.

Retype the expression with balanced square brackets.

CAN0011 missing right bracket

The expression contains unbalanced square brackets.

Retype the expression with balanced square brackets.

CAN0012 missing left curly brace

The expression contains an unbalanced curly brace.

Retype the expression with balanced curly braces.

CAN0013 missing operator

An operator was expected in the expression but was not found.

Check the syntax of the expression.

CAN0014 missing operand

An operator was specified without a required operand.

Check the syntax of the expression.

CAN0015 expression too complex (stack overflow)

The expression entered was too complex or nested too deeply for the amount of storage available to the C expression evaluator.

Overflow usually occurs because of too many pending calculations.

Rearrange the expression so that each component of the expression can be evaluated as it is encountered, rather than having to wait for other parts of the expression to be calculated.

Break the expression into multiple commands.

CAN0016 constant too big

The CodeView C expression evaluator cannot accept an unsigned integer constant larger than 4,294,967,295 (0FFFFFFFF hexadecimal), or a floating-point constant whose magnitude is larger than approximately 1.8E+308.

CAN0017 symbol not found

A symbol specified in an expression could not be found.

One possible cause of this error is a case mismatch in the symbol name. Since C and C++ are case-sensitive languages, a symbol name must be given in the exact case in which it is defined in the source.

CAN0018 bad register name

A specified register 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 with MS-DOS on an 80386 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.

CAN0019 bad type cast

The CodeView C expression evaluator cannot perform the type cast as written.

One of the following may have occurred:

- The specified type is unknown.
- There were too many levels of pointer types.

  For example, the type cast:

  ```
  (char far * far *)h_message
  ```

  cannot be evaluated by the CodeView C expression evaluator.

CAN0020 operand types bad for this operation

An operator was applied to an expression with an invalid type for that operator.

For example, it is not valid to take the address of a register, or subscript an array with a floating-point expression.

CAN0021 struct or union used as scalar

A structure or union was used in an expression, but no element was specified.

When manipulating 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.

CAN0022 function call before _main

The CodeView C expression evaluator cannot evaluate a function before CodeView has entered the function **_main**. The program is not properly initialized until **_main** has been called.

Execute `g main;p` to enable function calls in expressions.

CAN0023 bad radix

The radix specified is not recognized by the CodeView C expression evaluator. Only decimal, hexadecimal, and octal radixes are valid.

CAN0024 operation needs l-value

An expression that does not evaluate to an l-value was specified for an operation that requires an l-value.

An l-value (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 l-value because it points to a specific memory location. The logical comparison `zed != 0` is not a valid l-value because it evaluates to TRUE or FALSE, not a memory address.

CAN0025 operator needs struct/union

An operator that takes an expression of struct or union type was applied to an expression that is not a struct or union.

Components of class, structure, or union variables must have a fully qualified name. Components cannot be entered without full specification.

CAN0026 bad format string

A format string was improperly specified.

Check the syntax of the expression.

CAN0027 tp addr not l-value

Check the syntax of the expression.

CAN0028 not struct/union element

An expression of the form `Struct.Member` or `pStruct->Member` was specified, but *member* is not an element of the structure.

The expression may not be parenthesized correctly.

CAN0029 not struct pointer

The member-selection operator (->) was applied to an expression that is not a pointer to a structure.

Check that the entire expression is parenthesized correctly, or type cast the address expression to the appropriate structure pointer type.

CAN0030 expression not evaluatable

The expression could not be evaluated as written.

This error is frequently caused by dereferencing a pointer which is not valid.

Check that the syntax of the expression is correct, and that all symbols are specified in the exact case as they are defined in the program.

CAN0031 expression not expandable

The CodeView C expression evaluator encountered an internal error.

You may be able to write an equivalent expression that can be evaluated correctly.

Note the circumstances of the error and notify Microsoft Corporation by following the instructions in the “Microsoft Support Services” section of the introduction to this book.

CAN0032 divide by 0

The expression contains a divisor of zero, which is illegal. This divisor may be the literal number zero, or it may be an expression that evaluates to zero.

CAN0033 error in OMF type information

The executable file did not have a valid OMF (Object Module Format) for debugging by CodeView.

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 executable file was not created with the high-level language released with this version of CodeView. Recompile the program with the current version of the compiler.
- The .EXE file may have been corrupted. Recompile and relink the program.

CAN0034 types incompatible with operator

The operand types specified are not legal for the operation.

For example, a pointer cannot be multiplied by any value.

You may need to type cast the operands to a type compatible with the operator.

CAN0035 overlay not resident

An attempt was made to access an overlay that is not currently resident in RAM.

Execute the program until the overlay is loaded.

CAN0036 bad context {...} specification

This message can be generated by any of several errors in the use of the context-resolution operator ({}).

- The syntax of the context-resolution operator ({}) was given incorrectly.

The syntax of the context operator is:

```
{[function],[module],[dll]}expression
```

This specifies the context of *expression*. The context operator has the same precedence and usage as a type-cast.

Trailing commas can be omitted. If any of `[function]`, `[module]`, or `[dll]` contain a literal comma, you must enclose the entire name in parentheses.

- The function name was spelled incorrectly, or does not exist in the specified module or dynamic-link library.

Since C is a case-sensitive language, *function* must be given in the exact case as it is defined in the source. The C expression evaluator ignores the CodeView case-sensitivity state set with the OC command or the Case Sensitive command in the Options menu.

- The module or DLL could not be found.

Check the full path name of the specified module or DLL.

CAN0037 out of memory

The CodeView C expression evaluator ran out of memory evaluating the expression.

CAN0038 function argument count and/or type mismatch

The function call as specified does not match the prototype for the function.

Retype the call with the correct number of arguments. Type cast each argument to match the prototype, as necessary.

CAN0039 symbol is ambiguous

The CodeView C expression evaluator cannot determine which instance of a symbol to use in an expression. The symbol occurs more than once in the inheritance tree.

You must use the scope resolution operator (::) to explicitly specify the instance to use in the expression.

CAN0040 function requires implicit conversion

Implicit conversions involving constructor calls are not supported by the CodeView C expression evaluator.

CAN0041 class element must be static member or member function

A nonstatic member of a class (or structure or union) was used without specifying which instantiation of the class to use.

Only static data members or member functions can be used without specifying an instantiation.

CAN0042 bad line number

This error should never occur.

Note the circumstances of the error and notify Microsoft Corporation by following the instructions in the “Microsoft Support Services” section of the introduction to this book.

CAN0043 this pointer used outside member function

This pointer can only be used for nonstatic member functions.

CAN0044 use of _based(void) pointer requires :> operator

A pointer based on **void** cannot be used directly. You must form a complete pointer using the :> operator.

CAN0045 not a function

An argument list was supplied for a symbol in the program that is not the name of a function.

For example, this error is generated for the expression

```
queue( alpha, beta
```

when `queue` is not a function.

CAN0046 argument list required for member function

An expression called a member function but did not specify any actual parameters.

CAN0047 argument list does not match a function

An expression called a function with an actual parameter list that did not match the formal parameter list of any function with the same name defined in the program.

Overloaded functions can be called only if there is an exact parameter match, or a match that does not require the construction of an object.

CAN0048 calling sequence not supported

A function specified in the expression uses a calling sequence not supported by the CodeView C expression evaluator. You cannot call this function in a CodeView expression.

CAN0049 obsolete OMF - please relink program

The program used an old OMF (Object Module Format).

The program must be linked with LINK version 5.30 or later, and packed with CVPACK version 4.0 or later.

CAN0050 left side of :: must be class/struct/union

The symbol on the left side of the scope-resolution operator (::) was not a class, structure or union.

CAN0051 more than one overloaded symbol specified in breakpoint

CodeView could not determine which of more than one overloaded symbol to use as a breakpoint.

CAN0052 member function not present

A member function was specified as a breakpoint but could not be found. This error can be caused by setting a breakpoint at a function that has been inlined.

Recompile the file with inlining forced off (/Ob0) to set a breakpoint in this function.

An expression called a function that was not defined.

CAN0053 nonfunction symbol match while binding breakpoints

A symbol used as a breakpoint was not a function. This error can be caused by specifying a data member as a breakpoint.

CAN0054 register in breakpoint expression illegal

A register cannot be used in a breakpoint expression.

CAN0055 ambiguous symbol in context operator

A symbol in the context operator ({}) referred to more than one symbol in the program.

The scope resolution operator (::) may be able to resolve the ambiguity.

CAN0056 error in line number

An invalid line number was specified.

CAN0057 no code at line number

No code was generated for the specified line number. It cannot be used as a breakpoint.

CAN0058 overloaded operator not found

A class type was specified as the left operand in an expression, but an overloaded operator was not defined for the class.

CAN0059 left operand is class not a function name

The left operand of a function call was a class name and could not be resolved to a function call. This error can be caused by omitting the name of a member function in an expression.

CAN0060 register is not available

An expression specified a register that cannot be used.

This error can be caused by trying to access a register that does not exist on the machine running CodeView, for example, accessing 80386-specific registers on an 8088-based machine.

CAN0061 function nesting depth exceeded

The expression contains a function nesting depth greater than the limit.

The expression should be modified to reduce the nesting depth.

CAN0062 constructor calls not supported

An expression made a call to a constructor.

Expressions cannot make explicit calls to constructors or make conversions that require a call to a constructor.

CXX0063 overloaded operator -> not supported

The expression used an overloaded class member access operator (->).

CXX0064 can't set breakpoint on bound virtual member function

A breakpoint was set on a virtual member function through a pointer to an object, such as:

```
pClass->vfunc( int );
```

A breakpoint can be set on a virtual function by entering the class, such as:

```
Class::vfunc( int );
```

CodeView Error Messages

CV0000 no error; NOERROR; No Error Condition

You should not normally receive this error message since CV0000 indicates that no error occurred.

CV0002 **no such file or directory**

The specified file does not exist or a path 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 Source (VS) 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.

CV0003 **program terminated: restart to continue**

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.

CV0005 **I/O error**

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.

CV0007 **number of arguments exceeds DOS limit of 128**

CodeView is not able to restart the program that is being debugged because the number of arguments to the executable program exceeds the limit of 128.

CV0008 **executable file format error**

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 to see if it is a valid executable file.

This error can be caused if there is not enough memory available to run the program.

Try making more memory available to the program.

CV0012 out of memory

CodeView was unable to allocate or reallocate the memory that it required because not enough memory was available.

Possible solutions include the following:

- 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.

CV0013 access denied

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.

CV0014 invalid address

The command expected an address but was given 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.

CV0018 no such file or directory

The specified file does not exist or a path 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 Source (VS) 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.

CV0022 invalid argument

An invalid value was given as an argument.

CV0024 **too many open files**

CodeView could not open a file it needed because a file handle was not available.

Increase the number of file handles by changing the FILES setting in CONFIG.SYS to allow a larger number of open files. FILES=50 is the recommended setting.

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.

Additional files can be made available by closing source windows. If more files are needed, set helpbuffers=0 in the [pwb] section of TOOLS.INI. As a result, online help cannot be used but several file handles will be made available.

CV0028 **no space left on device**

The disk does not have any space available for writing.

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.

CV0101 **no CodeView information for** *filename*

The executable file or dynamic-link library (DLL) 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.

CV0102 **unpacked CodeView information in** *filename***: use CVPACK**

For this version of CodeView, you must process all executable files using CVPACK, which compresses the debugging information in the file.

Pass the file through CVPACK.EXE before starting CodeView.

CV0103 **relink** *filename* **with the current linker**

This version of CodeView expects the executable file to be in the format produced by the current version of the linker.

Make sure PWB, NMAKE, or the compiler is not running an older version of the linker.

CV0104 CodeView information for *filename* is newer than this version of CodeView

The executable file was compiled or linked with a version of a Microsoft compiler that is newer than the version of CodeView you are using.

Try one of the following:

- Reinstall CodeView that came with the new compiler.
- Remove older versions of CodeView that may be present on your hard disk.
- Recompile the program with an older version of a Microsoft compiler.

CV1001 invalid breakpoint command

CodeView could not interpret the breakpoint command.

The command probably used an invalid symbol or the incorrect command format.

CV1003 extra input ignored

The first part of the command line was interpreted correctly.

The remainder of the line could not be interpreted or was unnecessary.

CV1004 invalid register

The Register (**R**) command named a register that does not exist or cannot be displayed. CodeView can access the following registers: AX, SP, DS, IP, BX, BP, ES, FL, CX, SI, SS, DX, DI, CS.

When running with MS-DOS or the Windows operating system on an 80386 or an 80486 machine, the 80386 registers option can be selected to access the following registers: EAX, ESP, DS, GS, EBX, EBP, ES, SS, ECX, ESI, FS, EIP, EDX, EDI, CS, EFL.

When debugging p-code, CodeView can also access the following registers: TL, TH, PQ.

CV1006 breakpoint number or '*' expected

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 rika
```

CV1007 unable to open file

The specified file 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.

CV1011 no previous regular expression

The Repeat Last Find command was executed, but a regular expression (search string) was not previously specified.

CV1012 regular expression too long

The regular expression was too long or complex.

Use a simpler or more general regular expression.

CV1016 match not found

A string could not be found that matched the search pattern.

CV1017 syntax error

The command contained a syntax error.

This error is probably caused by an invalid command or expression.

CV1018 unknown symbol

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 turned on. Case sensitivity is toggled by the Case Sensitivity command from the Options menu or is 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.
- A search was made for an undefined label or function.

CV1021 unknown format specifier; specify one of A,B,I,IU,IX,L,LU,LX,R,RL,RT

An unknown format specifier was given to a View Memory (**VM**), Memory Dump (**MD**), or Memory Enter (**ME**) command.

The valid format specifiers are:

Specifier	Display Format
A	ASCII
B	byte
I	16-bit signed decimal integer
IU	16-bit unsigned decimal integer
IX	16-bit hexadecimal integer
L	32-bit signed decimal integer
LU	32-bit unsigned decimal integer
LX	32-bit hexadecimal integer
R	32-bit single precision floating point
RL	64-bit double precision floating point
RT	80-bit 10-byte real (long double)

This error is probably due to a mistyped command.

CV1022 invalid flag

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.

CV1023 no code at this line number

A line number was specified but code was not generated for that line. This error can be caused by a blank line, comment line, line with program declarations, or 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 off optimization.

Note that in a multiline statement the code is associated only with one line of the statement.

This error can be caused by debugging a program whose source has been modified after it was compiled. Recompile the file before running it through CodeView.

CV1027 invalid radix: specify 8, 10, or 16

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.

CV1031 no source lines at this address

An attempt was made to view an address that does not have source code.

This error can be caused by debugging a program whose source has been modified after it was compiled. Recompile the file before debugging it with CodeView.

CV1039 not a text file

An attempt was made to load a file that is not a text file. The file may be binary data.

This error can also occur if the first line of a file includes characters that are in the range of ASCII 0 to 8, 14 to 31, or 127 (0x0 to 0x8, 0xE to 0x1F, or 0x7F).

The Source window can only be used to view text files.

CV1040 video mode changed without /S option

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.

CV1041 file error

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.

CV1042 library module not loaded

The program being debugged uses load-on-demand dynamic-link libraries (DLLs). At least one of these libraries is needed but could not be found.

CV1043 application output lost; screen exchange is off

The program being debugged wrote to the display when the Flip (/F) or Swap (/S) option was turned off. The program output was lost.

When flipping is on, video page 1 is usually reserved for CodeView. Programs usually write to video page 0 by default. Programs that write to video page 1 must be debugged with swapping on.

Turn Flip or Swap on to be able to view program output.

CV1046 invalid executable file: relink

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.

CV1047 overlay not resident

An attempt was made to access machine code from an overlay section of code that is not currently resident in memory.

Execute the program until the overlay is loaded.

CV1048 floating-point support not loaded

An attempt was made to access the math processor registers in a program that does not use floating-point arithmetic.

One of the following can cause this error:

- Math processor registers can only be accessed through the floating-point library code. If the program does not perform floating-point calculations, this error can occur because the floating-point library code will not be loaded and cannot be used to access math processor registers.
- If the program does not use floating-point instructions, this error can occur when you attempt to access the math processor before any floating-point instructions have been performed. The 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 high-level language code that calls the routine, this error occurs.

CV1050 expression not a memory address

The expression does not evaluate to an address.

For example, `buffer[count]` is a valid address because it points to a specific memory location. The logical comparison `zed != 0` is not a valid address because it evaluates to TRUE or FALSE, not a memory address.

CV1051 missing or corrupt emulator information

Status information about the floating-point emulator is missing or corrupt.

The program probably wrote to this area of memory. Make sure the pointer points to its intended object.

CV1053 TOOLS.INI not found

The directory listed in the INIT environment variable did not contain a TOOLS.INI file.

Check the INIT variable to be sure that it points to the correct directory.

CV1054 cannot read this version of CURRENT.STS

The state file (CURRENT.STS) has a version number that is not recognized by this version of CodeView.

The old CURRENT.STS was ignored, and a new one will be created when CodeView exits.

CV1056 cannot understand entry in *filename*

At least one line in the given file (either the state file or the TOOLS.INI file) could not be interpreted.

On startup, 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.

CV1057 CURRENT.STS not found; creating

Since the state file (CURRENT.STS) could not be located at startup, CodeView created a state file.

CV1058 no source window open

A command was entered to manipulate the contents of a Source window, but a Source window was not open.

CV1059 no CodeView source information

CodeView symbol listing for the source file or module being debugged does not exist.

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.

CV1060 command not supported for current configuration

If you have specified the two monitors option (/2), you cannot specify the flip/swap option (/of- or /of+) from the CodeView Command Window.

CV1061 no second monitor connected to system

CodeView was invoked with the /2 option, but there was only one monitor for CodeView to use.

CV1062 invalid code-segment context change

An attempt was made to set the IP register to a line or address in a different segment.

CV1063 cannot create CURRENT.STS

CodeView could not find an existing state file (CURRENT.STS), and CodeView 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. Increase the number of file handles by changing the FILES setting in CONFIG.SYS to allow a larger number of open files.
- The environment variable INIT pointed to a directory that does not exist.

CV1064 window could not be opened

CodeView tried to open a window, but failed to do so.

This error is probably caused by a lack of memory available to CodeView.

Exit CodeView and make more memory available, then restart CodeView.

CV1065 cannot load expression evaluator *filename*

CodeView could not load the specified expression evaluator.

Make sure that *filename* is a valid expression evaluator DLL. If not, try reinstalling the CodeView DLLs from the distribution disks.

CV1066 cannot load expression evaluator *filename***; limit is 10**

Up to 10 expression evaluators can be specified in the TOOLS.INI file.

Try removing expression evaluators you won't be using in your debugging session.

CV1067 extension missing for Expression Evaluator: *filename* **in TOOLS.INI**

The Eval entry in the TOOLS.INI file expected a list of filename extensions.

CV1068 breakpoint specifier is out of range

The breakpoint number specified was higher than the number of current breakpoints.

CV1250 **general expression-evaluator error**

An error occurred in a CodeView expression evaluator.

This error is probably caused by a lack of memory available to the expression evaluator. You can free memory by doing one or more of the following:

- Close windows that are not needed. The Memory window should be closed if possible.
- Delete breakpoints that are not needed.
- Disable options that are not needed.

As a last resort, exit CodeView and start the debugging session again.

This error can also be caused by an expression that cannot be evaluated by the expression evaluator.

CV1251 *message*

An error occurred within a CodeView expression evaluator.

No further explanation is available.

CV1254 **invalid address expression**

The expression entered does not evaluate to an address.

The expression must be in a form that can appear on the left side of an assignment and refer to a memory location.

For example, `buffer[count]` is a valid l-value because it points to a specific memory location. The logical comparison `zed != 0` is not a valid l-value because it evaluates to TRUE or FALSE, not a memory address.

CV1255 **no data members**

The class, structure, or union that was expanded did not have data members. A class must contain at least one data member to be expanded.

CV2206 **corrupt CodeView information in** *filename***; discarding**

This error can be caused by using mismatched versions of development tools. Verify that the versions of all tools are current and synchronized.

Try recompiling the file with the /Zd switch (Prepare for Debugging option).

This option produces an object file containing only public symbols (global or external) and line numbers.

CV2207 **loaded symbols for** *module*

CodeView automatically loaded the symbols for the given dynamic-link library (DLL). The DLL can now be debugged.

This message is for your information only and does not indicate an error.

CV2209 **cannot restart; current process is not the process being debugged**

The debugging session was halted, and a different process was started.

Return to the debugged program's process by setting a breakpoint in it and issue a Go command.

CV2210 **invalid tab setting; using 8**

The value for tabs cannot be less than 0 or greater than 19. If you supply a value that is not in this range, the default tab value is 8.

CV2211 **cannot terminate; current process is not the process being debugged**

The debugging session was halted, and a different process was current.

Return to the debugged program's process by setting a breakpoint in it and issue a Go command.

CV2401 **missing argument for** *option* **option**

This error can be caused by splitting a response file line naming a program to be debugged and its command-line options. The program name and its command-line option must be on one line.

CV2402 **unknown option** *option***; ignored**

The specified option was not a valid option.

Check that the option was typed correctly.

CV2403 **response files cannot be nested**

A response file cannot refer to another response file.

CV2404 **cannot open response file:** *filename*

The specified response file could not be opened.

Check that the name of the file is spelled correctly and that the response file is correct.

CV2405 **command line option,** *option***, invalid for target operating system**

The specified command line option was illegal in this context.

CV2406 **command-line is too big. arguments truncated**

The command line argument in a response file was longer than the limit of 256 bytes.

CV3608 **out of memory**

CodeView needed additional memory, but insufficient memory was available.

Possible solutions include the following:

- Remove some drivers or applications that have been loaded in high memory.
- 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 high memory.
- Free some memory by removing terminate-and-stay-resident (TSR) software.
- Remove unneeded watch expressions or breakpoints.

CV3620 **bad DLL format in** *filename*

CodeView did not recognize the format of the specified CodeView dynamic-link library (DLL) file.

The DLL may be damaged or may be the wrong version.

This error is caused if the specified file is not a DLL.

CV3621 **cannot find DLL** *filename*

CodeView could not find the specified dynamic-link library (DLL). This may be caused by a mistyped filename in the TOOLS.INI file.

CV3622 **cannot load DLL** *filename*

CodeView was unable to load the specified dynamic-link library (DLL) file.

Reinstall the CodeView DLL from the distribution disks.

CV3623 **wrong DLL** *filename*

CodeView expected one type of dynamic-link library (DLL) but read a different type. This error is probably caused by specifying an incorrect filename in the TOOLS.INI file. For example, you may have specified an execution model in the expression-evaluator entry.

CV3624 **cannot load execution model** *filename* **- limit is 1**

Too many execution models are specified in the TOOLS.INI file.

Only one execution model can be used at a time.

Remove those execution models you are not using in your debugging session.

CV3625 **no transport layer; exiting**

CodeView needs a transport layer to make appropriate calls to the operating system in local debugging and to a remote computer in remote debugging.

Check your TOOLS.INI file, and make sure there is a Transport entry in the [cv] or [cvw] CodeView section.

CV3626 **no execution model; exiting**

CodeView needs an execution model in order to function.

Check your TOOLS.INI file, and make sure there is a Native entry specified.

CV3627 **no nonnative execution models found**

You must specify a nonnative execution model in order to debug a p-code program.

Add the following line to your TOOLS.INI file:

```
model:nmd1pcd.dll
```

CV3628 **too many transport layers: choose one**

Only one transport layer can be selected at one time.

CV3629 **too many execution models: choose one**

Only one execution model can be selected at a time.

Additional execution models should be removed.

CV3630 **no symbol handler found; exiting**

A symbol handler dynamic-link library (DLL) could not be found. The DLLs that CodeView uses must be in a location specified by the cvdllpath entry in the [cvw] or [cv] section of TOOLS.INI.

CV3631 **program being debugged contains p-code, but no *model*: specified in tools.ini**

The entry `model=nmd1pcd.dll` must be specified in TOOLS.INI to debug a program that contains p-code.

CV4000 **assembler: not enough operands**

Additional operands are required for this instruction.

The instruction was rejected and the address was not advanced.

CV4001 **assembler: too many operands**

Too many operands were specified for the most recently issued instruction.

The instruction was rejected and the address was not advanced.

CV4002 **assembler: incorrect operand size**

An instruction required an operand of a different size.

The instruction was rejected and the address was not advanced.

CV4003 **assembler: illegal range**

The size of a specified value exceeds the size expected by the instruction.

The instruction was rejected and the address was not advanced.

CV4004 **assembler: overflow**

Numeric overflow occurred while assembling the current instruction.

The instruction was rejected and the address was not advanced.

CV4005 **assembler: syntax error**

The syntax for the instruction is incorrect.

The instruction was rejected and the address was not advanced.

CV4006 **assembler: unknown opcode**

An instruction was not recognized.

Check that the instruction was typed correctly.

The instruction was rejected and the address was not advanced.

CV4007 **assembler: extra characters**

The instruction contained extra characters that could not be recognized. The instruction may have been mistyped.

The line was ignored.

The instruction was rejected and the address was not advanced.

CV4008 **assembler: illegal operand**

The wrong type of operand was used for this context.

The instruction may have been mistyped.

The instruction was rejected and the address was not advanced.

CV4009 **assembler: illegal segment**

An invalid segment was used.

The instruction was rejected and the address was not advanced.

CV4010 **assembler: illegal register**

An illegal or nonexistent register was accessed.

The register name may have been mistyped.

This error can be caused by trying to access 80386- or 80486-specific registers when CodeView is running on an 8088- or 80286-based machine.

The instruction was rejected and the address was not advanced.

CV4011 assembler: divide by zero

CodeView encountered a divide-by-zero error while assembling the current instruction.

The instruction was rejected and the address was not advanced.

CV4012 cannot assemble code with current execution model

This error can be caused by trying to assemble p-code in CodeView.

CV4500 bad fixed format length: using variable length

In invalid length was specified for the Memory window. CodeView will set the length based on the current window width.

Try specifying a different length.

CV4501 invalid window id

The window ID was invalid. It must be either 0 or 1.

CV4502 unable to open the requested memory window

CodeView could not open a Memory window.

The only valid window IDs are 0 and 1. You may need to close some windows.

CV5001 cannot select

The cursor was not on the same line as an automatically selectable symbol.

CV5004 cannot read file

CodeView could not read a file.

Read the file again. If the second read fails, exit and restart CodeView. If the read process still fails, the file may be corrupt.

CV5005 no file selected

A module must be selected before OK is chosen.

To exit the dialog box without selecting a module, choose Cancel.

CV5009 no watch expression to delete

An attempt was made to delete one or more watch variables (watch expressions), but watch expressions are not currently selected.

CV5012 packed executable file

CodeView cannot step through the beginning of files that are linked with the /EXEPACK option. There are two solutions to this problem:

- Relink without this option to debug the file and then switch back to linking with /EXEPACK for the release version of your program.
- Execute the program through startup code, and set breakpoints only after the program has entered main.

CV5013 no expression evaluators found; exiting

CodeView needs at least one expression evaluator in order to operate.

Check the [cv] or [cvw] section of your TOOLS.INI file and specify at least one Eval entry.

CV5014 cannot execute function in watch expression

A watch expression cannot specify a function to be executed.

CVPACK Error Messages

Microsoft Debugging Information Compactor (CVPACK) generates the following error messages:

- Fatal errors (CK1*xxx*) cause CVPACK to stop execution.
- Warnings (CK4*xxx*) indicate possible problems in the packing process.

CVPACK Fatal Error Messages

CK1000 unknown error; contact Microsoft Product Support Services

CVPACK detected an unknown error condition.

Note the circumstances of the error and notify Microsoft Corporation by following the instructions in the “Microsoft Support Services” section of the introduction to this book.

CK1001 out of memory

The executable file is too big for the available memory. This error can occur with MS-DOS when there is little extra memory. Even though CVPACK uses virtual memory, which involves swapping to disk, some information can be stored only in real memory.

One of the following may be a solution:

- Assemble and link in separate steps (that is, use NMAKE).
- Recompile one or more of the object files without debugging information. If the file was compiled using the /Zi option, use either /Zd or no option.
- Add more memory to your computer.

CK1002 out of virtual memory

There was not enough virtual memory for CVPACK to pack the executable file. Virtual memory can be any of the following:

- Conventional memory. Remove TSR (terminate-and-stay-resident) programs or run CVPACK outside of a shell or a makefile.
- Extended or expanded memory. Run CVPACK under a DPMI server, or as an MS-DOS session within the Windows operating system (386 Enhanced Mode).
- Disk space. Free some disk storage.

CK1003 cannot open file

CVPACK could not open the specified executable file.

One of the following may be a cause:

- The specified file does not exist. Check the spelling of the filename and path.
- The executable file was opened or deleted by another process.

CK1004 file is read-only

CVPACK cannot pack a read-only file. Change the read attribute on the executable file and run CVPACK again.

CK1005 invalid executable file

CVPACK could not process the executable file. One of the following may be a cause:

- The debugging information in the executable file is corrupt.
- The executable file is a zero-length file.

CK1006 invalid module *module*

The given object file did not have a valid format.

Check the linker version.

CK1007 **invalid** *table* **table in module** *module*

The given table in the given object file was not valid.

Check the compiler and linker versions.

CK1008 **cannot write packed information**

There was not enough space on disk for CVPACK to write the packed executable file. This leaves a corrupt file on disk.

Make more space available on disk and relink the program.

CK1009 **module** *module* **unknown type index** *number***;**
contact Microsoft Product Support Services

The debugging information in the executable file is corrupt. This is due to an internal error in either the compiler or CVPACK. Recompile the program. If the problem persists, note the circumstances of the error and notify Microsoft Corporation by following the instructions in the "Microsoft Support Services" section of the introduction to this book.

CK1010 **symbol error in module** *module***;**
contact Microsoft Product Support Services

The debugging information in the executable file is corrupt. This is due to an internal error in either the compiler or CVPACK. Recompile the program. If the problem persists, note the circumstances of the error and notify Microsoft Corporation by following the instructions in the "Microsoft Support Services" section of the introduction to this book.

CK1011 **error in type** *number* **for module** *module***;**
contact Microsoft Product Support Services

The debugging information in the executable file is corrupt. This is due to an internal error in either the compiler or CVPACK. Recompile the program. If the problem persists, note the circumstances of the error and notify Microsoft Corporation by following the instructions in the "Microsoft Support Services" section of the introduction to this book.

CK1012 **no Symbol and Type Information**

The executable file does not contain debugging information.

Link the program using the /CO option to put at least minimal debugging information in the executable file. To include full debugging information in an object file, compile or assemble using the /Zi option. To include minimal information and line numbers, compile or assemble using the /Zd option.

CK1013 debugging information missing or unknown format

One of the following has occurred:

- The program did not contain debugging information. Recompile using /Zi or /Zd, then link using /CO.
- The executable file was linked using an obsolete or unsupported linker. Use Microsoft LINK version 5.3x or later.
- The executable file was already packed using a previous version of CVPACK.

CK1014 module *module* **type** *number* **refers to skipped type index; contact Microsoft Product Support Services**

The debugging information in the executable file is corrupt. This is due to an internal error in the compiler. Recompile the program. If the problem persists, note the circumstances of the error and notify Microsoft Corporation by following the instructions in the "Microsoft Support Services" section of the introduction to this book.

CK1015 too many segments in module *module*

The **alloc_text** pragma was used more than 20 times in an object file that was compiled with Microsoft C version 6.x or earlier.

One of the following may be a solution:

- Recompile using Microsoft C/C++ version 7.0 or later.
- Split the object file into multiple files.
- Group the pragma statements according to segment.

CK1016 unable to execute MPC for CVPACK /PCODE

CVPACK could not find MPC.EXE on the path.

CK1017 precompiled types file *filename* **not found**

The program used a precompiled header, but the program was linked without the object file that was created when the header was precompiled.

CK1018 precompiled types object file *filename* **inconsistent with precompiled header used to compile object file** *filename*

The program used a precompiled header, but the object file linked to the program was not the object file that was created when the header was precompiled. Either the user or the creator changed since the last compilation.

Recompile and relink. If a makefile is used, check the makefile dependencies.

CK1020 **packed type index exceeds 65535 in module** *module*

The debugging information exceeded a CVPACK limit.

This error may occur when precompiled headers are used.

One of the following may be a solution:

- Eliminate unused type strings.
- Compile some object files without debugging information.

CK1021 **error in precompiled types signature in module** *module*

The program was compiled with an out-of-date precompiled header.

Delete the object file and recompile.

CK 1022 **Symbol table for** *file* **is too large**

The corrective action is to compile *file* without CodeView information, reduce the number of symbols in the file, or split the file into two or more pieces.

CVPACK Warning Messages

CK4000 **unknown warning; contact Microsoft Product Support Services**

CVPACK detected an unknown error condition.

Note the circumstances of the error and notify Microsoft Corporation by following the instructions in the “Microsoft Support Services” section of the introduction to this book.

CK4001 **file already packed**

CVPACK took no action because the executable file has already been processed by CVPACK 4.00.

CK4002 **duplicate public symbol** *symbol* **in module** *module*

The given symbol was redefined in the given module. CVPACK deleted the second occurrence of the symbol.

Probably an earlier version of the linker was used. Use LINK 5.30 or later.

CK4003 **error in lexical scopes for module** *module***, symbols deleted**

The scoping of symbols in the given object module was corrupted. CVPACK deleted the symbols in the module.

This is probably a compiler error. Recompile and relink the object file.

EXEHDR Error Messages

This section includes error messages generated by the Microsoft EXE File Header Utility (EXEHDR). EXEHDR errors (U1100 through U1140) are always fatal.

EXEHDR Fatal Error Messages

U1110 **malformed number** *number*

A command-line option for EXEHDR required a value, but the given 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**

The given command-line option for EXEHDR required a value, but nothing was specified. EXEHDR ignored the option.

U1115 **option** *option* **ignored**

The given 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 recognized as an 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.

Math Coprocessor Error Messages

The error messages listed below correspond to exceptions generated by the math coprocessor hardware. Refer to the manufacturer's documentation for your processor for a detailed discussion of hardware exceptions. These errors may also be detected by the floating-point emulator or alternate math library.

M6101 invalid

An invalid operation occurred. This error usually occurs when the operand is NAN (not a number) or infinity.

This error terminates the program with exit code 129.

M6102 denormal

A very small floating-point number was generated, which may no longer be valid because of a loss of significance. Denormal floating-point exceptions are usually masked, causing them to be trapped and operated upon.

This error terminates the program with exit code 130.

M6103 divide by 0

A floating-point operation attempted to divide by zero.

This error terminates the program with exit code 131.

M6104 overflow

An overflow occurred in a floating-point operation.

This error terminates the program with exit code 132.

M6105 underflow

An underflow occurred in a floating-point operation. Underflow floating-point exceptions are usually masked, causing the underflowing value to be replaced by 0.0.

This error terminates the program with exit code 133.

M6106 inexact

Loss of precision occurred in a floating-point operation. This exception is usually masked. Many floating-point operations cause a loss of precision.

This error terminates the program with exit code 134.

M6107 **unemulated**

An attempt was made to execute a coprocessor instruction that is invalid or is not supported by the emulator.

This error terminates the program with exit code 135.

M6108 **square root**

The operand in a square-root operation was negative.

This error terminates the program with exit code 136.

The **sqrt** function in the C run-time library and the FORTRAN intrinsic function **SQRT** do not generate this error. The C **sqrt** function checks the argument before performing the operation and returns an error value if the operand is negative. The FORTRAN **SQRT** function generates the DOMAIN error M6201 instead of this error.

M6110 **stack overflow**

A floating-point expression caused a stack overflow on the 8087/80287/80387 coprocessor or the emulator.

Stack-overflow floating-point exceptions are trapped up to a limit of seven levels in addition to the eight levels usually supported by the 8087/80287/80387 coprocessor.

This error terminates the program with exit code 138.

M6111 **stack underflow**

A floating-point operation resulted in a stack underflow on the 8087/80287/80387 coprocessor or the emulator.

This error terminates the program with exit code 139.

M6201 *function* **: _DOMAIN error**

An argument to the given function was outside the domain of legal input values for that function.

M6202 *function* **: _SING error**

An argument to the given function was a singularity value for this function. The function is not defined for that argument.

For example, in FORTRAN the following statement generates this error:

```
result = LOG10(0.0)
```

This error calls the **_matherr** function with the function name, its arguments, and the error type. You can rewrite the **_matherr** function to customize the handling of certain run-time floating-point math errors.

M6203 *function* **: _OVERFLOW error**

The given function result was too large to be represented.

This error calls the **_matherr** function with the function name, its arguments, and the error type. You can rewrite the **_matherr** function to customize the handling of certain run-time floating-point math errors.

M6205 *function* **: _TLOSS error**

A total loss of significance (precision) occurred.

This error may be caused by giving a very large number as the operand of **sin**, **cos**, or **tan** because the operand must be reduced to a number between 0 and 2π.

H2INC Error Messages

H2INC Fatal Errors

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 MS-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.

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 MS-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 MS-DOS, remove other programs or drivers running in the system which could be consuming significant amounts of memory.
- In MS-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.

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 This error existed in previous versions of H2INC as "unknown option *character* in *option*." This condition now generates warning HI4799.

HI1049 This error existed in previous versions of H2INC as "invalid numerical argument *string*." This condition now generates warning HI4052.

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 This error existed in previous versions of H2INC as "*option*: unrecognized option." This condition now generates warning HI4799.

HI1801 **incomplete model specification**

Only part of a custom memory-model specification was specified on the command line.

When you specify a custom memory model with the /A command-line option, you must specify code pointer distance, data pointer distance, and DS register setup. This error is equivalent to the D2013 error message for CL.

H2INC Nonfatal Errors

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 in the "Microsoft Support Services" section of the introduction to this book.

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 file-name 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, or 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 must 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.

H2INC Warnings

HI4000 (level 1) **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 in the “Microsoft Support Services” section of the introduction to this book.

HI4001 (level 1, 4) **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 (level 1) **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 (level 1) **not enough actual parameters for macro** *identifier*

The number of actual arguments specified with the given identifier was less 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 (level 1) **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 (level 1) *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 (level 1) **#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 (level 2) *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 (level 2) *identifier* **: _fastcall attribute on data ignored**

The **_fastcall** attribute on the given data identifier was ignored.

HI4009 (level 1) **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.

HI4010 (level 1) **identifier is a MASM keyword**

This warning is issued if the .h include file tries to redefine a MASM keyword.

H2INC will give a warning whenever such conflicts take place. This includes **#define**, **typedef**, structures, and other variables. If you want to redefine a MASM keyword, use **#define** instead. A **#define** in the .INC file will not try to redefine the MASM keyword unless the /Ht option is set.

This warning will also be issued anytime converting a **typedef** statement will result in a type with the same name as the type. The translation is not done in this case. For more information on warning HI4010, see "Miscellaneous Utilities."

HI4011 (level 1) **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 (level 1) *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 (level 3) *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 (level 1) **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 4-byte value.

HI4020 (level 1) *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 (level 1) *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 (level 1) *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 (level 1) *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 (level 1) *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 (level 1) **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 (level 1) **first parameter list longer than the second**

A function was declared more than once with different parameter lists.

The first declaration was used.

HI4031 (level 1) **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 (level 1) **sizeof returns 0**

The **sizeof** operator was applied to an operand that yielded a size of zero.

This warning is informational.

HI4040 (level 1) **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 (level 1) *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 the variable was declared with no storage class.

HI4044 (level 1) **_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 (level 1) *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 array or pointer), 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 (level 1) **array's declared subscripts different**

An expression involved pointers to arrays of different size.

The pointers were used without conversion.

HI4049 (level 1) *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 (level 4) *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 (level 2) **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.

HI4052 **invalid numerical argument** *string*

A numerical argument was expected instead of the given string.

HI4053 (level 1) **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 (level 2) *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.

HI4066 (level 2) **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 (level 1) **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 (level 2) *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 (level 1) *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 (level 1) **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 (level 1) *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 (level 1) **unexpected token** *token*

An unexpected separator token was found in the argument list of a pragma.

The remainder of the pragma was ignored.

HI4082 (level 1) **expected an identifier, found** *token*

An identifier was missing from the argument list.

The remainder of the pragma was ignored.

HI4083 (level 1) **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 (level 1) **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 (level 1) **expected [on | off]**

The pragma expected an on or off parameter, but the specified parameter was unrecognized or missing.

The pragma was ignored.

HI4086 (level 1) **expected [1 | 2 | 4]**

The pragma expected a parameter of either 1, 2, or 4, but the specified parameter was unrecognized or missing.

HI4087 (level 1) *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 (level 1) *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 (level 1) *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 (level 1) **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 (level 2) **no symbols were declared**

The compiler detected an empty declaration, as in the following example:

```
int ;
```

The declaration was ignored.

HI4092 (level 2) **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 (level 3) **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 (level 1) **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 (level 2) *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 (level 1) **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 (level 1) *identifier* **: near data in same_seg pragma, ignored**

The given near variable was specified in a **same_seg** pragma.

The *identifier* was ignored.

HI4105 (level 1) *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 (level 1) **unexpected identifier** *identifier*

The pragma contained an unexpected token.

The pragma was ignored.

HI4110 (level 1) **unexpected token** *int constant*

The pragma contained an unexpected integer constant.

The pragma was ignored.

HI4111 (level 1) **unexpected token** *string*

The pragma contained an unexpected string.

The pragma was ignored.

HI4112 (level 1) **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 (level 1) **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 (level 1) **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 (level 1) *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 (level 1) **(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 (level 1) **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 (level 1) **_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 (level 1) **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 (level 4) **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 (level 1) *flag* **: unknown memory model flag**

The flag used with the /A option was not recognized and was ignored.

HI4128 (level 4) **storage-class specifier after type**

A storage-class specifier (auto, extern, register, static) appears after a type in a declaration. The compiler assumed the storage class specifier occurred before the type.

New-style code places the storage-class specifier first.

HI4129 (level 4) *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 (level 4) *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 (level 4) *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 (level 4) *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 (level 3) **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 (level 4) **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 (level 1) ***/ 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 (level 1) *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 (level 1) **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 (level 1) **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 (level 3) **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 (level 4) **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 (level 1) *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.

HI4204 (level 3) *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 (level 4) **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 (level 4) **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 (level 2) **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.

HI4301 (level 2) **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 (level 3) **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 (level 3) **potential mod by 0**

The second operand in a remainder operation evaluated to zero at compile time, giving undefined results.

HI4799 (level 1) **unknown option** *character* **in** *option*

A command line option was specified that was not understood by H2INC, or the given character was not a valid letter for the option.

For example, the following line:

```
#pragma optimize("q", on)
```

causes the following warning:

```
unknown option 'q' in '#pragma optimize'
```

HI4800 (level 1) **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 (level 1) **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 (level 1) **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 (level 2) **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 (level 1) **static function prototype not translated**

H2INC does not translate static items, as they are not visible outside the C source file.

HI4812 (level 1) **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 (level 1) *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 (level 1) **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 (level 1) **ignoring function definition :** *function***()**

H2INC translates header information only; it cannot convert program code. H2INC does not translate function bodies.

HELPMAKE Error Messages

Microsoft Help File Maintenance Utility (HELPMAKE) generates the following error messages:

- 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.

HELPMAKE Fatal Error Messages

H1000 **/A requires character**

The /A option requires an application-specific control character.

The correct form is:

```
/Ac
```

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:

```
/En
```

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:

```
/Sn
```

where *n* specifies the format of the input file. 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:

```
/Sn
```

where *n* specifies the format of the input file. 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 can 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:

```
/En
```

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 during 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. It may be that the Help file is not a concatenated file and contains only one database, and the database has the same name as its physical Help file.

One of the following may be a solution:

- Rename the Help file so that the filename does not match any of the database names.
- Run HELPMAKE from a directory other than the one that holds the physical Help file. Since HELPMAKE creates the split files in the current directory, no filename conflict occurs.

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 in the “Microsoft Support Services” section of the introduction to this book.

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 in the “Microsoft Support Services” section of the introduction to this book.

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 Support Services” section of the introduction to this book.

HELPMAKE Error Messages

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.

HELPMAKE Warning Messages

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.

IMPLIB Error Messages

Microsoft Import Library Manager (IMPLIB) generates the following error messages:

- Fatal errors (IM16*xx*) cause IMPLIB to stop execution.
- Errors (IM26*xx*) prevent IMPLIB from creating an import library.
- Warnings (IM46*xx*) indicate possible problems in the output file being created.

IMPLIB Fatal Error Messages

IM1600 error writing to output file—*message*

IMPLIB could not create the import library for the given reason.

Probably the drive or directory where the import library is being created is full.

IM1601 out of memory, near/far heap exhausted

There was not enough room in memory for the heap needed by IMPLIB.

Increase the available memory. Some ways to do this include:

- Remove TSR (terminate-and-stay-resident) programs.
- Run IMPLIB outside of an NMAKE session.
- Run IMPLIB outside of a shell.

IM1602 syntax error in module-definition file

IMPLIB could not understand the contents of a .DEF input file.

IM1603 *filename* **: cannot create file—***message*

IMPLIB could not create the given file for the given reason.

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—***message*

IMPLIB could not find the specified module-definition (.DEF) file or dynamic-link library (DLL) for the given reason.

IM1605 **too many nested include files in module-definition file**

A module-definition (.DEF) file contained an **INCLUDE** statement specifying a nested set of include files that exceeded the limit for nesting. The limit is 10 levels.

IM1606 **missing or invalid include file name**

A syntax error occurred in an **INCLUDE** statement in a module-definition (.DEF) file.

One of the following may have occurred:

- A filename was not specified.
- More than one filename was specified.
- A long filename was specified without being enclosed in quotation marks or was enclosed in one single and one double quotation mark.

IM1607 *extension* **: invalid extension for target library**

The given extension was specified for the import library.

An import library cannot be given a .DEF or .DLL extension.

IM1608 **no .DLL or .DEF source file specified**

No input file was specified on the IMPLIB command line.

IMPLIB Error Messages

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 dynamic-link library (DLL) specified to IMPLIB was corrupted.

IM2603 *filename* **: invalid .DLL file**

IMPLIB did not recognize the given input file as a dynamic-link library (DLL).

IMPLIB Warning Messages

IM4600 **line** *number* **too long; truncated to 512 characters**

The given line in the module-definition (.DEF) file exceeded the limit on line length. IMPLIB ignored text after the first 512 characters.

IM4601 **unrecognized option** *option***; option ignored**

The given option was not a valid IMPLIB option. IMPLIB used the rest of the command line to try to build an import library.

LIB Error Messages

This section lists error messages generated by the LIB utility.

Microsoft Library Manager (LIB) generates the following error messages:

- Fatal errors (U1150 through U1203) cause LIB to stop execution.
- Errors (U2152 through U2159) do not stop execution but prevent LIB from creating a library.
- Warnings (U4150 through U4158) indicate possible problems in the library being created.

LIB Fatal Error Messages

U1150 page size too small; use option /PAGE:n to increase it

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 newline character was expected in the command line but did not appear.

One cause of this error is 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 supplied to LIB did not end with a newline character.

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 attribute on the .BAK file.

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, or add more memory.

U1172 **no more virtual memory**

The LIB session required more memory than the 1-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 in the “Microsoft Support Services” section of the introduction to this book.

U1174 **mark : not allocated**

Note the circumstances of the error and notify Microsoft Corporation by following the instructions in the “Microsoft Support Services” section of the introduction to this book.

U1175 **free : not allocated**

Note the circumstances of the error and notify Microsoft Corporation by following the instructions in the “Microsoft Support Services” section of the introduction to this book.

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 in the “Microsoft Support Services” section of the introduction to this book.

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 in the “Microsoft Support Services” section of the introduction to this book.

U1190 **interrupted by user**

LIB was interrupted with either CTRL+C or CTRL+BREAK.

U1191 *libraryname* **: cannot write to read-only file**

Operations cannot be performed on the given library because it is marked as a read-only file.

Change the protection attribute on the library.

U1200 *filename* **: invalid library header**

The input library file had an invalid format.

Either it was not a library file or it was corrupted.

U1203 *filename* **: invalid object file near** *location*

The given file was not a valid object file or was corrupted at the given location.

LIB Error Messages

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 containing a a hyphen or 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 path, such as C:\MS-C. The SETUP program for a language calls LIB to create 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 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.

LIB Warning Messages

U4150 *module* **: module redefinition ignored**

A module was specified with the add 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 that is 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 added 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. However, 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.

LINK Error Messages

Microsoft Segmented-Executable Linker (LINK) generates the following error messages:

- Fatal errors (L1*xxx*) cause LINK to stop execution.
- Errors (L2*xxx*) do not stop execution but might prevent LINK from creating the main output file.
- Warnings (L4*xxx*) indicate possible problems in the output file being created.

LINK Fatal Error Messages

L1001 *option* **: option name ambiguous**

A unique option name did not appear after the option indicator.

An option is specified by a forward slash (/) and a name. The name can be specified by an abbreviation of the full name, but the abbreviation must be unambiguous.

For example, since many options begin with the letter **N**, the following command causes this error:

```
LINK /N main;
```

This error can also occur if the wrong version of the linker is being used. Check the directories in the PATH environment variable for other versions of LINK.EXE.

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 was specified with a LINK option.

For example, this error occurs if a nonnumeric string is specified with an option that requires a number.

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–2**

One of the following may have occurred:

- The given value specified with the /STACK option exceeded the limit of 65,534 bytes.
- A space appeared before or after the colon (**:**) between /STACK and the argument specified with it.

L1007 **/OVERLAYINTERRUPT : interrupt number exceeds 255**

An overlay interrupt number greater than 255 was specified with the /OV option value.

Check the *Microsoft MS-DOS Programmer's Reference* or other MS-DOS technical manual for information about interrupts.

L1008 **/SEGMENTS : segment limit set too high**

The value specified with the /SEG option exceeded 16,375.

L1009 *value* **: /CPARM : illegal value**

The value specified with the /CPARM option was not in the range 1–65,535.

L1020 **no object files specified**

The object-files field was empty.

LINK requires the name of at least one object file.

L1021 **cannot nest response files**

A response file was specified in a response file.

L1022 **response line too long**

A line in a response file was longer than 255 characters.

To extend a field to another line, put a plus sign (+) at the end of the current line.

L1023 **terminated by user**

The LINK session was halted by CTRL+C or CTRL+BREAK.

L1024 **nested right parentheses**

The parentheses for assigning overlays were specified incorrectly.

L1025 **nested left parentheses**

The parentheses for assigning overlays were specified incorrectly.

L1026 **unmatched right parenthesis**

The parentheses for assigning overlays were specified incorrectly.

L1027 **unmatched left parenthesis**

The parentheses for assigning overlays were specified incorrectly.

L1030 **missing internal name**

An **IMPORTS** statement specified an ordinal value but not an internal name for the routine or data item being imported.

An item imported by ordinal must be given an internal name.

L1031 **module description redefined**

The module-definition (.DEF) file contained more than one **DESCRIPTION** statement.

L1032 **module name redefined**

The module-definition (.DEF) file contained more than one **NAME** or **LIBRARY** statement.

L1033 **input line too long;** *number* **characters allowed**

The LINK command line cannot exceed the given number of characters.

L1034 **name truncated to** *string*

A name specified either on the LINK command line or in a module-definition (.DEF) file exceeded 255 characters. The name was truncated to the given string.

This is a warning, not a fatal error. However, it indicates a serious problem. This message may be followed by another error as LINK tries to use the specified name. For example, if the string is a filename, LINK issues an error when it cannot open the file.

L1035 **syntax error in module-definition file**

A statement in the module-definition (.DEF) file was incorrect.

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.

Replace long references with short references wherever 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 file contained more than 255 TYPDEF records.

TYPDEF records describe communal variables. (TYPDEF is an MS-DOS term. It is explained in the *Microsoft MS-DOS Programmer's Reference* and in other reference books on MS-DOS.)

This error appears only with programs created by the Microsoft FORTRAN Compiler or other compilers that support communal variables.

L1046 too many external symbols in one module

An object file specified more than 1023 external symbols.

Break the object file into smaller files.

L1047 too many group, segment, and class names in one module

An object file contained too many group, segment, and class names.

Reduce the number of groups, segments, or classes in the object file, or break the object file into smaller files.

L1048 too many segments in one module

An object file had more than 255 segments.

Either create fewer segments or break the object file into smaller files.

L1049 too many segments

The program contained more than the maximum number of segments.

The maximum number of segments is set with the /SEG option (in the range 1–16,384). If /SEG is not specified, the default is 128.

If this error occurs when linking a p-code program, recompile and use CL's /NQ option to combine the temporary p-code segments.

L1050 too many groups in one module

An object file contained more than 21 group definitions (GRPDEF).

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 MS-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 than could fit in available memory. Symbolic information includes public, external, segment, group, class, and file names.

One of the following may be a solution:

- Eliminate as many public symbols as possible.
- Combine object files or segments.
- Link from the command line instead of from a makefile or PWB.
- Remove terminate-and-stay-resident programs or otherwise free some memory.

L1054 requested segment limit too high

LINK did not have enough memory to allocate tables describing the requested number of segments. The number of segments is the value specified with the /SEG option or the default of 128.

One of the following may be a solution:

- Assemble with /c and link in a separate step.
- Link again using the /SEG option to set fewer segments.
- Remove terminate-and-stay-resident programs or otherwise free some memory.

L1056 too many overlays

The program defined more than 127 overlays.

L1057 data record too large

An LEDATA record in an object module contained more than 1024 bytes of data. This is a translator error. (LEDATA is an MS-DOS term explained in the *Microsoft MS-DOS Programmer's Reference* and in other MS-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 in the "Microsoft Support Services" section of the introduction to this book.

L1063 out of memory for debugging information

LINK ran out of memory for processing debugging information.

Reduce the amount of debugging information by compiling some object files with /Zd instead of /Zi or with neither option.

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:

- Reduce the size of code, data, and symbols in the program.
- If the program is a segmented executable file, put some code into a dynamic-link library.

L1065 **too many interoverlay calls**
use /DYNAMIC:*nnn***; current limit is** *number*

The program had more than the given limit of interoverlay calls.

The maximum number of interoverlay calls is set with the /DYNAMIC option (in the range 1–10,922). If /DYNAMIC is not specified, the default is 256.

To determine the setting needed by the program, run LINK with the /INFO option. The output gives the number of interoverlay calls that are generated and the current limit.

L1066 **size of** *overlaynumber* **overlay exceeds 64K**

The overlay represented by the given number exceeded the MOVE size limit of 65,535 bytes.

L1067 **memory allocation error**

Note the circumstances of the error and notify Microsoft Corporation by following the instructions in the “Microsoft Support Services” section of the introduction to this book.

L1070 *segment* **: segment size exceeds 64K**

A single segment contained more than 65,536 bytes 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 **__based** keyword (or its predecessor, the **alloc_text** pragma) to build smaller segments.

L1071 **segment _TEXT exceeds 64K–16**

The segment named _TEXT grew larger than 65,520 bytes. 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 LINK /DOSSEG option.

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 **__based** keyword (or its predecessor, the **alloc_text** pragma) 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 optimizing 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 the Windows operating system 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.

The table contains an entry for each exported routine and for each address that is the target of a far relocation, when **PROTMODE** is not enabled and the target segment is designated **MOVABLE**.

Declare **PROTMODE** if applicable, reduce the number of exported routines, or make some segments **FIXED** 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 or the root directory was full.

Delete or move files to make space.

L1082 *filename* **: stub file not found**

LINK could not open the file given in the **STUB** statement in the module-definition (.DEF) 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—*message*

LINK could not open a temporary file for the given reason.

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 temporary file missing

An internal error has occurred.

Note the circumstances of the error and notify Microsoft Corporation by following the instructions in the “Microsoft Support Services” section of the introduction to this book.

L1087 unexpected end-of-file on temporary file

A problem occurred with the temporary linker-output file.

One of the following may have occurred:

- The disk that holds the temporary file was removed.
- The disk or directory specified in the TMP environment variable was full.

L1088 out of space for list file

The disk or the root directory was full.

Delete or move files to make space.

L1089 *filename* **: cannot open response file**

LINK could not find the given response file.

One of the following may have occurred:

- The response file does not exist.
- The name of the response file was incorrectly specified.
- An old version of LINK was used. Check your path. To see the version number of LINK, run LINK with the /? option.

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 find the specified module-definition (.DEF) file.

Check that the name of the .DEF file is spelled correctly.

L1093 *filename* **: object file not found**

LINK could not find the given object file.

Check that the name of the object file is spelled correctly.

L1094 *filename* **: cannot open file for writing**

LINK was unable to open the given file with write permission.

Check the attributes for the file.

L1095 *filename* **: out of space for file**

LINK ran out of disk space for the specified output file.

Delete or move files to make space.

L1096 **unexpected end-of-file in response file**

LINK encountered a problem while reading the response file.

One of the following may be a cause:

- The response file is corrupt.
- The file was deleted between reads.

L1097 **I/O error—***message*

LINK encountered the given input or output error.

L1098 **cannot open include file** *filename—message*

LINK could not open the given include file for the given reason.

An include file is specified in an **INCLUDE** statement in the module-definition (.DEF) file.

L1100 **stub .EXE file invalid**

The file specified in the **STUB** statement in the module-definition (.DEF) file is not a valid MS-DOS executable file.

L1101 **invalid object module**

LINK could not link one of the object files.

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 in the "Microsoft Support Services" section of the introduction to this book.

L1102 **unexpected end-of-file**

The given library or object file had an invalid format.

L1103 **attempt to access data outside segment bounds**

A data record in an object file 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 the error to Microsoft Corporation by following the instructions in the "Microsoft Support Services" section of the introduction to this book.

L1104 *filename* **: invalid library**

The given file had an invalid format for a library.

L1105 **invalid object due to interrupted incremental compile**

Delete the object file, recompile the program, and relink.

L1106 **unknown COMDAT allocation type for** *symbol***; record ignored**

This is a translator error. The given symbol is either a routine or a data item.

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 in the "Microsoft Support Services" section of the introduction to this book.

L1107 **unknown COMDAT selection type for** *symbol***; record ignored**

This is a translator error. The given symbol is either a routine or a data item.

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 in the "Microsoft Support Services" section of the introduction to this book.

L1108 **invalid format of debugging information**

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 in the "Microsoft Support Services" section of the introduction to this book.

L1113 **unresolved COMDEF; internal error**

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 in the "Microsoft Support Services" section of the introduction to this book.

L1114 **unresolved COMDAT** *symbol***; internal error**

This is a translator error. The given symbol is either a routine or a data item.

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 in the "Microsoft Support Services" section of the introduction to this book.

L1115 *option* **: option incompatible with overlays**

The given option cannot be used when linking an overlaid program.

L1117 **unallocated COMDAT** *symbol***; internal error**

This is a translator error. The given symbol is either a routine or a data item.

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 in the "Microsoft Support Services" section of the introduction to this book.

L1123 *segment* **: segment defined both 16-bit and 32-bit**

Define the segment as either 16-bit or 32-bit.

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, the following causes this error:

```
mov     ax, seg mydata
```

L1128 too many nested include files in module-definition file

Nesting of **INCLUDE** statements in a module-definition (.DEF) file is limited to 10 levels.

L1129 missing or invalid include file name

The file specification in an **INCLUDE** statement in the module-definition (.DEF) file was missing or was not a valid filename.

LINK Error Messages

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 the Windows operating system.

L2002 fixup overflow at *number* **in segment** *segment*

This error message is followed by one of these strings:

- target external *symbol*
- **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 as 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 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 in the "Microsoft Support Services" section of the introduction to this book.

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 8 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 optimizing 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 optimizing 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 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 in the "Microsoft Support Services" section of the introduction to this book.

L2022 *entry* **(alias** *internalname***) : export undefined**

The internal name of the given exported routine or data item is undefined.

L2023 *entry* **(alias** *internalname***) : export imported**

The internal name of the given exported routine or data item conflicts with the internal name of a previously imported routine 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 size of the sum of the following exceeds 64K:

- Data declared in DGROUP
- The size of the heap specified in the **HEAPSIZE** statement in the module-definition (.DEF) file
- The size of the stack specified in either the /STACK option or the **STACKSIZE** statement in the .DEF file

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, followed by a list of object modules that contain references to this symbol. This message and the list of object modules 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 in identifiers.

This error can also occur when a program compiled with C/C++ version 7.0 (or later) is linked using /NOD. The /NOD option tells LINK to ignore all default libraries named in object files. C/C++ 7.0 embeds in an object file both the name of the default run-time library and OLDNAMES.LIB. To avoid this error, either specify OLDNAMES.LIB in the *libraries* field or specify /NOD:*library* where *library* is the name of the default run-time library to be excluded from the search.

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 a segmented-executable file.

The error message can be disabled by including the **REALMODE** statement in the module-definition (.DEF) 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 startup 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 file QUICKLIB.OBJ was missing. 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.

L2046 **share attribute conflict—segment** *segment* **in group** *group*

The given segment has a different sharing attribute than other segments that are assigned to the given group.

All segments assigned to a group must have the same attribute, either **SHARED** or **NONSHARED**. The attributes cannot be mixed.

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 an overlay manager was missing.

By default, the overlay manager is the Microsoft Overlay Virtual Environment (MOVE). This is provided in MOVE.LIB, which is a component library of the default combined libraries provided with Microsoft C/C++ version 7.0. The error occurs when LINK cannot find the **_moveinit** routine.

If the /OLDOVERLAY option is specified, the overlay manager is the Microsoft Static Overlay Manager, which is also provided in the default combined libraries.

L2050 **USE16/USE32 attribute conflict—segment** *segment* **in group** *group*

You cannot group 16-bit segments with 32-bit segments.

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, followed by a list of object modules that contain references to this symbol. The error message and the list of object modules 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.

L2057 **duplicate of** *function* **with different size found; record ignored**

An inconsistent class definition was found.

Check the include files and recompile.

L2058 **different duplicate of** *function* **found; record ignored**

An inconsistent class definition was found.

Check the include files and recompile.

L2060 **size of data block associated with** *symbol* **(16-bit segment) exceeds 64K**

A class had too many virtual functions. The given symbol is the v-table for the class, in the form of a decorated name.

L2061 **no space for data block associated with** *function* **inside** *segment* **segment**

The given function was allocated to the given segment, but the segment was full.

L2062 **continuation of COMDAT** *function* **has conflicting attributes; record ignored**

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 in the "Microsoft Support Services" section of the introduction to this book.

L2063 *function* **is allocated in undefined segment**

The given function was allocated to a nonexistent segment.

L2064 **starting address not in the root overlay**

The segment or object file that contains the starting address for the program was placed into an overlay.

The starting address in a C-language program is provided by the **main** function.

LINK Warning Messages

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 with MS-DOS.

L4004 **possible fixup overflow at** *offset* **in segment** *segment*

A near call or jump was made to another segment that 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 for segmented-executable programs.

L4014 *option* **: option ignored for DOS executable file**

The given option is not allowed for MS-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**

The given option was not a valid LINK option. LINK ignored the option specification.

One of the following may be a cause:

- An obsolete option was specified to the current version of LINK. For example, the /INCR option is obsolete in LINK version 5.30. The current options are described in the manual and in online Help. To see a list of options, run LINK with the /? option.
- An old version of LINK was used. Check your path. To see the version number of LINK, run LINK with the /? option.
- The name was incorrectly specified. For example, the option specification /NODEFAULTLIBSEARCH is an invalid abbreviation of the /NODEFAULTLIBRARYSEARCH option. Option names can be shortened by removing letters only from the end of the name.

L4018 **missing or unrecognized application type; option** *option* **ignored**

The /PM option accepts only the keywords **PM**, **VIO**, and **NOVIO**.

L4020 *segment* **: code-segment size exceeds 64K–36**

Code segments that are 65,501 through 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. You can ignore this message if you have 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 segmented-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 error can occur if a module tries to import a function or data item from itself or from another source (such as a DLL) that has the same name.

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 a **SEGMENTS** statement of the module-definition (.DEF) 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 1024 overlay segments; extra put in root**

The limit on the number of segments that can go into overlays is 1024. Segments starting with the 1025th 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 segmented-executable applications require DGROUP.

This warning will not be issued if **DATA NONE** is declared or if the executable file is a dynamic-link library.

L4037 *group* **: both USE16 and USE32 segments in group; assuming USE32**

The given group was allocated contributions from both 16-bit segments and 32-bit segments.

L4038 program has no starting address

The segmented-executable application had no starting address. A missing starting address will usually cause the program to fail.

High-level languages automatically specify a starting address. In a C-language program, this is provided by the **main** function.

If you are writing an assembly-language program, specify a starting address with the **END** statement.

MS-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 (.DEF) file could not be opened.

L4043 old version not segmented executable format

The file specified in the **OLD** statement in the module-definition (.DEF) file was not a valid segmented-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. In the following cases, if either .EXE or no extension is specified, LINK assumes the appropriate extension:

/TINY option: .COM
/Q option: .QLB
LIBRARY statement: .DLL

L4050 file not suitable for /EXEPACK; relink without

The size of the packed load image plus packing overhead was larger than it would be for the unpacked load image. There is no advantage to packing this program.

Remove /EXEPACK from the LINK command line. In PWB, clear the Pack EXE File check box in the Additional Debug/Release Options dialog box under Link Options.

This warning also occurs if the name specified in the **LIBRARY** statement in the module-definition (.DEF) file does not match the name specified in the *exefile* field.

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, or 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.

L4055 **start address not equal to 0x100 for /TINY**

The starting address for a .COM file must be 100 hexadecimal.

Put the following line of assembly source code in front of the code segment:

ORG 100h

L4056 **/EXEPACK valid only for OS/2 and real-mode DOS; ignored**

The /EXEPACK option is incompatible with Windows-based programs.

L4057 **stack specified for DLL; ignored**

A stack was specified for a dynamic-link library (DLL). Either the /STACK option was used on the command line or the **STACKSIZE** statement was used in the module-definition (.DEF) file. LINK ignored the specification and did not create a stack.

A DLL does not have a stack.

L4058 **ignoring alias for already defined symbol** *symbol*

The specified symbol was redefined in the program. However, it is an identifier from a C run-time library that has an alias to a new name in OLDNAMES.LIB. LINK ignored the alias for the symbol.

This warning appears only when the /INFO option is specified.

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.

L4076 **no segments defined**

There was no code in the program.

This warning can occur if the file contains only resources.

L4077 **symbol** *function* **not defined; ordered allocation ignored**

The given function was specified in a **FUNCTIONS** statement in the module-definition (.DEF) file, but the function was not defined.

L4079 **symbol** *function* **already defined for ordered allocation; duplicate ignored**

The given function was specified twice in **FUNCTIONS** statements in the module-definition (.DEF) file.

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.

L4081 **cannot execute** *program arguments—message*

LINK could not run the given program (with the given arguments) for the given reason.

L4082 **changing overlay assignment for segment** *segment* **from** *oldnumber* **to** *newnumber*

The given segment was assigned to two overlays, represented by *oldnumber* and *newnumber*. LINK assumed the *newnumber* overlay.

Probably a command-line overlay specification with parentheses conflicted with an overlay specification in the module-definition (.DEF) file.

L4083 **changing overlay assignment for symbol** *symbol* **from** *oldnumber* **to** *newnumber*

The given symbol was assigned to two overlays, represented by *oldnumber* and *newnumber*. LINK assumed the *newnumber* overlay.

Probably a command-line overlay specification with parentheses conflicted with an overlay specification in the module-definition (.DEF) file.

L4084 *option* **: argument missing; option ignored**

The given option requires an argument, but none was specified.

For example, the following option specification causes this error:

```
/ONERROR
```

L4085 *option* **: argument invalid; assuming** *argument*

The given option was specified with a numeric argument that was out of range for the option. LINK assumed the given argument.

For example, the option specification /DYNAMIC:11000 causes the following error:

```
/DYNAMIC:11000 : argument invalid; assuming 10922
```

L4086 **/r not first on command line; ignored**

This message appears if the /r option is not specified before other LINK options. /r must be the first option specified or it will be ignored.

ML Error Messages

ML Fatal Errors

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 MS-DOS, increase the number of file handles by changing the FILES setting in CONFIG.SYS to allow a larger number of open files. FILES=50 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.

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.

This error is generally caused when there is a syntax error on the command line.

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.

A1017 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 in the "Microsoft Support Services" section of the introduction to this book.

ML Nonfatal Errors

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.

A2007 **non-benign record redefinition**

A RECORD definition conflicted with a previous definition.

One of the following occurred:

- There were different numbers of fields.
- There were different numbers of bits in a field.
- There was a different label.
- There were different initializers.

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 like ||.

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 variables 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 righthand 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 attempted 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 when generating 16-bit code.

A2199 .STARTUP does not work with 32-bit segments

The **.STARTUP** directive cannot be used in a 32-bit segment; it is valid only when generating 16-bit code.

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.

A2202 illegal use of segment register

You cannot use segment overrides for the FS or GS segment registers when generating floating-point emulation instructions with the /FPi command-line option or **OPTION EMULATOR**.

A2203 cannot declare scoped code label as PUBLIC

A code label defined with the “label:” syntax was declared **PUBLIC**. Use the “label::” syntax, the **LABEL** directive, or **OPTION NOSCOPED** to eliminate this error.

A2204 .MSFLOAT directive is obsolete : ignored

The Microsoft Binary Format is no longer supported. You should convert your code to the IEEE numeric standard, which is used in the 80x87-series coprocessors.

A2205 ESC instruction is obsolete : ignored

The **ESC** (Escape) instruction is no longer supported. All numeric coprocessor instructions are now supported directly by the assembler.

A2206 missing operator in expression

An expression cannot be evaluated because it is missing an operator. This error message may also be a side-effect of a preceding program error.

The following line will generate this error:

```
value1 = ( 1 + 2 ) 3
```

A2207 missing right parenthesis in expression

An expression cannot be evaluated because it is missing a right (closing) parenthesis. This error message may also be a side-effect of a preceding program error.

The following line will generate this error:

```
value1 = ( ( 1 + 2 ) * 3
```

A2208 missing left parenthesis in expression

An expression cannot be evaluated because it is missing a left (opening) parenthesis. This error message may also be a side-effect of a preceding program error.

The following line will generate this error:

```
value1 = ( ( 1 + 2 ) * 3 ) )
```

A2209 reference to forward macro redefinition

A macro cannot be accessed because it has not been yet defined.

Move the macro definition ahead of all references to the macro.

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.

ML Warnings

A4000 cannot modify READONLY segment

An attempt was made to modify an operand in a segment marked with the READONLY 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.

A4005 unknown default prologue argument

An unknown argument was passed to the default prologue.

The default prologue understands only the **FORCEFRAME** and **LOADDS** arguments.

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 insufficient memory for /EP : /EP ignored

There is not enough memory to generate a first-pass listing.

A4010 expected '>' on text literal

A macro was called with a text literal argument that was missing a closing angle bracket.

A4011 multiple .MODEL directives found : .MODEL ignored

More than one **.MODEL** directive was found in the current module. Only the first **.MODEL** statement is used.

A4012 line number information for segment without class 'CODE'

There were instructions in a segment that did not have a class name that ends with "CODE." The assembler did not generate CodeView information for these instructions.

CodeView cannot process modules with code in segments with class names that do not end with "CODE."

A4013 instructions and initialized data not supported in AT segments

An instruction or initialized data was found in a segment defined with the AT attribute. The code or data will not be loaded at run time.

Data in AT segments must be declared with the **?** initializer.

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.

A6004 procedure argument or local not referenced

You passed a procedure argument or created a variable with the LOCAL directive that was not used in the procedure body.

Unnecessary parameters and locals waste code and stack space.

A6005 expression condition may be pass-dependent

Under the /Zm command-line option or the OPTION M510 directive, the value of an expression changed between passes.

This error message may indicate that the code is pass-dependent and must be rewritten.

NMAKE Error Messages

This section lists error messages generated by the NMAKE utility.

Microsoft Program Maintenance Utility (NMAKE) generates the following error messages:

- Fatal errors (U1000 through U1099) cause NMAKE to stop execution.
- Errors (U2001) do not stop execution but prevent NMAKE from completing the make process.
- Warnings (U4001 through U4011) indicate possible problems in the make process.

NMAKE Fatal Error Messages

U1000 syntax error : ')' missing in macro invocation

A left parenthesis, **(**, appeared without a matching right parenthesis, **)**, in a macro invocation. The correct form is **$**(*name*), and **$***n* is allowed for one-character names.

U1001 syntax error : illegal character *character* **in macro**

The given character appeared in a macro but was not a letter, number, or underscore (_).

If the colon (**:**) is omitted in a macro expansion, the following error occurs:

```
syntax error : illegal character '=' in 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 specify a dollar sign, use a double dollar sign (**$$**) or precede it with a caret (^).

U1003 syntax error : '=' missing in macro substitution

A macro invocation contained a colon (**:**), which begins a substitution, but it did not contain an equal sign (**=**).

The correct form is:

```
$(macroname:oldstring=newstring)
```

U1004 syntax error : macro name missing

One of the following occurred:

- The name of a macro being defined was itself a macro invocation that expanded to nothing. For example, if the macro named `ONE` is undefined or has a null value, the following macro definition causes this error:

  ```
  $(ONE)=TWO
  ```

- A macro invocation did not specify a name in the parentheses. The following specification causes this error:

  ```
  $()
  ```

  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 specified directive 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**

The limit on nesting of **!IF** directives was exceeded.

The **!IF** preprocessing directives include **!IF**, **!IFDEF**, **!IFNDEF**, **!ELSE IF**, **!ELSE IFDEF**, and **!ELSE IFNDEF**.

U1020 **end-of-file found before next directive**

An expected directive was missing.

For example, an **!IF** was not followed by an **!ENDIF**.

U1021 **syntax error : !ELSE unexpected**

An **!ELSE** directive was found that was not preceded by an **!IF** directive, or the directive was placed in a syntactically incorrect place.

The **!IF** preprocessing directives include **!IF**, **!IFDEF**, **!IFNDEF**, **!ELSE IF**, **!ELSE IFDEF**, and **!ELSE IFNDEF**.

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 a macro representing the filename expanded to nothing.

U1033 **syntax error : *string* unexpected**

The given string is not part of the valid syntax for a makefile.

The following are examples of causes and results of this error:

- If the closing set of angle brackets for an inline file are not at the beginning of a line, the following error occurs:

  ```
  syntax error : 'EOF' unexpected
  ```

- If a macro definition in the makefile contained an equal sign (=) without a preceding name or if the name being defined is a macro that expands to nothing, the following error occurs:

  ```
  syntax error : '=' unexpected
  ```

- If the semicolon (**;**) in a comment line in TOOLS.INI is not at the beginning of the line, the following error occurs:

  ```
  syntax error : ';' unexpected
  ```

- If the makefile has been formatted by a word processor, the following error can occur:

  ```
  syntax error : ':' unexpected
  ```

U1034 **syntax error : separator missing**

The colon (**:**) that separates targets and dependents is missing.

U1035 **syntax error : expected ':' or '=' separator**

Either a colon (**:**) or an equal sign (=) was expected.

Possible causes include the following:

- A target was not followed by a colon.
- A single-letter target was followed by a colon and no space (such as `a:`). NMAKE interpreted it as a drive specification.
- An inference rule was not followed by a colon.
- A macro definition was not followed by an equal sign.
- A character followed a backslash (\) that was used to continue a command to a new line.
- A string appeared that did not follow any NMAKE syntax rule.
- The makefile was formatted by a word processor.

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 in the “Microsoft Support Services” section of the introduction to this book.

U1039 **internal error : parser**

Note the circumstances of the error and notify Microsoft Corporation by following the instructions in the “Microsoft Support Services” section of the introduction to this book.

U1040 **internal error : macro expansion**

Note the circumstances of the error and notify Microsoft Corporation by following the instructions in the “Microsoft Support Services” section of the introduction to this book.

U1041 **internal error : target building**

Note the circumstances of the error and notify Microsoft Corporation by following the instructions in the “Microsoft Support Services” section of the introduction to this book.

U1042 **internal error : expression stack overflow**

Note the circumstances of the error and notify Microsoft Corporation by following the instructions in the “Microsoft Support Services” section of the introduction to this book.

U1043 **internal error : temp file limit exceeded**

Note the circumstances of the error and notify Microsoft Corporation by following the instructions in the “Microsoft Support Services” section of the introduction to this book.

U1045 **spawn failed :** *message*

A program or command, called by NMAKE, failed for the given reason.

U1047 **argument before ')' expands to nothing**

The parentheses following the preprocessing operator **DEFINED** or **EXIST** either were empty or contained an argument that evaluated to a null string.

U1048 **cannot write to file** *filename*

NMAKE could not write to the given file.

One cause of this error is a read-only file specified with /X.

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 makefile was too large or complex for available memory.

U1052 file *filename* **not found**

NMAKE could not find the given file, which was specified with one of the following:

- The /F option
- The **!INCLUDE** preprocessing directive
- The at sign (**@**) specifier for a response file

Check that the file exists and the filename is spelled correctly.

U1053 file *filename* **unreadable**

The file cannot be read.

One of the following may be a cause:

- The file is in use by another process.
- A bad area exists on disk.
- A bad file-allocation table exists.

U1054 cannot create inline file *filename*

NMAKE failed to create the given inline file.

One of the following may be a cause:

- A file by that name exists with a read-only attribute.
- The disk is full.

U1055 out of environment space

The operating system ran out of room for environment variables.

Either increase the environment space or set fewer environment variables.

U1056 cannot find command processor

The command processor was not in the path specified in the COMSPEC or PATH environment variables.

NMAKE uses COMMAND.COM or CMD.EXE as a command processor when executing commands. It looks for the command processor first in the path set in COMSPEC. If COMSPEC does not exist, NMAKE searches the directories specified in PATH.

U1057 cannot delete temporary file *filename*

NMAKE failed to delete the temporary inline file.

U1058 terminated by user

NMAKE was halted by CTRL+C or CTRL+BREAK.

U1059 syntax error: '}' missing in dependent

A search path for a dependent was incorrectly specified. Either a space existed in the path or the closing brace (**}**) was omitted. The syntax for a directory specification for a dependent is:

```
{directories}dependent
```

where *directories* specifies one or more paths, each separated by a semicolon (;). No spaces are allowed.

If part or all of a search path is replaced by a macro, be sure that no spaces exist in the macro expansion.

U1060 unable to close file : *filename*

NMAKE encountered an error while closing a file.

One of the following may be a cause:

- 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 must be followed by either a makefile name or a dash (**–**), which represents standard input.

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 '='

A macro definition on the NMAKE command line contained an equal sign (**=**) without a preceding name.

This error can occur if the macro name being defined is itself a macro that expands to nothing.

U1064 MAKEFILE not found and no target specified

The NMAKE command line did not specify a makefile or a target, and the current directory did not contain a file named MAKEFILE.

NMAKE requires either a makefile or a command-line target. To make a makefile available to NMAKE, either specify the /F option or place a file named MAKEFILE in the current directory. NMAKE can create a command-line target by using an inference rule if a makefile is not provided.

U1065 invalid option *option*

The specified option is not a valid option for NMAKE.

U1069 no match found for wildcard *filename*

There is no file that matches the given filename, which was specified using one or more wildcards (***** and **?**).

A target file specified using a wildcard must exist on disk.

U1070 cycle in macro definition *macroname*

The given macro definition contained a macro whose definition contained the given macro. Circular macro definitions are invalid.

For example, the following macro definitions:

```
ONE=$(TWO)
TWO=$(ONE)
```

cause the following error:

```
cycle in macro definition 'TWO'
```

U1071 cycle in dependency tree for target *targetname*

A circular dependency exists in the dependency tree for the given target. The given target is a dependent of one of the dependents of the given target. Circular dependencies are invalid.

U1072 cycle in include files : *filename*

The given file includes a file that eventually includes the given file. Inclusions (using the **!INCLUDE** preprocessing directive) cannot be circular.

U1073 don't know how to make *targetname*

The specified target does not exist, and there is no command to execute or inference rule to apply.

One of the following may be a solution:

- Check the spelling of the target name.
- If *targetname* is a pseudotarget, specify it as a target in another description block.
- If *targetname* is a macro invocation, be sure it does not expand to a null string.

U1076 **name too long**

A string exceeded one of the following limits:

- A macro name cannot exceed 1024 characters.
- A target name (including path) cannot exceed 256 characters.
- A command cannot exceed 2048 characters.

U1077 *program* **: return code** *value*

The given command or program called by NMAKE failed and returned the given exit code.

To suppress this error and continue the NMAKE session, use the /I option, the **.IGNORE** dot directive, or the dash (**–**) command modifier. To continue the NMAKE session for unrelated parts of the dependency tree, use the /K option.

U1078 **constant overflow at** *expression*

The given expression contained a constant that exceeded the range –2,147,483,648 to 2,147,483,647. The constant appeared in one of the following situations:

- An expression specified with a preprocessing directive
- An error level specified with the dash (**–**) command modifier

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 *filename* **: program not found**

NMAKE could not find the given program in order to run it.

Make sure that the program is in a directory specified in the PATH environment variable and is not misspelled.

U1082 *command* **: cannot execute command; out of memory**

There is not enough memory to execute the given command.

U1083 **target macro** *target* **expands to nothing**

The given target is an invocation of a macro that has not been defined or has a null value. NMAKE cannot process a null target.

U1084 **cannot create temporary file** *filename*

NMAKE was unable to create the temporary file it needs when it processes the makefile.

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 directory specified in the TMP environment variable does not exist.

U1085 **cannot mix implicit and explicit rules**

A target and a pair of inference-rule extensions were specified on the same line. Targets cannot be named in inference rules.

U1086 **inference rule cannot have dependents**

The colon (**:**) in an inference rule must be followed by one of the following:

- A newline character
- A semicolon (**;**), which can be followed by a command
- A number sign (**#**), which can be followed by a comment

U1087 **cannot have : and :: dependents for same target**

A target cannot be specified in both a single-colon (**:**) and a double-colon (**::**) dependency.

To specify a target in multiple description blocks, use **::** in each dependency line.

U1088 **invalid separator '::' on inference rule**

An inference rule must be followed by a single colon (**:**).

U1089 **cannot have build commands for directive** *targetname*

Dot directives cannot be followed by commands. The dot directives are **.IGNORE**, **.PRECIOUS**, **.SILENT**, and **.SUFFIXES**.

U1090 **cannot have dependents for directive** *targetname*

Dot directives cannot be followed by dependents. The dot directives are **.IGNORE**, **.PRECIOUS**, **.SILENT**, and **.SUFFIXES**.

U1092 **too many names in rule**

An inference rule cannot specify more than two extensions.

U1093 **cannot mix dot directives**

Multiple dot directives cannot be specified on one line. The dot directives are **.IGNORE**, **.PRECIOUS**, **.SILENT**, and **.SUFFIXES**.

U1094 **syntax error : only (NO)KEEP allowed here**

Something other than **KEEP** or **NOKEEP** appeared after the closing set of angle brackets (<<) specifying an inline file. Only **KEEP**, **NOKEEP**, or a newline character may follow the angle brackets. No spaces, tabs, or other characters may appear.

KEEP preserves the inline file on disk. **NOKEEP** deletes the file after the NMAKE session. The default is **NOKEEP**.

U1095 **expanded command line** *commandline* **too long**

After macro expansion, the given command line exceeded the limit on length of command lines for the operating system.

MS-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 inline file** *filename*

NMAKE could not create the given inline file.

One of the following occurred:

- The disk was full.
- A file with that name exists as a read-only file.

U1097 **filename-parts syntax requires dependent**

The current dependency does not have either an explicit dependent or an implicit dependent. Filename-parts syntax, which uses the percent (**%**) specifier, represents components of the first dependent of the current target.

U1098 **illegal filename-parts syntax in** *string*

The given string does not contain valid filename-parts syntax.

U1099 The makefile being processed was too complex for the current stack allocation in NMAKE. NMAKE has an 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.

NMAKE Error Messages

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 MS-DOS, increase the number of file handles by changing the FILES setting in CONFIG.SYS to allow a larger number of open files. FILES=50 is the recommended setting.

NMAKE Warning Messages

U4001 command file can be invoked only from command line

A command file, which is invoked by the at sign (**@**) specifier, cannot contain a specification for another command file. Such nesting is not allowed. The specification was ignored.

U4002 resetting value of special macro *macroname*

The given predefined macro was redefined.

U4004 **too many rules for target** *targetname*

More than one description block was specified for the given target using single colons (**:**) as separators. NMAKE executed the commands in the first description block and ignored later blocks.

To specify the same target in multiple dependencies, use double colons (**::**) as the separator in each dependency line.

U4005 **ignoring rule** *rule* **(extension not in .SUFFIXES)**

The given rule contained a suffix that is not specified in the **.SUFFIXES** list. NMAKE ignored the rule.

This warning appears only when the /P option is used.

U4006 **special macro undefined :** *macroname*

The given 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 8 characters, or the extension has more than three characters. NMAKE truncated the name to an 8-character base and a 3-character extension.

If long filenames are supported by your file system, enclose the name in double quotation marks (").

U4008 **removed target** *target*

NMAKE was interrupted while trying to build the given target, and the target file was incomplete. Because the target was not specified in the **.PRECIOUS** list, NMAKE deleted the file.

U4010 *target* **: build failed; /K specified, continuing ...**

A command in the commands block for the given target returned a nonzero exit code. The /K option told NMAKE to continue processing unrelated parts of the build and to issue an exit code 1 when the NMAKE session is finished.

If the given target is itself a dependent for another target, NMAKE issues warning U4011 after this warning.

U4011 *target* **: not all dependents available; target not built**

A dependent of the given target either did not exist or was out of date, and a command for updating the dependent returned a nonzero exit code. The /K option told NMAKE to continue processing unrelated parts of the build and to issue an exit code 1 when the NMAKE session is finished.

This warning is preceded by warning U4010 for each dependent that failed to be created or updated.

PWB Error Messages

PWB displays an error message whenever it detects a command it cannot execute. Most errors terminate the command that is in error, but do not terminate PWB.

For most errors, PWB displays a message box with only the text of the message. The error number does not appear. With these messages, press F1 or click Help when the message box is displayed for Help on the error. Some errors terminate PWB. PWB displays these fatal errors on the command line after returning to the operating system.

This section lists only the fatal PWB errors.

PWB Fatal Errors

PWB3089 Out of local memory. Unable to recover.

PWB has run out of memory and cannot recover. This is a fatal PWB condition. However, PWB is able to save your files, and you can restart PWB to continue.

This can happen when using PWB continuously for a long time.

This can also happen when creating a project with a very large number of files or adding files to a large project. To make the largest amount of memory available to PWB for creating a very large project, load only the PWBUTILS extension and only the language extensions you need for the project. Start PWB with the /DS option, and create the project before doing any other work.

If the project is too large for PWB to handle as a PWB project, you can use a non-PWB makefile for your project.

PWB3090 Out of virtual memory space. Unable to recover.

PWB has run out of virtual memory and cannot recover. This is a fatal PWB condition. However, PWB is able to save your files, and you can restart PWB to continue.

PWB3096 Unsupported video mode. Please change modes and restart.

A request was made to start PWB with the **Savescreen** switch set to yes (the default), but PWB does not support the current operating-system video mode.

Change the video mode and restart PWB.

PWB3178 **Cannot start: unable to open swapping file**

PWB is unable to create its virtual-memory file on disk.

PWB creates this file in the directory pointed to by the TMP environment variable. If no TMP environment variable is set, PWB creates the file in the current directory.

Check that the disk has at least 2 free megabytes and that the directory can be accessed with permission to create a file. Check that the TMP environment variable lists a single existing directory.

PWB3180 **Cannot start: not enough far memory**

PWB ran out of memory while starting up.

Make more memory available to PWB and restart PWB.

PWB3181 **Cannot initialize**

PWB cannot initialize itself.

Check that there is enough memory available for PWB. Also, check that there is no conflict with a TSR (terminate-and-stay-resident) program.

PWB3901 **RE: error** *number***, line** *line*

PWB has encountered an error while processing a regular expression. The expression may be malformed or too complex.

Check that the syntax of the regular expression is correct.

PWB3909 **RemoveFile can't find file**

PWB has encountered an internal error.

Note the circumstances of the error and notify Microsoft Corporation by following the instructions in the "Microsoft Support Services" section of the introduction to this book.

PWB3912 **Internal VM Error**

PWB has encountered an internal error.

Note the circumstances of the error and notify Microsoft Corporation by following the instructions in the "Microsoft Support Services" section of the introduction to this book.

PWB12078 **Cannot access** *file*: *reason*

PWB cannot access the given file for the stated reason.

Correct the situation and restart PWB.

PWB12086 **Cannot access TMP directory:** *reason*

PWB cannot access the directory listed in the TMP environment variable for the stated reason.

Correct the situation and restart PWB.

SBRPACK Error Messages

This section lists error messages generated by the Microsoft Browse Information Compactor (SBRPACK). SBRPACK errors (SB *xxx*) are always fatal.

SB1000 UNKNOWN ERROR
Contact Microsoft Product Support Services

SBRPACK detected an unknown error condition.

Note the circumstances of the error and notify Microsoft Corporation by following the instructions in the "Microsoft Support Services" section of the introduction to this book.

This error ends SBRPACK with exit code 1.

SB1001 *option* **: unknown option**

SBRPACK did not recognize the given option.

This error ends SBRPACK with exit code 1.

SB1002 *sbrfile* **: corrupt file**

The given .SBR file is corrupt or does not have the expected format.

Recompile to regenerate the .SBR file.

This error ends SBRPACK with exit code 2.

SB1003 *sbrfile* **: invalid .SBR file**

SBRPACK did not recognize the given file as an .SBR file.

One of the following may be a solution:

- Check the spelling of the specified file.
- Recompile to regenerate the .SBR file.

This error ends SBRPACK with exit code 2.

SB1004 *sbrfile* **: incompatible .SBR version**

The given .SBR file cannot be packed by this version of SBRPACK.

One of the following may be a cause:

- The .SBR file was created by a compiler that is not compatible with this version of SBRPACK.
- The .SBR file is corrupt.

This error ends SBRPACK with exit code 2.

SB1005 *sbrfile* **: cannot open file**

SBRPACK cannot open the given .SBR file.

One of the following may be a cause:

- The .SBR file does not exist. Check the spelling.
- The .SBR file was locked by another process.

This error ends SBRPACK with exit code 3.

SB1006 **cannot create temporary .SBR file**

One of the following may have occurred:

- No more file handles were available. Increase the number of file handles by changing the FILES setting in CONFIG.SYS to allow a larger number of open files. FILES=50 is recommended.
- The disk was full.

This error ends SBRPACK with exit code 4.

APPENDIX B

Regular Expressions

A regular expression (sometimes called a “pattern”) is a find string that uses special characters to match patterns of text. You can use regular expressions to find patterns such as 5-digit numbers or strings in quotation marks. Selected portions of found text can be used in a replacement.

In PWB you can specify regular expressions in two ways: UNIX syntax and non-UNIX syntax. UNIX regular expressions have a syntax similar to regular expressions in the UNIX and XENIX operating systems. CodeView uses a subset of the UNIX regular-expression syntax. Non-UNIX regular-expression syntax has the features of UNIX regular expressions but includes additional features and uses a more compact syntax.

The **Unixre** switch determines whether PWB uses UNIX or non-UNIX regular expressions in searches. PWB switches that accept regular expressions, such as **Build** and **Word**, always use UNIX syntax.

Regular-Expression Summaries

The following table summarizes PWB’s UNIX regular-expression syntax.

Table B.1 UNIX Regular-Expression Summary

Syntax	Description
c	Escape: literal character *c*
.	Wildcard: any character
^	Beginning of line
$	End of line
[*class***]**	Character class: any one character in set
[^*class***]**	Inverse class: any one character not in set
*x******	Repeat: zero or more occurrences of *x*
*x***+**	Repeat: one or more occurrences of *x*

Table B.1 UNIX Regular-Expression Summary (*continued*)

Syntax	Description
\{*x***\}**	Grouping: group subexpression for repetition
\{*x***\!***y***\!***z***\}**	Alternation: match one from the set
\~*x*	"Not": fail if *x* at this point
\(*x***\)**	Tagged expression
n	Reference to tagged expression number *n*
\:*e*	Predefined expression

The following table summarizes the UNIX predefined expressions.

Table B.2 UNIX Predefined Expressions

Syntax	Description
\:a	Alphanumeric character
\:b	White space
\:c	Alphabetic character
\:d	Digit
\:f	Part of a filename
\:h	Hexadecimal number
\:i	Microsoft C/C++ identifier
\:n	Unsigned number
\:p	Path
\:q	Quoted string
\:w	English word
\:z	Unsigned integer

CodeView uses a subset of the UNIX regular-expression syntax. You can use regular expressions as arguments to the Search (/) command and Examine Symbols (**X**) command. The following table summarizes CodeView regular expressions.

Table B.3 CodeView Regular Expressions

Character	Syntax	Meaning
Backslash	*c*	Matches a literal character *c*. (Escape)
Period	**.**	Matches any single character. (Wildcard)
Caret	^	Matches the beginning of a line. The caret must appear at the beginning of the pattern.
Dollar sign	$	Matches the end of a line. The dollar sign must appear at the end of the pattern.
Asterisk	*c******	Matches zero or more occurrences of *c*.
Brackets	**[...]**	Matches any one character in the set of the characters within the brackets.

Within the brackets, you can specify a negated set and ranges of characters by using the following notation:

Character	Syntax	Meaning
Dash	–	Specifies a range of characters in the ASCII order between the characters on either side, inclusive. For example, `[a-z]` matches the lowercase alphabet.
Caret	^	Matches any one character not within the brackets. The caret must be the first character within the brackets. For example, `[^0-9]` matches any character except a digit.

The following table summarizes the non-UNIX regular-expression syntax.

Table B.4 Non-UNIX Regular-Expression Summary

Syntax	Description
c	Escape: literal character *c*
?	Wildcard: any character
^	Beginning of line
$	End of line
[*class***]**	Character class: any one character in set
[~*class***]**	Inverse class: any one character not in set
*x******	Repeat: zero or more occurrences of *x*
*x***+**	Repeat: one or more occurrences of *x*
*x***@**	Repeat: maximal zero or more occurrences of *x*
*x***#**	Repeat: maximal one or more occurrences of *x*
(*x***)**	Grouping: group subexpression for repetition
(*x***!***y***!***z***)**	Alternation: match one from the set

Table B.4 Non-UNIX Regular-Expression Summary (*continued*)

Syntax	Description
~*x*	"Not": fail if *x* at this point
x^*n*	"Power": match *n* copies of *x*
{*x***}**	Tagged expression
$*n*	Reference to tagged expression number *n*
:*e*	Predefined expression

The following table summarizes the non-UNIX predefined expressions.

Table B.5 Non-UNIX Predefined Expressions

Syntax	Description
:a	Alphanumeric character
:b	White space
:c	Alphabetic character
:d	Digit
:f	Part of a filename
:h	Hexadecimal number
:i	Microsoft C/C++ identifier
:n	Unsigned number
:p	Path
:q	Quoted string
:w	English word
:z	Unsigned integer

UNIX Regular-Expression Syntax

PWB uses the following UNIX-style regular-expression syntax:

Table B.6 UNIX Regular-Expression Syntax

Syntax	Description
c	Escape: matches a literal occurrence of the character *c* and ignores any special meaning of *c* in a regular expression. For example, the expression `\?` matches a question mark (?), `\^` matches a caret (^), and `\\` matches a backslash (\).
.	Wildcard: matches any single character. For example, the expression `a.a` matches `aaa` and `a0a`.

Table B.6 UNIX Regular-Expression Syntax (*continued*)

Syntax	Description
^	Beginning of line. For example, the expression `^The` matches the word `The` only when it occurs at the beginning of a line.
$	End of line. For example, the expression `end$` matches the word `end` only when it occurs at the end of a line.
[*class***]**	Character class: matches any one character in the class. Use a dash (–) to specify a range of characters. Within a class, all characters except `^-\]` are treated literally. For example, `[a-zA-Z0-9]` matches any character or digit, and `[abc]` matches `a`, `b`, or `c`.
[^*class***]**	Inverse of character class: matches any character not specified in the class. For example, `[^0-9]` matches any character that is not a digit.
*x******	Repeat operator: matches zero or more occurrences of *x*, where *x* is a single character, a character class, or a grouped expression. For example, the regular expression `ba*b` matches `baaab`, `bab`, and `bb`. This operator always matches as many characters as possible.
x+	Repeat operator (shorthand for *xx**): matches one or more occurrences of *x*. For example, the regular expression `ba+b` matches `baab` and `bab` but not `bb`.
\(*x***\)**	Tagged expression: marked text, which you can refer to as *n* elsewhere in the find or replacement string. Within a find string, PWB finds text that contains the previously tagged text. Within a replacement string, PWB reuses the matched text in the replacement.
n	References the characters matched by a tagged expression, where *n* is a one-digit number and indicates which expression. The first tagged expression is `\1`, the second is `\2`, and so on. The entire expression is represented as `\0`.
\{*x***\}**	Grouping. Groups a regular expression so that you can use a repeat operator on the subexpression. For example, the regular expression `\{Test\}+` matches `Test` and `TestTest`.
\{*x***\!***y***\!***z***\}**	Alternation: matches one from a set of alternate patterns. The alternates are tried in left-to-right order. The next alternate is tried only when the rest of the pattern fails. For example, `\{ +\!$\}` matches a sequence of blanks or the end of a line.
\~*x*	"NOT" function: matches nothing but checks to see whether the text matches *x* at this point and fails if it does. For example, `^\~\{ \!$\}.*` matches all lines that do not begin with white space or end of line.
\:*e*	Predefined regular expression, where *e* is a letter specifying the regular expression.

Examples

In PWB, to find the next occurrence of a number (a string of digits) that begins with the digit 1 or 2:

1. Execute **Arg Arg** (ALT+A ALT+A)
2. Type `[12][0-9]*`
3. Execute **Psearch** (F3)

The special characters in regular expression syntax are most powerful when they are used together. For example, the following combination of the wildcard (**.**) and repeat (*****) characters

```
.*
```

matches any string of characters. This expression is useful when it is part of a larger expression, such as

```
B.*ing
```

which matches any string beginning with `B` and ending with `ing`.

Tagged Regular Expressions

Tagged expressions are regular expressions enclosed by the delimiters \ and \ (UNIX) or **{** and **}** (non-UNIX). Use tagged expressions to match repeated elements and to mark substrings for use in a replacement. Note that a tagged expression is not the same as a grouped expression.

When you specify a regular expression with tagged subexpressions, PWB finds text that matches the regular expression and marks each substring matching a tagged subexpression.

Example

The UNIX regular expression

```
\(<\)\([^>]+\)\(>\)
```

matches the string

```
<bracketed>
```

and tags the `<`, `bracketed`, and `>` substrings.

To refer to tagged text in a find or replacement pattern, use *n* (UNIX) or **$***n* (non-UNIX), where *n* is the number of a tagged subexpression from 1 to 9. In a find pattern, this reference matches another occurrence of the previously matched text, not another occurrence of the regular expression. In a replacement, PWB uses the matched text.

The entire match is implicitly tagged for use in replacement text. Use **\0** (UNIX) or **$0** (non-UNIX) to refer to the entire match. For example, the UNIX find pattern

```
^\([^ ]+\) +\([^ ]+\).*
```

with the replace pattern

```
\2 \1 (\0)
```

matches lines without leading spaces and at least two words. It replaces them with lines that consist of the transposed words followed by the original line in parentheses.

Example

The tagged expressions:

UNIX	Non-UNIX
`\([A-Za-z]+\)==\1`	`{[A-Za-z]+}==$1`

match one or more letters followed by two equal signs (==) and a repetition of the letters. They match the first two strings below, but not the third:

```
ABCxyz==ABCxyz
i==i
ABCxyz==KBCxjj
```

The following example finds one or more hexadecimal digits followed by the letter H. Each matching string is replaced by a string that consists of the original digits (which were tagged so they could be reused) and the prefix 16#.

1. Find strings of the form *hexdigits***H** with the UNIX and non-UNIX patterns:

UNIX	Non-UNIX
`\([0-9a-fA-F]+\)H`	`{[0-9a-fA-F]+}H`

   These patterns can also be expressed by using the predefined pattern for hexadecimal digits:

UNIX	Non-UNIX
`\(\:h*\)H`	`{:h}H`

2. Replace with the patterns:

UNIX	Non-UNIX
`16#\1`	`16#$1`

Tagged Expressions in Build:Message

PWB uses tagged UNIX regular expressions to find the location of errors and warnings displayed in the Build Results window. The tagged portions of the message indicate the file and the location or token in error.

To define new messages for PWB to recognize, add a new **Build:message** switch definition to the [PWB] section of TOOLS.INI. The syntax for this switch is:

Build:message "*pattern*" ⟦*file* ⟦*line* ⟦*column*⟧ ⟧ | *token*⟧

The pattern is a macro string that specifies a tagged UNIX regular expression. The file, line, col, and token keywords indicate the meaning of each tagged subexpression.

For example, if the messages you want to match look like:

```
Error: Missing ';' on line 123 in SAMPLE.XYZ
```

Place the following setting in TOOLS.INI:

```
Build:message "^Error: .* on line \\(\\:z\\) in \\(\\:p\\)"  \
     file line
```

Note that each backslash in the regular expression is doubled within the macro string. This pattern uses the predefined expressions for integer (**\:z**) and path (**\:p**).

Justifying Tagged Expressions

To justify a tagged regular expression, use the following syntax in the replacement string:

UNIX	Non-UNIX
\(*width*,*n***)**	**$(***width*,*n***)**

The *width* is the field size (negative for left justification), and *n* is the number of the tagged expression to justify.

PWB justfies the tagged text according to the following rules:

- If *width* is greater than the length of the tagged text, PWB right-justifies the tagged expression within the field and pads the field with leading spaces.
- If *width* is negative and its magnitude is greater than the length of the text, PWB left-justifies the expression and pads the field with trailing spaces.

- If *width* is less than or equal to the length of the text, PWB uses the whole text and does not pad the field. PWB never truncates justified text.

Predefined Regular Expressions

PWB predefines several regular expressions. The definitions in the following table are listed in quoted non-UNIX syntax, as they would appear in a PWB macro. Use a predefined expression by entering **\:***e* (UNIX) or **:***e* (non-UNIX).

Table B.7 Predefined Regular Expressions and Definitions

:*e*	**Description** **Definition (non-UNIX)**
:a	Alphanumeric character `"[a-zA-Z0-9]"`
:b	White space `"([ \t]#)"`
:c	Alphabetic character `"[a-zA-Z]"`
:d	Digit `"[0-9]"`
:f	Part of a filename `"([~/\\\\ \\\"\\[\\]\\: +=;,.]#!..!.)"`
:h	Hexadecimal number `"([0-9a-fA-F]#)"`
:i	Microsoft C/C++ identifier `"([a-zA-Z_$][a-zA-Z0-9_$]@)"`
:n	Unsigned number `"([0-9]#.[0-9]@![0-9]@.[0-9]#![0-9]#)"`
:p	Path `"(([A-Za-z]\\:!)(\\\\!/!)(:f(.:f!)(\\\\!/))@:f(.:f!.!))"`
:q	Quoted string `"(\"[~\"]@\"!'[~']@')"`
:w	English word `"([a-zA-Z]#)"`
:z	Unsigned integer `"([0-9]#)"`

Non-UNIX Regular-Expression Syntax

PWB uses the following non-UNIX regular-expression syntax:

Table B.8 Non-UNIX Regular Expression Syntax

Syntax	Description
c	Escape: matches a literal occurrence of the character *c* and ignores any special meaning of *c* in a regular expression. For example, the expression `\?` matches a question mark `?`, \^ matches a caret `^`, and `\\` matches a backslash `\`.
?	Wildcard: matches any single character. For example, the expression `a?a` matches `aaa` and `a1a` but not `aBBBa`.
^	Beginning of line. For example, the expression `^The` matches the word `The` only when it occurs at the beginning of a line.
$	End of line. For example, the expression `end$` matches the word `end` only when it occurs at the end of a line.
[*class***]**	Character class: matches any one character in the class. Use a dash (–) to specify a range of characters. Within a class, all characters except `~-\]` are treated literally. For example, `[a-zA-Z*]` matches any alphabetic character or asterisk, and `[abc]` matches a single `a`, `b`, or `c`.
[~*class***]**	Inverse of character class: matches any single character not in the class. For example, `[~0-9]` matches any character that is not a digit.
*x******	Minimal matching: matches zero or more occurrences of *x*, where *x* is a single character or a grouped expression. For example, the expression `ba*b` matches `baaab`, `bab`, and `bb`.
x+	Minimal matching plus (shorthand for *xx******): matches one or more occurrences of *x*. For example, the expression `ba+b` matches `baab` and `bab` but not `bb`.
*x***@**	Maximal matching: identical to *x******, except that it matches as many occurrences as possible.
x#	Maximal matching plus: identical to *x*+, except that it matches as many occurrences as possible.
(*x*!*y*!*z***)**	Alternation: matches one from a set of alternate patterns. The alternates are tried in left-to-right order. The next alternate is tried only when the rest of the pattern fails. For example, the expression `(ww!xx!xxyy)zz` matches `xxzz` on the second alternative and `xxyyzz` on the third.
(*x***)**	Grouping. Groups an expression so that you can use a repeat operator with the expression. For example, the expression `(Test)+` matches `Test` and `TestTest`.
~*x*	"NOT" function: matches nothing but checks to see if the text matches *x* at this point and fails if it does. For example, `^~(if!while)?*$` matches all lines that do not begin with `if` or `while`.

Table B.8 Non-UNIX Regular Expression Syntax (*continued*)

Syntax	Description
x^*n*	Power function: matches *n* copies of *x*. For example, `w^4` matches `wwww`, and `(a?)^3` matches `a#aba5`.
{*x***}**	Tagged expression: marked text, which you can refer to as **$***n* elsewhere in the find or replacement string. Within a find string, PWB finds text that contains the previously tagged text. Within a replacement string, PWB reuses the matched text in the replacement.
$*n*	Reference to text matched by a tagged expression. The specific substring is indicated by *n*. The first tagged substring is indicated as `$1`, the second as `$2`, and so on. A `$0` represents the entire match.
:*e*	Predefined regular expression, where *e* is a letter that specifies the regular expression.

Examples

In PWB, to find the next occurrence of a number (a string of digits) that begins with the digit 1 or 2:

1. Execute **Arg Arg** (ALT+A ALT+A).
2. Type `[12][0-9]*`
3. Execute **Psearch** (F3).

Regular expressions are most powerful when they are used together. For example, the combination of the wildcard (**?**) and repeat (*****) operators

```
?*
```

matches any string of characters. This expression is useful when it is part of a larger expression, such as

```
B?*ing
```

which matches any string beginning with `B` and ending with `ing`.

Non-UNIX Matching Method

The type of non-UNIX matching method is significant only when you use a find-and-replace command. “Matching method” refers to the technique used to match repeated expressions. For example, does the expression `a*` match as few or as many characters as it can? The answer depends on the matching method.

PWB supports two matching methods in non-UNIX regular expressions:

- "Minimal matching" matches as few characters as possible to find a match. For example, `a+` matches only the first character in `aaaaa`. However, `ba+b` matches the entire string `baaaab` because it is necessary to match every `a` to match both occurrences of `b`.
- "Maximal matching" matches as many characters as possible. For example, `a#` matches the entire string `aaaaaa`.

Example

If `a+` (minimal matching plus) is the find string and `EE` is the replacement string, PWB replaces `aaaaa` with `EEEEEEEEEE` because at each occurrence of `a`, PWB immediately replaces it with `EE`.

However, if `a#` (maximal matching plus) is the find string, PWB replaces the same string with `EE` because it matches the entire string `aaaaa` at once and replaces that string with `EE`.

Glossary

386 enhanced mode A mode in the Windows operating system that runs on the 80386 and 80486 processors. It provides access to extended memory and the ability to run non-Windows-based applications. This and standard mode are both referred to as protected mode in the Windows operating system and offer more capability than real mode.

8086 family of processors All processors in the Intel 8086 family, including the 8086, 80286, 80386, and 80486 CPU chips.

8087 family of math processors All math processors (also called math coprocessors) in the Intel 8087 family, including the 8087, 80287, and 80387 chips. These processors perform high-speed floating-point and binary-coded-decimal number processing. The 80486 chip includes a math processor.

8087 window The CodeView window in which the floating-point math processor's registers are displayed. This window remains empty until a math processor instruction is executed. If the program uses the Microsoft math processor emulator library, the contents of the emulated math processor's registers are displayed.

A

actual parameter See "argument."

adapter A printed-circuit card that plugs into a computer and controls a device, such as a video display or a printer.

address The memory location of a data item or procedure, or an expression that evaluates to an address. In CodeView, the expression can represent just the offset (a default segment is assumed), or it can be in *segment*:*offset* format.

address range A range of memory bounded by two addresses.

anonymous allocation Assignment to a segment at link time.

ANSI (American National Standards Institute) The institute responsible for defining programming-language standards to promote portability of languages between different computer systems.

ANSI character set An 8-bit character set that contains 256 characters. See "ASCII character set."

API (application programming interface) A set of system-level routines that can be used in an application program for tasks such as 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 API.

argc The conventional name for the first argument to the **main** function in a C source program (an integer specifying the number of arguments passed to the program from the command line).

argument A value passed to a routine or specified with an option in the command line for a utility. Also called an actual parameter. See also "parameter."

argv The conventional name for the second argument to the **main** function in a C source program (a pointer to an array of strings). The first string is the program name, and each following string is an argument passed to the program from the command line.

array A set of elements of the same type.

ASCII character set The American Standard Code for Information Interchange 8-bit character set, consisting of the first 128 (0 to 127) characters of the ANSI character set. The term ASCII characters is sometimes used to mean all 256 characters defined for a particular system, including the extended ASCII characters. ASCII values represent letters, digits, special symbols, and other characters. See also "extended ASCII."

ASCII file See "text file."

.ASM The extension for an assembly-language source file.

Assembly mode The mode in which CodeView displays the assembly-language equivalent of the machine code being executed. CodeView disassembles the executable file in memory to obtain the code.

automatic data segment See "DGROUP."

available memory The portion of conventional memory not used by system software, TSR utilities, or other programs.

B

.BAK The extension that is often used to indicate a backup file.

.BAS The extension for a Basic language source file.

base name The part of a filename before the extension, usually 1 to 8 characters. For example, README is the base name of the filename README.TXT.

.BAT The extension for an MS-DOS batch file.

batch file A file containing operating-system commands that can be run from the command line. Also called a command file.

binary file A file that contains numbers in binary, machine-readable form. For example, an executable file is a binary file.

binary operator An operator that takes two operands.

BIOS (basic input/output system) The code built into system memory that provides hardware interface routines for programs. You can trace into the BIOS with CodeView when using Assembly mode.

.BMP The extension for a bitmap file.

breakpoint A specified address where program execution halts. CodeView interrupts execution when the program reaches the address where a breakpoint is set. See also "conditional breakpoint."

.BSC The extension for a database file for use with the Source Browser. A .BSC file is created by BSCMAKE.

buffer An area in memory that holds data temporarily, most often during input/output operations.

C

.C The extension for a C source file.

call gate A special descriptor-table entry that describes a subroutine entry point rather than a memory segment. A far call to a call gate selector transfers to the entry point specified in the call gate. This is a feature of the 80286–80486 hardware and is typically used to provide a transition from a lower privilege state to a higher one.

case sensitivity The distinction made between uppercase and lowercase letters. For example, “MyFile” and “MYFILE” are considered to be different strings in a case-sensitive situation but are understood to be the same string if case is not sensitive.

CGA (color graphics adapter) A video adapter capable of displaying text characters or graphics pixels in low resolution in up to 16 colors.

character string A sequence of bytes treated as a set of ASCII letters, numbers, and other symbols. A character string is often enclosed in single quotation marks (`' '`) or double quotation marks (`" "`).

child process A process created by another process (its parent process).

click To press and release quickly one of the mouse buttons (usually the left button) while pointing the mouse pointer to an object on the screen.

clipboard A temporary storage area for text. The clipboard is used for cut, copy, and paste operations.

.COB The extension for a COBOL source file.

code symbol The address of a routine.

.COM The extension for an MS-DOS executable file that contains a single segment. Tiny-model programs have a .COM extension. See also “tiny memory model.”

command An instruction you use to control a computer program, such as an operating system or application.

command file A file containing operating-system commands that can be run from the command line. If the file’s extension is .BAT, the command file contains MS-DOS commands. Also called a batch file.

command file (in NMAKE) A text file containing input expected by utilities such as NMAKE.

compact memory model A program with one code segment and multiple data segments.

compile To translate programming language statements into a form that can be executed by the computer.

conditional breakpoint A breakpoint that is taken when a specified expression becomes nonzero (true). A conditional breakpoint is evaluated after every instruction is executed unless an address is also specified. Formerly called tracepoint and watchpoint.

constant A value that does not change during program execution.

constant expression Any expression that evaluates to a constant. It may include integer constants, character constants, floating-point constants, enumeration constants, type casts to integral and floating-point types, and other constant expressions. It cannot include a variable or function call.

conventional memory The first 640K (or sometimes 1MB) of memory under MS-DOS. Also called low memory.

coprocessor See “8087 family of math processors.”

.CPP An extension for a C++ source file.

CPU (central processing unit) The main processor in a computer. For example, the CPU that receives and carries out instructions in the PC/AT is an 80286 processor. See also “8086 family of processors.”

CS:IP The address of the current program location. This is the address of the next instruction to be executed. CS is the value of the Code Segment register, and IP is the value of the Instruction Pointer register.

cursor The thin blinking line or other character that represents the location of typed input or mouse activity.

.CXX An extension for a C++ source file.

D

.DAT The extension that is often used to indicate a data file.

data symbol The address of a global or static data object. The concept of data symbol includes all data objects except local (stack-allocated) or dynamically allocated data.

.DBG The extension for a file that is created by LINK when the /CO and /TINY options are used. The file contains symbolic debugging information.

debugger A program that allows the programmer to execute a program one line or instruction at a time. The debugger displays the contents of registers and memory to help locate the source of problems in the program. An example is the Microsoft CodeView debugger.

debugging information Symbolic information used by a debugger, especially information in the Microsoft Symbolic Debugging Information format that is used by the Microsoft CodeView debugger.

.DEF The extension for a module-definition file.

default data segment See "DGROUP."

default library A standard library that contains routines and data for a language. The language's compiler embeds the name of the default library in the object file in a COMMENT record. The embedded name tells LINK to search the default library automatically.

DGROUP The group that contains the segments called _DATA (initialized data), CONST (constant data), _BSS (uninitialized data), and STACK (the program's stack). Also called default (or automatic) data segment.

dialog box A box that appears when you choose a command that requires additional information.

disassemble To translate binary machine code into the equivalent assembly-language representation. Also called unassemble.

disassembly The assembly-language representation of machine code, obtained by disassembling the machine code.

.DLL The extension for a dynamic-link library.

DLL A dynamic-link library.

.DOC The extension that is often used to indicate a document file.

DOS application A program that runs only with MS-DOS. An MS-DOS executable file contains a header and one contiguous block of segments.

DOS-extended An application that is able to be run by the DOS Extender in extended or expanded memory.

DOS Extender A program that lets an application run in extended or expanded memory.

DOS session Under the Windows operating system, a full-screen emulation of the MS-DOS environment started using the MS-DOS Prompt in the Program Manager Main Group. The MS-DOS Prompt program item starts a copy of the MS-DOS command interpreter (COMMAND.COM).

double precision A real (floating-point) numeric value that occupies 8 bytes of memory. Double-precision values are accurate to 15 or 16 digits.

DPMI (DOS Protected Mode Interface)
A server that provides extended or expanded memory. An example of a DPMI server is an MS-DOS session in Windows.

drag To move the mouse while holding down one of its buttons.

dump To display the contents of memory at a specified memory location.

dynamic link A method of postponing the resolution of external references until load time or run time. A dynamic link allows the called routines to be created, distributed, and maintained independently of their callers.

dynamic-link library A file, usually with a .DLL extension, that contains the binary code for routines and data that are linked to a program at run time.

E

EGA (enhanced graphics adapter)
A video adapter capable of displaying all the modes of the color graphics adapter (CGA) plus additional modes in medium resolution in up to 64 colors.

EMM386.EXE An example of a VCPI server. EMM386.EXE simulates expanded memory in extended memory for an 80386 or higher processor.

EMS Expanded Memory Specification. See "expanded memory."

emulator A floating-point math package that provides software emulation of the operations of a math processor.

environment strings A series of user-definable and program-definable strings associated with each process. The initial values of environment strings are established by a process's parent.

environment table The memory area, defined by the operating system, that stores environment variables and their values.

environment variable A string associated with an identifier and stored by the operating system. Environment variables are defined by the SET command. The identifier and the string associated with it can be used by a program.

.ERR The extension for a file of error-message text or error output.

error code See "exit code."

escape sequence A specific combination of an escape character (often a backslash) followed by a character, keyword, or code. Escape sequences often represent white space, nongraphic characters, or literal delimiters within strings and character constants.

.EXE One of the extensions for an executable file, which is a file that can be loaded and executed by the operating system.

executable file A program ready to be run by an operating system, usually with one of the extensions .EXE, .COM, or .BAT. When the name of the file is typed at the system prompt, the statements in the file are executed.

exit code An integer returned by a program to the operating system or the program's caller after completion to indicate the success, failure, or status of the program. Also called a return code or error code.

Exit code also refers to the executable code that a compiler places in every program to terminate execution of the program. This code typically closes open files and performs other housekeeping chores. When a program

terminates in CodeView, the current line is in the exit code. No source code is shown since none is available. See also “startup code.”

expanded memory Memory above 640K made available to real-mode programs and controlled through paging by an expanded memory manager.

expanded memory emulator A device driver that allows extended memory on computers with an 80286 or later processor to behave like expanded memory.

expanded memory manager (EMM) A device driver for controlling expanded memory.

explicit allocation Assignment to a segment at compile time.

expression A combination of operands and operators that yields a single value.

extended ASCII ASCII codes between 128 and 255. The meanings of extended ASCII codes differ depending on the system.

extended dictionary A summary of the definitions contained in all modules of a standard library. LINK uses extended dictionaries to search libraries faster.

extended memory Memory above either 640K or 1 megabyte made available to protected-mode programs on computers with an 80286 or later processor. Extended memory is used by the Windows operating system in standard mode or 386 enhanced mode.

extended memory manager A device driver for controlling extended memory, for example, HIMEM.SYS for the Windows operating system.

extender-ready See “DOS-extended.”

extension One, two, or three characters that appear after a period (.) following the base name in a filename. For example, .TXT is the extension of the filename README.TXT. A filename does not necessarily have an extension. Sometimes the extension is considered to include the preceding period.

external reference A routine or data item declared in one module and referenced in another.

F

far address A memory location specified by using a segment (location of a 64K block) and an offset from the beginning of the segment. Far addresses require 4 bytes—2 for the segment and 2 for the offset. Also called a segmented address. See also “address” and “near address.”

FAT (file allocation table) The standard file system for MS-DOS.

fatal error An error that causes a program to terminate immediately.

.FD The extension for a declaration file (a type of include file) in FORTRAN.

.FI The extension for an interface file (a type of include file) in FORTRAN.

file handle A value returned by the operating system when a file is opened and used by a program to refer to the file when communicating to the system. Under MS-DOS, COMMAND.COM opens the first five file handles as **stdin**, **stdout**, **stderr**, **stdaux**, and **stdprn**.

filename A string of characters identifying a file on disk, composed of a base name optionally followed by a period (.) and an extension. A filename may be preceded by a path. For example, in the filename README.TXT, .TXT is the extension and README is the base name.

fixup The linking process that resolves a reference to a relocatable or external address.

flags register A register that contains individual bits, each of which signals a condition that can be tested by a machine-level instruction. In other registers, the contents of the register are considered as a whole, while in the flags register only the individual bits have meaning. In CodeView, the current values of the most commonly used bits of the flags register are shown at the bottom of the Register window.

flat memory model A nonsegmented memory model that can address up to 4 gigabytes of memory.

flipping A screen-exchange method that uses the video pages of the CGA or EGA to store both the debugging and output screens. When you request the other screen, the two video pages are exchanged. See also "screen exchange" and "swapping."

.FOR The extension for a FORTRAN source file.

formal parameter See "parameter."

frame The segment, group, or segment register that specifies the segment part of an address.

full-screen application A program that runs with Windows but cannot run in a window.

function A routine that returns a value.

function call An expression that invokes a function and passes arguments (if any) to the function.

G

gigabyte (GB) 1024 megabytes or 1,073,741,824 bytes (2 to the power of 30).

global symbol A symbol that is available throughout the entire program. In CodeView, function names are always global symbols. See also "local symbol."

grandparent process The parent of a process that created a process.

group A collection of segments having the same segment base address.

H

.H The extension for an include (or header) file in C.

HELPFILES The environment variable used by a program to find .HLP files.

hexadecimal The base-16 numbering system whose digits are 0 through F. The letters A through F represent the decimal numbers 10 through 15. Hexadecimal is easily converted to and from binary, the base-2 numbering system the computer itself uses.

highlight To select an area in a text box, window, or menu as a command or as text to be deleted or copied. A highlight is shown in reverse-video or a bright color.

high memory Memory between the 640K of conventional memory and the 1MB limit of a PC's address space.

HIMEM.SYS An example of an XMS server. HIMEM.SYS manages extended memory for an 80286 or higher processor.

.HLP The extension for a help file created by HELPMAKE.

HPFS (high-performance file system) An installable file system that uses disk caching and that allows filenames to be longer and to contain certain nonalphanumeric characters.

huge memory model A program with multiple

code and data segments, and data items that can exceed 64K.

I

identifier A name that identifies a register or a location in memory and usually represents a program element such as a constant, variable, type, or routine. The terms identifier and symbol are used synonymously in most documentation.

IEEE format (Institute for Electrical and Electronic Engineers)
A standard for representing floating-point numbers, performing math with them, and handling underflow/overflow conditions. The 8087 family of math processors and the Microsoft emulator library implement this format.

import library A library, created by IMPLIB, that contains entry points in DLLs. It does not contain the actual code for routines and data. An import library is used to resolve references at link time in the same way as a standard library; each is a type of static library. See "dynamic-link library" and "standard library."

.INC The extension for an include file in Microsoft Macro Assembler.

include file A file that is merged into a program's source code with a file-inclusion command. In C, this command is the **#include** preprocessor directive. In FORTRAN, it is the **INCLUDE** keyword or the **$INCLUDE** metacommand. In Microsoft Macro Assembler, the equivalent command is the **INCLUDE** directive. In a .DEF file, the **INCLUDE** statement performs this action. In an NMAKE makefile, it is the **!INCLUDE** directive.

.INI The extension for an initialization file.

INIT The environment variable usually used by a program to find an initialization file.

installable file system A file system that exists in addition to the FAT file system.

integer In CodeView and the C language, a whole number represented as a 16-bit two's complement binary number that has a range from –32,768 through +32,767. See also "long integer."

interoverlay call A call from a function in one overlay to a function in another overlay, represented internally by an entry in a thunk table.

interrupt call A machine-level procedure that can be called to execute a BIOS, DOS, or other function. You can trace into BIOS interrupt-service routines with CodeView, but not into the DOS interrupt (0x21).

I/O privilege mechanism A facility that allows a process to ask a device driver for direct access to the device's I/O ports and any dedicated or mapped memory locations it has. The I/O privilege mechanism can be used directly by an application or indirectly by a dynamic-link library.

K

kilobyte (K) 1024 bytes (2 to the power of 10).

L

label An identifier representing an address.

large memory model A program with multiple code and data segments.

LIB The environment variable used by LINK to find default libraries.

.LIB The extension for a static library.

library A collection of routines or data made available to one or more programs through static or dynamic linking.

LIM EMS Lotus/Intel/Microsoft Expanded Memory Specification.

LINK The environment variable used by LINK for command-line options.

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.

list file A text file of information produced by a utility such as LIB. See "map file."

listing A generic term for a map, list, or cross-reference file.

.LNK The extension that is often used to indicate a response file.

load library A static library specified to the linker as an object file, causing all modules in the library to be linked into the program. See "static library."

local symbol An identifier that is visible only within a particular routine. See "global symbol."

Local window The CodeView window in which the local variables for the current routine are displayed.

logical segment A segment defined in an object module. Each physical segment other than DGROUP contains exactly one logical segment, except when you use the **GROUP** directive in a Microsoft Macro Assembler module. (Linking with the /PACKC option can also create more than one logical segment per physical segment.)

long integer In CodeView and the C language, a whole number represented by a 32-bit two's complement value. Long integers have a range of –2,147,483,648 to +2,147,483,647. See "integer."

low memory See "conventional memory."

.LRF The extension that is often used to indicate a response file. PWB creates response files with the .LRF extension.

.LST The extension that is often used to indicate a list file.

l-value An expression (such as a variable name) that refers to a single memory location and is required as the left operand of an assignment operation or the single operand of a unary operator. For example, X1 is an l-value, but `X1+X2` is not.

M

machine code A series of binary numbers that a processor executes as program instructions. See also "disassemble."

macro A block of text or instructions that has been assigned an identifier. For example, you can create a macro that contains a set of functions that you perform repeatedly and assign the macro to a single keystroke.

.MAK The extension that is often used to indicate a makefile or description file.

.MAP The extension for a map file.

map file A text file of information produced by a utility such as LINK. Also called a list file or listing.

math coprocessor See "8087 family of math processors."

MB Megabyte.

MDI Multiple Document Interface.

medium memory model A program with multiple code segments and one data segment.

megabyte (MB) 1024 kilobytes or 1,048,576 bytes (2 to the power of 20).

memory model A convention for specifying the number of code and data segments in a program. Memory models include tiny, small, medium, compact, large, huge, and flat.

memory-resident program See “terminate-and-stay-resident.”

menu bar The bar at the top of a display containing menus.

Mixed mode The CodeView source display mode that shows each source line of the program being debugged, followed by a disassembly of the machine code generated for that source line. This mode combines Source mode and Assembly mode.

modification time See “time stamp.”

module A discrete group of statements. Every program has at least one module (the main module). In most cases, each module corresponds to one source file.

module (in LIB) See “object module.”

module-definition file A text file, usually with a .DEF extension, that describes characteristics of a program. A module-definition file is used by LINK and by IMPLIB.

monochrome adapter A video adapter capable of displaying only in medium resolution in one color. Most monochrome adapters display text only; individual graphics pixels cannot be displayed.

mouse pointer The reverse-video or colored square that moves to indicate the current position of the mouse. The mouse pointer appears only if a mouse is installed.

MS32EM87.DLL A DLL required by the DOS Extender. The SYSTEM environment variable must be set to the directory that contains this file.

MS32KRNL.DLL A DLL required by the DOS Extender. The SYSTEM environment variable must be set to the directory that contains this file.

multitasking operating system An operating system in which two or more programs or threads can execute simultaneously.

N

NAN An acronym for “not a number.” The math processors generate NANs when the result of an operation cannot be represented in IEEE format.

near address A memory location specified by only the offset from the start of the segment. A near address requires only two bytes. See also “address” and “far address.”

newline character The character used to mark the end of a line in a text file, or the escape sequence (\n in C language) used to represent this character.

null character The ASCII character whose value is 0, or the escape sequence (\0 in C language) used to represent this character.

null pointer A pointer to nothing, expressed as the integer value 0.

O

.OBJ The extension for an object file produced by a compiler or assembler.

object file A file produced by compiling or assembling source code, containing relocatable machine code.

object module A group of routines and data items stored in a standard library, originating from an object file. See also “standard library.”

object module format The specification for the structure of object files. Microsoft languages conform to the Microsoft Relocatable Object-Module Format (OMF), which is based on the Intel 8086 OMF.

offset The number of bytes from the beginning

of a segment or other address to a particular byte.

OMF Object module format.

Output screen The CodeView screen that contains program output. To switch to this screen, choose the Output command from the View menu or press F4.

overlay A program component loaded into memory only when needed.

P

packaged function A function that exists in an object file as a COMDAT record. Packaged functions allow function-level linking. Functions that are not packaged can be linked only at the object level.

parameter A data item expected by a routine or information expected in the command line for a utility. Also called a formal parameter. See also "argument."

parent process A process that creates another process, called the child process.

.PAS The extension for a Pascal source file.

path A specification of the location of a file or a directory. A path consists of one or more directory names and may include a drive (or device) specification. For example, C:\PROJECT\PROJLIBS is the path to a subdirectory called PROJLIBS in a directory called PROJECT that is located on the C drive. Sometimes "path" refers to multiple path specifications, each separated by a semicolon (;). In certain circumstances, a path specification must include a trailing backslash; for example, specify C:\PROJECT\PROJLIBS\ to tell LINK the location of the PROJLIBS directory containing additional libraries.

.PCH The extension for a precompiled C header (or include) file.

physical segment A segment listed in the executable file's segment table. Each physical segment has a distinct segment address, whereas logical segments may share a segment address. A physical segment usually contains one logical segment, but it can contain more.

PID (process identification number) A unique code that the operating system assigns to a process when the process is created. The PID may be any value except 0.

pointer A variable containing an address or offset.

pop-up menu A menu that appears when you click the menu title with the mouse or press the ALT key and the first letter of the menu at the same time.

port The electrical connection through which the computer sends and receives data to and from devices or other computers.

precedence The relative position of an operator in the hierarchy that determines the order in which expressions are evaluated.

privileged mode A special execution mode (also known as ring 0) supported by the 80286–80486 hardware. Code executing in this mode can execute restricted instructions that are used to manipulate system structures and tables. Device drivers run in this mode.

procedure A routine that does not return a value.

procedure call A call to a routine that performs a specific action.

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.

processor See "CPU (central processing unit)."

program step To trace the next source line in Source mode or the next instruction in Mixed

mode or Assembly mode. If the source line or instruction contains a function, procedure, or interrupt call, the call is executed to the end and the CodeView debugger is ready to execute the instruction after the call. See also “trace.”

protected mode The operating mode of the 80286–80486 processors that allows the operating system to protect one application from another.

protected mode (in Windows) Either of two modes in the Windows operating system version 3.0: standard mode or 386 enhanced mode. See also “standard mode” and “386 enhanced mode.”

Q

.QLB The extension for a Quick library.

R

radix The base of a number system. In CodeView, numbers can be entered in three radixes: 8 (octal), 10 (decimal), or 16 (hexadecimal). The default radix is 10.

RAM Random access memory. Usually refers to conventional memory.

.RC The extension for a resource script file. An .RC file defines resources for an application such as icons, cursors, menus, and dialog boxes. The Microsoft Windows Resource Compiler compiles an .RC file to create an .RES file.

real mode The operating mode of the 80286–80486 processors that runs programs designed for the 8086/8088 processor. All programs for MS-DOS run in real mode.

real mode (in the Windows operating system) An operating mode that provides compatibility with versions of Windows-based applications prior to 3.0. Real mode is the only mode of the Windows operating system version 3.0 available for computers with less than 1 megabyte of extended memory.

redirection The process of causing a command or program to take its input from a file or device other than the keyboard (standard input), or causing the output of a command or program to be sent to a file or device other than the display (standard output). The operating-system redirection symbols are the greater-than (>) and less-than (<) signs.

The same symbols are used in the CodeView Command window to redirect input and output of the debugging session. In addition, the equal sign (=) can be used to redirect both input and output.

Register window The CodeView window in which the CPU registers and the bits of the flags register are displayed.

registers Memory locations in the processor that temporarily store data, addresses, and logical values. See also “flags register.”

regular expression A text expression that specifies a pattern of text to be matched (as opposed to matching specific characters). CodeView supports a subset of the regular-expression characters used in the XENIX and UNIX operating systems. PWB supports both the full UNIX syntax and an extended Microsoft syntax for regular expressions.

relocatable Not having an absolute address.

.RES The extension for a file produced by the Microsoft Windows Resource Compiler from an .RC file.

response file A text file containing input expected by utilities such as LINK and LIB. Commonly used extensions for response files include .LRF, .LNK, and .RSP.

return code

See “exit code.”

ROM Read-only memory.

root In an overlaid MS-DOS program, the part of the program that always remains in memory. Also called the root overlay.

routine A generic term for a procedure, function, or subroutine.

.RSP The extension that is often used to indicate a response file.

RTF Rich text format.

run-time error A math or logic error that occurs during execution of a program. A run-time error often results in termination of the program.

S

.SBR The extension for a file used by BSCMAKE to create a .BSC file.

scope The parts of a program in which a given symbol has meaning. The scope of an item may be limited to the file, function, block, or function prototype in which it appears.

screen exchange The method by which both the output screen and the debugging screen are kept in memory so that both can be updated simultane-ously and either can be viewed at the user's convenience. The two screen-exchange modes are flipping and swapping. See also "flipping" and "swapping."

scroll To move text up, down, left, and right in order to see parts that cannot fit on the screen.

segment A section of memory containing code or data, limited to 64K for 16-bit segments or 4 gigabytes for 32-bit segments. Also refers to the starting address of that memory area.

segmented-executable file The executable file format of a Windows-based application or DLL. A segmented-executable file contains an MS-DOS header, a new .EXE header, and multiple relocatable segments.

semaphore A software flag or signal used to coordinate the activities of two or more threads. A semaphore is commonly used to protect a critical section.

shell To gain access to the operating-system command line without actually leaving the PWB or CodeView environment or losing the current context. You can execute operating-system commands and then return to the environment.

single precision A real (floating-point) value that occupies 4 bytes of memory. Single-precision values are accurate to six or seven decimal places.

small memory model A program with one code segment and one data segment.

SMARTDRV.EXE A driver that creates a disk cache in extended or expanded memory.

source file A text file containing the high-level description that defines a program.

Source mode The mode in which CodeView displays the source code that corresponds to the machine code being executed.

stack A dynamically expanding and shrinking area of memory in which data items are stored in consecutive order and removed on a last-in, first-out basis. The stack is most commonly used to store information for function and procedure calls and for local variables.

stack frame A portion of a program's stack that contains a routine's local and temporary variables, arguments, and control information.

stack trace A symbolic representation of the functions that have been executed to reach the current instruction address. As a function is executed, the function address and any function arguments are pushed on the stack. A trace of the stack shows the currently active functions and the values of their arguments. See also "stack frame."

standard error The device to which a program sends error messages. COMMAND.COM opens standard error with a file handle named **stderr**. The default device is the display (CON). Standard error cannot be redirected.

standard input The device from which a program reads input. COMMAND.COM opens standard input with a file handle named **stdin**. The default device is the keyboard (CON). Standard input can be redirected using a redirection symbol (<).

standard library A library created by LIB that contains compiled routines and data. It is used to resolve references at link time.

standard mode The normal Windows version 3.0 operating mode that runs on the 80286–80486 processors. This and 386 enhanced mode are both referred to as protected mode in the Windows operating system and offer more capability than real mode.

standard output The device to which a program sends output. COMMAND.COM opens standard output with a file handle named **stdout**. The default device is the display (CON). Standard output can be redirected using a redirection symbol.

startup code The code placed at the beginning of a program to control execution of the program code. When CodeView is loaded, the first source line executed runs the entire startup code. If you switch to Assembly mode before executing any code, you can trace through the startup code. See also “exit code.”

static library A library used for resolving references at link time. A static library can be either a standard library or an import library. See also “standard library” and “import library.”

static linking The combining of multiple object and library files into a single executable file with all external references resolved at link time.

status bar The bar at the bottom of the CodeView or PWB display containing status information and command buttons or a short description of the dialog or menu item currently displayed.

stderr See “standard error.”

stdin See “standard input.”

stdout See “standard output.”

string A contiguous sequence of characters, often identified by a symbolic name as a constant or variable.

structure A set of elements which may be of different types, grouped under a single name. See also “user-defined type.”

structure member One of the elements of a structure.

stub file An MS-DOS executable file added to the beginning of a segmented executable file. The stub is invoked if the file is executed with MS-DOS.

subroutine A unit of FORTRAN code terminated by the **RETURN** statement. Program control is transferred to a subroutine with a **CALL** statement.

swapping A screen-exchange method that uses buffers to store the CodeView display and program output screens. When you request the other screen, the two buffers are exchanged. See also “flipping” and “screen exchange.”

symbol See “identifier.”

symbolic debugging information See “debugging information.”

.SYS The extension for a system file or device driver.

SYSTEM An environment variable used by the DOS Extender to find the files MS32EM87.DLL and MS32KRNL.DLL.

T

TEMP The environment variable usually used by a program to find the directory in which to create temporary files. Other programs use the TMP variable in a similar way.

temporary file A file that is created for use by a command while it is running. The file is usually deleted when the command is completed. Most programs create temporary files in the directory indicated by the TMP or TEMP environment variable.

terminate-and-stay-resident (TSR)
An MS-DOS program that remains in memory and is ready to respond to an interrupt.

ternary operator An operator that takes three operands. For example, the C-language **?** operator.

text file A file containing only ASCII characters in the range of 1 to 127.

thread An operating-system mechanism that allows more than one path of execution through the same instance of a program.

thread ID The name or handle of a particular thread within a process.

thread of execution The sequence of instructions executed by the CPU in a single logical stream. In MS-DOS, there is only one thread of execution.

thunk An interoverlay call in an overlaid MS-DOS program.

time stamp The time of the last write operation to the file. Sometimes the term time stamp refers to the combination of the date and time of the last write operation. Also called modification time.

tiny memory model A program with a single segment holding both code and data, limited to 64K, with the extension .COM.

TMP The environment variable usually used by a program to find the directory in which to create temporary files. Other programs use the TEMP variable in a similar way.

.TMP The extension that is often used to indicate a temporary file.

toggle A feature with two states. Often used to describe a command that turns a feature on if it is off, and off if it is on. When used as a verb, “toggle” means to reverse the state of a feature.

TOOLS.INI A file that contains initialization information for Microsoft tools such as PWB, CodeView, and NMAKE.

trace To execute a single line or instruction. The next source line is traced in Source mode and the next instruction is traced in Assembly mode. If the source line or instruction contains a function, procedure, or interrupt call, the first source line or instruction of the call is executed. CodeView is ready to execute the next instruction inside the call. See also “program step.”

tracepoint (obsolete) A breakpoint that is taken when an expression, variable, or range of memory changes. This is now a type of conditional breakpoint. See also “conditional breakpoint.”

TSR See “terminate-and-stay-resident.”

.TXT The extension for a text file.

type cast An operation in which a value of one

type is converted to a value of a different type.

type casting Including a type specifier in parentheses in front of an expression to indicate the type of the expression's value.

U

unary operator An operator that takes a single operand.

unassemble To translate binary machine code into the equivalent assembly-language representation. Also called disassemble.

unresolved external 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.

user-defined type A data type defined by the user. See also "structure."

V

variable A value that may change during program execution.

VCPI Virtual Control Program Interface

VCPI server A server that provides expanded memory. An example of a VCPI server is Microsoft's EMM386.EXE.

VGA (video graphics adapter) A video adapter capable of displaying both text and graphics at medium to high resolution in up to 256 colors.

virtual memory A memory management system that provides more memory to a program than is actually in the system. Virtual memory can consist of a file on disk, extended memory, or expanded memory.

W

watchpoint (obsolete) A breakpoint that is taken when an expression becomes true (nonzero). This is now a type of conditional breakpoint. See also "conditional breakpoint."

wildcard A character that represents one or more matching characters. MS-DOS wildcards (***** and **?**) in a filename specification are expanded by COMMAND.COM.

Windows-based application A program that runs only with the Windows operating system.

X

XMS Extended Memory Standard (or Specification). See “extended memory.”

XMS server A server that provides extended memory. An example of an XMS server is Microsoft’s HIMEM.SYS.

Index

! (exclamation point)
HELPMAKE command 602
makefile syntax 544, 572
replacing text, PWB 86
Shell Escape command, CodeView 399, 443–445
! command, CodeView 398, 443–445
" (double quotation marks)
.DEF file syntax 494
LINK syntax 461
makefile syntax 537, 553
" (quotation marks)
character strings 859
CodeView syntax 312–313
Pause command, CodeView 400, 445
(number sign)
custom builds 53
HELPMAKE syntax 596–597
makefile syntax 536, 551–552, 564
Tab Set command, CodeView 400, 445
TOOLS.INI syntax 534
$ (dollar sign)
end of line, regular expression syntax 845, 847, 849, 854
makefile syntax 536, 552, 554–556, 560, 562
reference to tagged expression 848, 855
$ macros (NMAKE) 555
% (percent sign)
Filename-Parts Syntax, PWB 247
makefile syntax 536
& (ampersand)
C address operator 391–392
CodeView 381
line continuation character, LIB 586
' (single quotation marks), LINK syntax 461
( ) (parentheses)
balancing, PWB 180–181
makefile syntax 554, 556, 560
(brackets), character class 849, 854
* (asterisk)
Comment command, CodeView 400, 446
Copy command, LIB 589
deleting watch expressions, CodeView 437
hyperlink, Microsoft Advisor 672
makefile syntax 555–556
match character, regular expression syntax 847
regular expressions, PWB 85
wildcard operator
HELPMAKE syntax 595
UNDEL syntax 655
* (asterisk) (*continued*)
wildcards 536
+ (plus sign)
concatenating help files 680
LIB syntax 587–588
LINK syntax 461, 463, 469–470
searching, PWB 84
, (comma)
argument separator, CodeView 326–327
CodeView operator 381
LIB syntax 582–583
LINK syntax 460, 470
with context operator, CodeView 397–398
- (dash)
character classes, PWB 84
character ranges, regular expression syntax 845
HELPMAKE options 595
LIB syntax 584, 588
LINK syntax 471
makefile syntax 544, 572
NMAKE syntax 529–532
-*, LIB syntax 589
-+, LIB syntax 589
. (period)
Current Location command, CodeView 400, 446
line number specifier, CodeView 339
LINK syntax 461, 466, 469
makefile syntax 564, 570
match character 844
wildcard character 845, 847–848, 850
... (ellipsis)
call tree, PWB 92
menu commands, PWB 72, 74, 116
/ (forward slash), LIB syntax 584
/ (slash)
EXEHDR syntax 514
HELPMAKE options 595
LINK syntax 463, 471
makefile syntax 535, 538, 543, 551–552
NMAKE syntax 529
Search command, CodeView 335, 336, 400, 447–48
/? option
BSCMAKE 621
CVPACK 633
EXEHDR 515
EXP 656
HELPMAKE 599

IMPLIB 654
LIB 586
/? option (*continued*)
LINK 488
NMAKE 532
RM 655
SBRPACK 625
UNDEL 656
/2 option,CodeView 310
/25 option, CodeView 310–311
/43 option, CodeView 310–311
/50 option, CodeView 311
/8 option, CodeView 310
: (colon)
CodeView operator 381
.DEF file syntax 494
Delay command, CodeView 400, 450
HELPMAKE commands 605
LINK syntax 471
makefile syntax 538–542, 560, 564, 570
:: (scope operator), CodeView precedence 382
:> (base operator), CodeView precedence 382
:> command, HELPMAKE 607
; (semicolon)
comments, PWB 126–127
.DEF file syntax 494
LIB syntax 582–584
LINK syntax 461, 467–470
makefile syntax 542, 544, 564
TOOLS.INI syntax 534
< (less than operator), Redirect Input command, CodeView 312–313, 450
< > (angle brackets)
command buttons, PWB 72, 74
makefile syntax 547–548, 555
= (equal sign)
.DEF file syntax 494
makefile syntax 551, 560
Redirect Input and Output command, CodeView 400, 452
> (DOS redirection symbol), HELPMAKE syntax 597
> (greater than operator), Redirect Output command, CodeView 312–313, 400, 450–451
> > (help delimiter), HELPMAKE 609–610, 612
? (question mark)
call tree, PWB 91
decorated names, C++ 385–386
Display Expression command, CodeView 400, 452–453
makefile syntax 555–556
wildcard operator
HELPMAKE syntax 595
regular expression syntax 847 854–855
UNDEL 655
wildcards 536
?: (conditional operator), CodeView 381
?? command, CodeView 400, 453–454
@ (at sign)
BSCMAKE syntax 622
LIB syntax 583
LINK syntax 470
local contexts, HELPMAKE 603–604
makefile syntax 544–556
naming registers, CodeView 377, 395
NMAKE syntax 533
Redraw command, CodeView 400, 454
@ command, CodeView 400, 454
[ ] (brackets)
balancing, PWB 180–181
call tree, PWB 92
character class 845, 847, 849, 854
match character 844
\ (backslash)
escape 847–848, 854
HELPMAKE syntax 604–605
line continuation character, PWB 105–107, 126
match character 847–848, 854
regular expressions, PWB 88
Screen Exchange command, CodeView 400, 454
\\\ formatting attribute, HELPMAKE 604–605
^ (caret)
character ranges 847
makefile syntax 535, 551–552, 561
regular expressions syntax
line beginning 845, 847, 849, 854
match line beginning 847, 849, 854
PWB 84, 87
_ (underscore), symbol format, CodeView 385–386
32-Bit Registers command, CodeView 343, 346
386 enhanced mode defined 857
7 command, CodeView 400, 448–449
8086
instruction mnemonics, assembling 400–402
processors defined 857
8087
command, CodeView 347–348, 400, 448–449
processors defined 855
window, CodeView
defined 857
function 330
opening 348
overview 322
8259, interrupt trapping 314
8514 display, specifying, CodeView 311

A

a, alphanumeric character, predefined expression syntax 846, 848, 853
A command, and Restart command, CodeView 412–413
A command, CodeView 398, 400–402
\a formatting attribute, HELPMAKE 602, 604–605
/A option
    LINK 472
    NMAKE 530
About command
    CodeView 349–350
    PWB 70
/Ac option, HELPMAKE 596
Access control, CodeView 386
Activating windows, PWB 244
Actual parameters *See* Arguments
Adapter defined 857
Add command, LIB 587–588
Add Watch command, CodeView 338–339, 399
Add Watch dialog box, CodeView 339
Add Watch Expression command, CodeView 436
Adding
    breakpoints, CodeView 341–342
    commands, PWB Run menu 115, 117
    custom sections, PWB 53
    files, PWB 38–39, 41, 44
    Program Item, PWB 58
    watch expressions, CodeView 339, 436
Address ranges
    CodeView expressions 379, 396–397
    defined 857
ADDRESS, /MAP option 478
Addresses
    CodeView expressions 378, 396
    defined 857
    variables, debugging asembly code 391
AH register, CodeView syntax 395, 426
AHelp, Help files 219–220
AL register, CodeView syntax 395, 426
/ALIGN option LINK 472
/ALIGNMENT option LINK 472
Aligning tabs, PWB 276
Alignment, EXEHDR 522
All Files command, PWB 64
All Windows command, PWB 69
alloc_text pragma 509
Alphabetic characters, predefined expression syntax 846, 848, 853
Alphanumeric characters, predefined expression syntax 846, 848, 853
Alternation, regular expression syntax 846–847, 849, 854
ALTGR key, enabling 257
Ampersand (&)
    C address operator 391–392
    CodeView 381
    line continuation character, LIB 586

Angle brackets (< >)
command buttons, PWB 72, 74
makefile syntax 547–548, 555
Animate command, CodeView 336–337, 344, 398, 408, 429
Anonymous allocation defined 857
ANSI
defined 857
escape sequence, CodeView expressions 385
API defined 857
Application programming interface defined 855
Application type
determining EXEHDR 523
specifying
EXEHDR 514
NAME statement 495
/PM option, LINK 485
Windows *See* Windows, programs for
APPLOADER statement 498
Archives, in libraries, LIB 581
Arg function, PWB 140, 146–147
executing 96–98
getting help 670
replacing text 86
argc defined 857
Arguments
CodeView
entering 326
format 326–327
setting 337–338
command line 309
defined 857
functions, PWB 146–147
argv defined 857
Arrange command
CodeView 347–348
PWB 69, 135
Arrangewindow function, PWB 140, 147
Arrays
debugging assembly language 392
defined 858
expanding and contracting, CodeView 342, 453–454
AS macro (NMAKE) 558
ASCII
characters defined 858
HELPMAKE format 612
memory format, CodeView 331
Askexit switch, PWB 244, 248
Askrtn switch, PWB 244, 249
.ASM files defined 858
Assemble command
and Restart command 412–413
CodeView 398, 400–402
Assembler, changing options, PWB 47, 49–50
Assembling 8086 instruction mnemonics 400–402
Assembly language, debugging 389–392
Assembly mode defined 858
Assign function
executing 98
key assignment, changing 111
PWB 140, 147–149
switch settings, changing 114
Asterisk (*)
Comment command, CodeView 400, 446
deleting watch expressions, CodeView 437
hyperlink, Microsoft Advisor 672
LIB syntax 589
makefile syntax 555–556
match character 844
regular expressions, PWB 85
wildcard operator
HELPMAKE syntax 595
UNDEL syntax 655
wildcards 536
At sign (@)
BSCMAKE syntax 622
LIB syntax 583
LINK syntax 470
local contexts, HELPMAKE 603–604
makefile syntax 544, 555–556
naming registers, CodeView 377, 395
NMAKE syntax 533
Redraw command, CodeView 400, 454
Attributes
formatting, HELPMAKE 602, 604–605
segment *See* Code segments; Data segments; Segments
Auto option, Language command, CodeView 344
AUTOEXEC.BAT
HELPFILES environment variable 679
PWB configuration 127
Autoload switch, PWB 244, 249–250
Automatic data segment *See* DGROUP
Autosave switch, PWB 112, 244, 250
Autostart entry, TOOLS.INI file, CodeView 302
Autostart macro, PWB 210
Available memory defined 858
AX register, CodeView syntax 395, 426

B

\b formatting attribute, HELPMAKE 603–605
\b formatting code, HELPMAKE 610
B option, CodeView 310
/B option
CodeView 312
LINK 472

NMAKE 530
b, white space, predefined expression syntax 846, 848, 853
Backing up files, PWB 87, 281, 654–656
Backslash (\)
escape, regular expression syntax 847–848, 854
HELPMAKE syntax 604–605
line continuation character, PWB 105, 107, 126
LINK syntax 463
makefile syntax 535, 538, 543, 551–552
match character 847–848, 854
regular expressions, PWB 88
Screen Exchange command, CodeView 400, 454
Backtab function, PWB 118, 140, 149
Backup files
creating 654–656
setting number, in PWB 281
Backup switch, PWB 244, 251
.BAK file defined 858
Banner *See* /NOLOGO option
.BAS file defined 858
Base names
Curfilenam predefined macro, PWB 211
defined 858
Shortnames switch, PWB 275
Base operator (:<), CodeView precedence 382
__based keyword, explicit allocation 509
.BAT file defined 858
Batch files
backing up .ASM files, PWB 87
building browser database, PWB 94–96
defined 858
/BATCH option, LINK 472
BC command, CodeView 398, 402–403
BC macro (NMAKE) 558
BD command, CodeView 398, 403–404
BE command, CodeView 398, 404–405
Beep switch, PWB 244, 251
Begfile function, PWB 149
Begline function, PWB 150
BFLAGS macro (NMAKE) 559
BH register, CodeView syntax 395, 426
Binary
files defined 858
operators defined 858
BIOS defined 858
Bit rate, remote debugging 370
BL command, CodeView 398, 405
BL register, CodeView 395, 426
Black color value 254
Blue color value 254
.BMP file defined 858
Bold text, HELPMAKE formatting
QuickHelp 604–605
rich text format 610
Boolean switches, PWB 112, 248
Box Mode command, PWB 64, 133
Boxes, command execution, PWB 72
BP command, CodeView 358–359, 376, 398, 405–408, 420–421
BP register, CodeView syntax 395, 426
Braces ({ })
context operator, CodeView 381–384, 397–398
key box, PWB 110
makefile syntax 542, 565
RTF formatting codes 610
Brackets ([ ])
balancing, PWB 180–181
call tree, PWB 92
character class 847, 849, 854
match character 844
Breakpoint Clear command, CodeView 398, 402–403
Breakpoint Disable command, CodeView 398, 403–404
Breakpoint Enable command, CodeView 398, 404–405
Breakpoint List command, CodeView 398, 405
Breakpoint Set command, CodeView 358–359, 398, 405–408, 420–421
Breakpoints
CodeView
clearing 402–403
disabling 403–404
enabling 404–405
listing 405
saving 317
setting 298, 339–341, 358–359, 405–408
defined 858
Bright Back check box, customizing colors, PWB 115
Bright Fore check box, customizing colors, PWB 115
Browcase switch, PWB 286
Brown color value 254
Browse Information Compactor *See* SBRPACK
Browse menu, PWB 68, 134, 187
BROWSE option, and BSCMAKE, COBOL 616
Browse Options command, PWB 67
Browser database
See also BSCMAKE; SBRPACK
.BSC file 615, 618
builds
described 616
full 617, 619, 621
incremental 617, 619
optimizing 617, 624
options 620, 622
creating .SBR files 616
empty .SBR files 617
file size 617–618
full build 617, 619, 621
include files, excluding 620–621, 624

incremental build 617, 619
Browser database (*continued*)
information 621
local symbols, excluding 616, 620
macro expansions, excluding 620
naming the .BSC file 621
optimizing 617, 624
options 620, 622
overview 615
packing .SBR files 616, 623–624
public symbols 616
PWBRMAKE.EXE 615
removing an .SBR file 617
.SBR file 616
specifying .BSC file 617
symbols
excluding 616, 620
unreferenced 621, 623
tools 615
truncated .SBR files 617
unreferenced symbols 621, 623
updating a database 616
verbose output 621
zero-length .SBR files 617
Browser Database Maintenance Utility *See* BSCMAKE
Browser database, PWB
building 52, 92, 94–96
creating 89–90
estimating file size 94
finding symbol definitions 92–93
makefiles 55
menu commands 68
specifying switches 287
Browser information files, PWB
browser database 89–90
building browser database, non-PWB 95
estimating size 94
Browser Output command, PWB 69
Browser Output pseudofile, PWB 93
Browser, source *See* Source browser
BSCMAKE
See also Browser database; SBRPACK
.BSC file 615, 618
building a database 616
builds
full 617, 619, 621
incremental 617, 619
command line 619–620
copyright message 621
creating .SBR files 616
empty .SBR files 617
environment variable, INCLUDE 620
error codes 623
exit codes 623
full build 617, 619, 621
BSCMAKE (*continued*)
help 620–621
INCLUDE environment variable 620
include files, excluding 620–621, 624
incremental build 617, 619
information 621
local symbols, excluding 620
macro expansions, excluding 620
naming .BSC file 617, 621
operating system 618
options 620–622
overview 615, 618
packing .SBR files 616, 623–624
PWBRMAKE.EXE 615
removing an .SBR file 617
response file 622
return codes 623
rules 619
running 619–620, 622
running requirements 618
.SBR file 616
symbols
excluding 620
unreferenced 621, 623
syntax 619, 622
system requirements 618
truncated .SBR files 617
unreferenced symbols 621, 623
updating a database 616
verbose output 621
zero-length .SBR files 617
BSCMAKE command, building browser database
non-PWB 95
PWB 89–90
BSCMAKE.EXE 618
.BSC files
defined 858
PWB
building browser database 89
estimating size 94
use 615, 618
Buffers
CodeView command window 328
decompression, specifying size 304
defined 858
Bugs *See* Debugging
Build command, PWB 66, 134
Build errors in PWB 23
Build Results command, PWB 69
Build Results window, PWB
clearing 152
Nextmsg function 177
PWB 242

retaining results 268
Build switch, PWB 244
Build:message switch, tagged expressions 852
Building
browser database, PWB 89–90, 92, 94–96
canceling, _pwbcancelbuild macro 214
canceling, _wbcancelbuild macro 215
customized PWB projects 52–55
described, PWB 51–52
menu commands, PWB 66
multimodule programs, PWB 40
projects *See* NMAKE
_pwbbuild macro 214
targets, PWB 153–154
Buttons
command execution, PWB 72–74
hyperlinks
index screens 672
navigating with 666–668
BX register, CodeView syntax 395, 426
BY operator, CodeView 381, 390–392
Bytes, displaying, CodeView 330–332

C

C Compiler Options command, PWB 67
C expression evaluator
choosing 380
defined 375
overview 375
using 381–384
C function prototypes 648
C header files 634
C option, CodeView 310
C preprocessor keywords recognized by H2INC (list) 652
C preprocessor pragmas recognized H2INC (list) 652
C++ Compiler Options command, PWB 67
C++ expression evaluator
choosing 380
overview 375
using 381–384, 386–392
c, alphabetic character, syntax 846, 848, 853
C, C++ expression evaluator, debugging assembly language programs
with BY memory operator 389–390
with DW memory operator 389–390
with WO memory operator 389–390
C, C++ expression evaluators
and MASM 389
symbols, search order 382
:c command, HELPMAKE 606
\c, escape, regular expression syntax 843, 846
.C files defined 858
/C option
CodeView 312–313
/C option (*continued*)
HELPMAKE 596
NMAKE 530
/c option, ML, debugging considerations 296
C, C++ programs, debugging restrictions with CodeView 382
Call gates defined 858
Call Tree (Fwd/Rev) command, PWB 68
Call trees, PWB 78, 91–92
Calling functions, CodeView expressions 382
Calls menu, CodeView 346–347
Cancel function, PWB 140, 150
Cancelling
background search, _pwbcancelsearch macro 215–216
builds, _pwbcancelbuild macro 214–215
print operations, _pwbcancelprint macro 215
Cancelsearch function, PWB 141, 151
Caret (^)
character ranges 845
line beginning 845, 847, 849, 854
makefile syntax 535, 551–552, 561
regular expressions, PWB 84, 87
Cascade command, PWB
described 69
predefined macros 135
Cascading window arrangements, _pwbcascade macro 216
Case sensitivity
browser database 286
CodeView
commands 393
expression evaluators 382
generally 346
options 422–424
command, CodeView 342, 346
.DEF file syntax 494
defined 859
global searches, in Microsoft Advisor 674
/IGN option, LIB 585
IMPLIB 653
/NOI option
LIB 585
LINK 480
PWB
options 131–132
search functions 251–252
Case switch, PWB 244, 251–252
Case, matching, CodeView search option 335, 336
CASEMAP:NONE argument, OPTION directive 638
.category command, HELPMAKE 606
CC macro (NMAKE) 558

_cdecl keyword, symbol format, CodeView 385–386
Cdelete function, PWB 140, 151
CFLAGS macro (NMAKE) 559
CGA
  defined 859
  displays, suppressing snow, CodeView option 314
CH register, CodeView syntax 395, 426
Character range, regular expression syntax 847
Character strings defined 859
Characters
  ASCII, defined 856
  classes,
    PWB 84
    regular expression syntax 845, 847, 849, 854
  control, HELPMAKE 596
  deleting, PWB 151, 157, 196
  inserting, PWB 163
  matching, regular expression syntax 847
  predefined expression syntax 846, 848
  searching 83
Check box, PWB 74
Checksum, EXEHDR 518, 522
Child process defined 859
CL
  options, debugging considerations 296
  register CodeView syntax 395, 426
Class Hierarchy command, PWB
  described 68
  function 134
CLASS keyword, SEGMENTS statement 502
Class Tree (Fwd/Rev) command, PWB 68
Class Tree command, PWB 134
Classes
  characters, syntax 845, 847, 849, 854
  CodeView accessibility 386–388
Clearing breakpoints in CodeView 341–342, 402–403
Clearmsg function, PWB 140, 152
Clearsearch function, PWB 140, 152
Click defined 859
Clipboard defined 859
Clipboard Results command, PWB 69
Close All command, PWB
  described 69
  predefined macros 135
Close command
  CodeView 347–348
  PWB 64, 69, 132, 134–135
Close Project command, PWB 66
Closefile function, PWB 140, 153
Closing
  files, PWB 64, 217–218
  help files
    PWB 200
    QuickHelp 677
  menus, PWB 71
  projects, PWB 218
Closing (*continued*)
  windows, PWB 206, 217
CLRFILE.CV4, CodeView 315–316, 334–345
CLRFILE.CVW, CodeView 315–316, 334–345
!CMDSWITCHES directive 534, 572
/CO option, LINK 296–297, 473, 632
.COB files defined 859
COBFLAGS macro (NMAKE) 559
COBOL macro (NMAKE) 558
Code
  inline, debugging 294
  links, HELPMAKE 602
  searching, PWB 77
  segments
    See also Segments
    attributes 501, 503
    discardable 503
    loading 504
    moving 504
    packing 481–482
    permissions 503
    sharing 504
  source, displaying 324, 433–436
  symbols defined 859
  tracing p-code to native code 366–367
CODE statement 501
Codes, inserting, HELPMAKE rich text format 610
CodeView 437
  32-bit register command 346
  access control 386
  active window
    identifying 322
    selecting 322
  ambiguous references, qualifying 386
  and [386 Enh] section of SYSTEM.INI 352
  and Assembly mode 324
  and CVPACK 631
  and environment table 294
  and library code 324
  and MASM 389
  and Microsoft Windows 3.0
    386 enhanced mode 353
    and PIF file 353
    Real mode 353
    Standard mode 353
  and system code 324
  animating 408
  arguments
    entering 326
    format 326
    setting 337–338
  breakpoints

clearing 402–403
disabling 403–404
CodeView (*continued*)
breakpoints (*continued*)
editing 341–342
enabling 404–405
listing 405
on constructors 387
on destructors 387
setting 298, 339–341, 358–359, 405–408
C, C++ program debugging restrictions 382
case sensitivity
commands 346, 393
expression evaluators 382
options 422–424
/CO option, LINK 473
code segment attribute 503
command line 308–316
commands
copying text 327–328
described 436
entering 326
executing 312–313, 409–410
compacting files with CVPACK 631–632
component DLLs, Table of 300
configuring 301
contracting elements 342, 453–454
current radix, overriding 384
CURRENT.STS, PWB 128
CVW
commands 357–360
compared to CV 351–357
general protection faults (GPF) 363
multiple applications 355–357
multiple instances 354
restarting halted debugging sessions 363
running 352–353
techniques 360–363
CVW.EXE and CVW1.386 in current path 352
debugging
assembly language 389–392
p-code 363–367
RND.ASM example 26
displays
black-and-white 312
line-display mode 311
overview 319–321
redrawing 454
screen exchange 313, 315, 345, 422–424, 454
specifying 310–311
suppressing snow 314
dynamic-link libraries 299–300, 314, 337–338
editing 334–335
execution
controlling 361
speed of 429
CodeView (*continued*)
execution (*continued*)
terminating 362
exiting 309
expanding elements 342, 453–454
expression evaluators
and MASM 380
choosing 380–381, 430
defined 375
listing 344
numbers 384
operators 381–384
string literals 385
symbol format 385–386
expressions
See also Expressions
address ranges 379, 396–397
addresses 378, 396
C++ 386–392
line numbers 376, 394
overview 375
registers 377
first time startup, open windows 322
flags, changing values 426–428
functions
listing 411–412
tracing 428
GlobalLock function 439
help
See also Microsoft Advisor
getting 664–673
structure 663
Help menu 665
identifying bugs 297
installing 299–301
interrupt trapping 314
interrupting execution 361
line-display mode with EGA and VGA 311
loading symbolic information 314
locating bugs 298
memory
comparing 413–414
dumping 414–415
entering data 416–418
filling 418–419
format 330–332
management of 308
moving blocks of 419
searching 419–420
viewing 32, 431–433
menus
Calls menu 346–347

Data menu 338–342
Edit menu 334–335
CodeView (*continued*)
menus (*continued*)
File menu 332–334
Help menu 349
Options menu 342–346
Run menu 336–338
Search menu 335–336
Windows menu 347
modules
configuring 337–338
listing 439
mouse, disabling 315
opening windows 322
options 310–315, 317, 370–371, 422–424
packed files 476
p-code, debugging considerations 367
preparing programs 293–297
printing 333
PWB menu commands 66
quitting 334
radix 420–421
radix, syntax 384
registers, changing values 426–428
remote debugging
overview 367
requirements 368–370
starting a session 371–373
syntax 370
restarting 337, 412–413
searching 335–336
shell escape 443–445
slow motion execution 337, 344
source code, displaying 433–436
source files
loading 333
opening 333
state file
overview 316–317
toggling status 315
stepping through a program 28
syntax 308–316
CVW commands 357–360
expressions 376–379, 394–397
regular expressions 845
TOOLS.INI file entries 302–308
TOOLS.INI file entries 302–308
trace speed 429
variables
listing 344
local 328–329
program 324–325
viewing output 349
watch expressions
adding 339, 436
CodeView (*continued*)
watch expressions (*continued*)
deleting 339, 437
listing 441
setting 298–299
watch window
adding expressions 325
and multi-level structures 325
changing variables 325
opening 325
windows
8087 window 330
Command window 326–328, 393
Help window 332
Local window 328–329
Memory windows 330–332
navigation 323
opening 347
overview 320–322
Register window 329–330
Source windows 324
Watch window 324–325
WKA command, and mouse pointer queue blocking 360
CodeView (CVW)
startup position in Microsoft Windows, setting with /X and /Y options 316
/X option, when starting CodeView from Microsoft Windows 316
/Y option, when starting CodeView from Microsoft Windows 316
/CODEVIEW option, LINK 473
CodeView Options command, PWB 67
CODEVIEW.LST, CodeView, Printfile entry in TOOLS.INI 306
Colon (:)
CodeView operator 381
.DEF file syntax 494
Delay command, CodeView 400, 450
HELPMAKE commands 605
LINK syntax 471
makefile syntax 538–540, 542, 560, 564, 570
Color
entry, TOOLS.INI file, CodeView 302
graphics adapter defined 857
switch, PWB 244, 252, 254
Colors
customizing, PWB 114–115
setting, CodeView 345
specifying, PWB 252, 254, 288
values 254
Colors command

CodeView 342, 345
PWB 67
Colors dialog box, CodeView 345
.COM files, 487, 859
Combo box, PWB 73–74
COMDAT record 484, 508
Comma (,)
argument separator, CodeView 326–327
CodeView operator 381
LIB syntax 582–583
LINK syntax 460, 470
with context operator, CodeView 397–398
Command buffer, using CodeView 328
Command button, PWB 74
Command command, CodeView 347–348
.command command, HELPMAKE 606
Command files
See also Response files
defined 859
inline, in makefiles 547
NMAKE 533–534, 859
Command lines
BSCMAKE 619–620
CodeView 308–316
CVPACK 632
EXEHDR 513–514
EXP 656
IMPLIB 653
LIB 582
limit 469
LINK 460–466, 468
NMAKE 529
PWB 131–132
RM 654–655
SBRPACK 624
UNDEL 655–656
Command shell, DOS Shell command, CodeView 333
Command window, CodeView
function 326–328
opening 348
overview 321
Command-line options, H2INC 636
COMMAND.COM, file handles 860
Commands
CodeView
copying text for 327–328
CVW 357–360
Data menu 338–342
Edit menu 334–336
entering 326
executing 312–313, 409–410
File menu 332–334
for Windows applications 351
format 326–327
generally 400–454
Help menu 349
Commands (*continued*)
CodeView (*continued*)
Options menu 342–346
Run menu 336–338
Windows menu 347
defined 859
dot commands, HELPMAKE 597–598, 605–606, 608
Help menu, PWB 70
library *See* LIB
makefile *See* Makefiles; NMAKE
PWB
choosing 70–71
cursor movement 144
Edit menu 64–65
executing 70,–74, 132, 160, 205
File menu 64
Options menu 67
predefined 132–135
Project menu 66
Run menu 66
Run menu, adding 115, 117
Search menu 65
Window menu 69
QuickHelp 678
.comment command, HELPMAKE 606
Comment command, CodeView 400, 446
Comment line, custom builds in PWB 53
Comments
.DEF files 494
DESCRIPTION statement 497
macros (NMAKE) 551
makefiles 536
pragma 465
TOOLS.INI file 126–127, 301, 534
Compact memory model defined 859
Compacting files, CVPACK 631–632
Compile command, PWB, predefined macros 134
Compile File command, PWB 66
Compile function, PWB 140, 153–154
Compiler options, debugging considerations 295–296
Compilers, menu commands, PWB 67
Compiling
debugging considerations 295
defined 859
files, PWB 218–219
Compressing
help database 595–596
keywords, HELPMAKE option 596–597
Concatenating help files 680
Conditional breakpoints defined 859
Conditional operator (?:), CodeView 381
CONFIG.SYS

editing, PWB 58
memory management, CodeView 308
CONFIG.SYS (*continued*)
PWB configuration 127
Configuring CodeView
modules 337–338
TOOLS.INI 301
Consistency check, LIB 584
Constant expressions defined 859
Constants defined 859
Constructors, using C++ expressions 387–388
Contents command
CodeView 349
PWB 70, 135, 665
Context
loperator ({ }), CodeView 381
operator ({ }), CodeView 382–384, 397–398
prefixes, HELPMAKE 613
.context command, HELPMAKE 600–609
contextstring command, HELPMAKE 602–603
Contracting elements in CodeView 342, 453–454
Control characters, specifying, HELPMAKE 596
Conventional memory
browser database 617–618, 621
defined 859
Conversion functions using C++ expressions 387–388
Coprocessors
defined 859
displaying registers, CodeView 330
Copy command
CodeView 335
LIB 589
PWB 64, 133
COPY command, MS-DOS
concatenating help databases 594–595
concatenating help files 680
Copy function, PWB 141, 154–155
Copying
files, PWB 87
text
CodeView commands 327–328
Microsoft Advisor 668
QuickHelp 679
Copyright message *See* /NOLOGO option
/CP option, LINK 473
/CPARM option, LINK 473
/CPARMAXALLOC option, LINK 473
.CPP file defined 859
CPP macro (NMAKE) 559
CPPFLAGS macro (NMAKE) 559
CPU defined 859
Creating
backup files 654–656
browser database, PWB 89–90
call tree, PWB 91–92
import libraries, IMPLIB 652–653
Creating (*continued*)
pseudofiles, in PWB 175–176, 227–228
Cross-reference listing, LIB 590
CS command, CodeView 399
CS register, CodeView syntax 395, 426
CS:IP
defined 860
saving, CodeView 317
Curdate function, PWB 141, 155
Curday function, PWB 141, 155
Curfile predefined macro, PWB 207–210
Curfileext predefined macro, PWB 207, 208, 211
Curfilenam predefined macro, PWB 207, 208, 211
Current date, PWB 155
Current Location command, CodeView 400, 446
CURRENT.STS
CodeView
overview 316–317
saving 334
toggling status of 315
PWB 128
Cursor
defined 860
PWB commands 144–146
shape of, in PWB 254
Cursormode switch, PWB 244, 254
Curtime function, PWB 141, 156
Customize Project Template command, PWB 67
Customize Run Menu command, PWB 66
Cut command, PWB 64, 133
CV *See* CodeView
Cvdllpath entry TOOLS.INI file, CodeView 302–303
CVPACK
and CodeView 631
and LINK 457
command line 632
exit codes 633
help 633
options 633
overview 631–632
syntax 632
CVW
See also CodeView
commands 357–360
compared to CV 351
debugging techniques 360–363
general protection faults (GPF), handling 363
multiple applications, debugging 354–357
running 352, 353
CX register, CodeView syntax 395, 426
.CXX files defined 860
CXX macro (NMAKE) 559

CXXFLAGS macro (NMAKE) 559
Cyan color value 254

D

d., context prefix, HELPMAKE 613–614i
\d: (digit) predefined expression syntax 846, 848, 853
/D option
  CL 405–408
  HELPMAKE 598
  NMAKE 530
  PWB 131
/DA option, PWB 131
Dark Gray color value 254
Dash (-)
  character classes, PWB 84
  character ranges, syntax 845
  HELPMAKE options 595
  LIB syntax 584, 588
  LINK syntax 471
  makefile syntax 544, 572
  NMAKE syntax 529–530, 532
.DAT files defined 860
Data
  dumping, CodeView 414–415
  entering, CodeView 416–418
  importing, module-definition files 507
  moving blocks, CodeView 419
  symbol defined 858
Data menu, CodeView 338–342
Data segments
  See also Segments
  attributes 501, 503
  default *See* DGROUP
  loading 475, 504
  moving 504
  packing 483
  permissions 504
  sharing 503–504
DATA statement 501
Database
  browser *See* Browser database
  help
    context prefixes 613
    creating 595–596
    decoding 597–598
    overview 594–595
Date, current, in PWB 155
Day, current, in PWB 155
.DBG files 473, 487, 860
Dblclick switch, PWB 244, 255
Debug command, PWB 66
Debugger defined 860
Debugging
  See also CodeView
  assembly language 389–392
  /CO option, LINK 473
Debugging (*continued*)
  CodeView options 310–315, 317
  CVW
    commands 357–360
    compared to CV 351–357
    multiple applications 355–357
    multiple instances 352–354
    techniques 360–363
  examining memory 32
  identifying bugs 297
  information
    See also Symbolic debugging information
    defined 860
  libraries 462
  locating bugs 298
  makefiles 530–531
  p-code 363–367
  packaged functions 481
  packed files 476
  programs
    preparing 293–297
    PWB 26
  remote
    bit rate 370
    options 370–371
    overview 367
    requirements 368–370
    starting a session 371–373
    syntax 370
  RND.ASM example 26
  stepping through a program 28
  syntax, TOOLS.INI file entries 302–308
  watch expressions, setting 298–299
Debugging information *See* Symbolic debugging information
Debugging Information Compactor *See* CVPACK
Decoding HELPMAKE options 597–598
Decompressing
  help database 598
  help files, specifying buffer size 304
Decorated names, debugging considerations 296
.DEF files *See* Module-definition files
Default
  data segment *See* DGROUP
  keys, PWB 132–140
  libraries
    defined 860
    LINK 464–465, 479
Define Mark command, PWB 65
DEFINED operator, NMAKE 574

Deflang switch, PWB 244, 255
Defwinstyle switch, PWB 244, 256
Delay command, CodeView 400, 450
Delete command
  LIB 588
  PWB
    described 64
    predefined macros 133
Delete function, PWB 141, 156
Delete Watch command, CodeView 338–339, 399
Delete Watch dialog box, CodeView 339
Delete Watch Expressions command, CodeView 437
DELETED directory, backup utilities 654
Deleting
  breakpoints, CodeView 341–342
  characters, PWB 151, 157, 196
  files
    during debugging session 333
    EXP 656
    PWB 42
    RM 654–655
  lines, PWB 165–166
  marks, PWB 167–168
  text, PWB 156, 216–217
  watch expressions, CodeView 339, 437
Delimiters
  help (> >) 609–610, 612
  regular expressions, PWB 83
Dependency
  command 544
  dependents
    described 542
    filenames 555–556
    macros, predefined 555–556
  described 538
  extending a line 538
  macros, predefined 555–556
  PWB programs 39, 41
  targets
    described 538
    filenames 555–556
    macros, predefined 555–556
    multiple description blocks 539
    pseudotargets 540–541
  time stamps 528
  tree 538–539, 542, 563
  wildcards 536
Dependents
  defined 528
  described 542
  filenames 546, 555–556
  inferred 542, 563, 569–570
  macros, predefined 555–556
  paths 542
  pseudotargets 541
Dereference Global Handle command, CodeView 439–440
Dereference Local Handle command, CodeView 441–442

Dereferencing memory handles, CodeView 360
Description blocks
commands 543
described 537
reusing targets 539–540
TOOLS.INI 534
Description file *See* Makefiles; NMAKE
DESCRIPTION statement 496–497, 518
Destructors using C++ expressions 387–388
DGROUP
attributes, viewing 518
defined 860
/DOSSEG option, LINK 474
segment number 519
DH register, CodeView syntax 395, 426
DI register, CodeView syntax 395, 426
Dialog boxes
CodeView, getting help 669
defined 860
HELPMAKE context prefix 613
PWB
default key assignments 139
displaying 264–265, 268, 273
function 72–74
getting help 669
help 666, 671
Dictionaries, extended 465, 479
Dictionary
extended, /NOE option, LIB 585
extended, suppressing, LIB 585
in a library, LIB 586
Digits, predefined expression syntax 846, 848, 853
DIR command, replacing text, PWB 86–87
Directives
makefile *See* Makefile; NMAKE
preprocessing, NMAKE 573–575
Directories, listing .ASM files, PWB 86
Disable Mouse in CodeView option 315
Disabling
breakpoints, CodeView 341–342, 403–404
Mouse, CodeView option 315
Disassembling defined 860
DISCARDABLE attribute 503
Display
CodeView
arranging 299
black-and-white display 312
line-display mode 311
memory format 330–332
overview 319–321
redrawing 454
screen exchange 313, 315, 345, 422–424, 454
specifying 310–311
Display (*continued*)
CodeView (*continued*)
suppressing snow 314
PWB
height 263
Display
specifying color 252, 254
width 284
screen, PWB 59
Display Expression command, CodeView 400, 452–453
Display modules, listing, CodeView 358
DL register, CodeView syntax 395, 426
.DLL files defined 860
DLLs *See* Dynamic-link libraries
/DO option, LINK 457
.DOC files defined 860
Dollar sign ($)
end of line, syntax 845, 847, 849, 854
makefile syntax 536, 552, 554–556, 560, 562
match line end, syntax 847
reference to tagged expressions, syntax 848, 855
DOS
applications defined 860
Extender defined 860
help, getting 676
keyword, EXETYPE statement 499
managing memory, browser database 617–618
redirection symbol (>), HELPMAKE syntax 597
session defined 860
DOS Protected-Mode Interface server, memory management, CodeView 308
DOS Shell command
CodeView 332–333
PWB 64, 132
DOS shell, creating, PWB 201, 239
DOS-extended defined 860
/DOSS option, LINK 474
/DOSSEG option, LINK 474
Dot commands, HELPMAKE 597–598, 605–606, 608
Dot directives, makefile *See* Makefiles; NMAKE
Double precision defined 860
Down function, PWB 141, 157
DPMI
defined 8561
server *See* DOS Protected-Mode Interface server
Dragging defined 861
/DS option
HELPMAKE 598, 681
LINK 475
PWB 131
DS register
CodeView syntax 395, 426
/DSALLOC option, LINK 475
/DSALLOC option, LINK 475
/DSALLOCATE option, LINK 475

/DT option, PWB 131
/Du option, HELPMAKE 598
Dumping
  defined 861
  math registers, CodeView 448–449
  memory, CodeView 414–415
DW operator, CodeView 381, 390–392
DX register, CodeView syntax 395, 426
/DY option, LINK 475
Dynamic
  address, viewing memory, CodeView 331
  links defined 861
Dynamic Data Exchange, debugging 354–357
/DYNAMIC option, LINK 475
Dynamic-link libraries
  See also Windows, programs for
  creating, LINK 459, 466
  debugging p-code 364
  defined 861
  EXEHDR output 519
  export ordinals 505
  initialization address 519
  initialization routine, debugging 356–357
  LIBRARY statement 496
  listing modules, CodeView 358, 439
  loading symbolic information, CodeView 314
  loading, CodeView 337–338
  name 496
  PRIVATELIB 496
  values, CodeView 299–300

E

E command, CodeView 398, 408, 429
e, predefined expression, syntax 846, 848, 855
e. context prefix, HELPMAKE 613
:e command, HELPMAKE 606
/E option
  HELPMAKE 595–596
  LINK 475
  NMAKE 530, 561, 563
  PWB 132
/EAX register, CodeView syntax 395, 426
EBP register, CodeView syntax 395, 426
EBX register, CodeView syntax 395, 426
ECX register, CodeView syntax 395, 426
__edata variable 474
EDI register, CodeView syntax 395, 426
Edit Breakpoints command, CodeView 338, 341–342
Edit Breakpoints dialog box, CodeView 341–342
Edit menu
  CodeView 334–335
  PWB
Edit menu (*continued*)
  PWB (*continued*)
    described 64
    functions 133
    predefined macros 133
Edit Project command, PWB 66
Editing
  breakpoints, CodeView 341–342
  CONFIG.SYS, PWB 58
  files, Editreadonly switch, PWB 256
  macros, PWB 101
  Noedit function, PWB 178–179
  projects, PWB 41–42, 49
  repeat function, PWB 192
  text, menu commands, PWB 64
Editor, PIF, starting PWB 58
Editor Settings command, PWB 67, 675
Editreadonly switch, PWB 244, 256
EDX register, CodeView syntax 395, 426
EGA defined 861
Ei option, BSCMAKE 620
/El option, BSCMAKE 620
Ellipsis (...)
  call tree, PWB 92
  menu command, PWB 72, 74, 116
!ELSE preprocessing directive, NMAKE 573
!ELSEIF preprocessing directive, NMAKE 573
!ELSEIFDEF preprocessing directive, NMAKE 573
!ELSEIFNDEF preprocessing directive, NMAKE 573
/Em option, BSCMAKE 620
Emacscdel function, PWB 141, 157
Emacsnewl function, PWB 141, 158
EMM386.EXE
  CodeView 308, 310
  defined 861
EMM.386.SYS, CodeView 308, 310
EMS defined 861
Emulators defined 861
Enablealtgr switch, PWB 244, 257
Enabling breakpoints, CodeView 341–342, 404–405
Encoding HELPMAKE options 596–597, 611
.end command, HELPMAKE 606
__end variable 474
Endfile function, PWB 141, 158
!ENDIF preprocessing directive, NMAKE 573
Endline function, PWB 141, 158
English word, predefined expression syntax 846, 848, 853
Enhanced graphics adapter defined 859
Entab switch, PWB 118–119, 244, 257–258
Enterinsmode switch, PWB 244, 258
Enterlogmode switch, PWB 244, 259
Enterselmode switch, PWB 244, 259
Envcursave switch, PWB 128, 245, 259–260
Environment function, PWB 141, 159, 160

Environment strings defined 861
Environment tables
  change in CodeView 294
  defined 861
  saving, in PWB 259–260
Environment variables
  defined 859
  HELPFILES
    defined 861
    help file location 679
    opening help files 677
    restricting global search 675
  INCLUDE 620
  INIT
    defined 862
    remote debugging 371
    use, 534
  LIB 466, 862
  LINK 488–489, 862
  makefiles 530, 561–562
  menu commands, PWB 67
  NMAKE 534
  PATH, installing CodeView 299
  PWB
    function 159–160
    starting 59
    TOOLS.INI file 127–128
  SET command 562–563
  SYSTEM, defined 868
  TEMP defined 869
  TMP 490, 548, 869
Environment Variables command, PWB 67
Envprojsave switch, PWB 128, 245, 260
Equal sign (=)
  .DEF file syntax 494
  makefile syntax 551, 560
  Redirect Input and Output command, CodeView 400, 452
/Er option, BSCMAKE 620
.ERR files defined 861
Error bit
  checking, EXEHDR 519
  clearing, EXEHDR 515
  linking 481
Error codes
  BSCMAKE 623
  CVPACK 633
  defined 861
  LIB 592
  LINK 490
  makefiles 531, 544–545, 571
  NMAKE 580
  SBRPACK 626
!ERROR directive 572
Error messages, getting help 671–672
Error numbers, HELPMAKE context prefix 613
Errors
  building a PWB program 40
  defined 867
  help, getting 671–672
  menu commands, PWB 66
  simulating in CodeView 360, 440–441
/Es option, BSCMAKE 620
ES register, CodeView syntax 395, 426
Escape sequence
  CodeView expressions 385
  defined 861
Escapes, regular expression syntax 845, 847–848, 854
ESI register, CodeView syntax 395, 426
ESP register, CodeView syntax 395, 426
Eval entry, TOOLS.INI file
  CodeView 299–300, 302–303, 380
  remote debugging 368–369
Examine Symbols command, CodeView 399, 443
Exception-mask bits, 8087 command, CodeView 448–449
Exclamation point (!)
  HELPMAKE command 602
  makefile syntax 544, 572
  replacing text, PWB 86
  Shell Escape command, CodeView 399, 443–445
EXE File Header Utility *See* EXEHDR
.EXE files defined 861
Executable files
  alignment, EXEHDR 522
  application type 485, 495, 514, 523
  checksum 518, 522
  .COM file 487
  creating *See* LINK
  defined 861
  error bit
    checking, EXEHDR 519
    clearing, EXEHDR 515
    linking 481
  header
    See also EXEHDR
    format 515
    size 517
  heap *See* Heap
  inserting text 496
  linker version 522
  loading 477
  memory allocation
    EXEHDR 514
    LINK 473
  MS-DOS stub 497
  name of program
    EXEHDR 518–519
    LIBRARY statement 496

Executable files (*continued*)
name of program (*continued*)
NAME statement 495
name, LINK 462
operating system
.DEF file 491
EXEHDR 521
packing
determining, EXEHDR 517
iterated (segment attribute) 523
relocations *See* Relocations
segments *See* Segments
size
/ALIGN option, LINK 472
EXEHDR 517
/EXEPACK option, LINK 476
stack *See* Stack
starting address, EXEHDR 518
Windows, programs for *See* Windows, programs for
.execute command, HELPMAKE 606
Execute command, PWB 66
Execute Commands option, CodeView 312–313
Execute function, PWB 98, 141, 160
EXECUTEONLY, EXECUTE-ONLY attribute 503
EXECUTEREAD attribute 503
Executing
commands, PWB 70–74, 132, 205
functions, PWB 96–98, 160
macros, PWB 96–98
Execution
controlling, CodeView 361
model, specifying, CodeView 305
EXEHDR
32-bit checksum 522
address (in segment table) 520
application type 514, 523
bytes on last page 517
checksum 518
command line 513–514
copyright message 514
data 518–519
description (in output) 518–519
DGROUP
(in output) 519
attribute 518
segment number 519
DLL output 519
entry point 518
entry table 522
error bit
checking 519
clearing 515
.EXE size 517
exports table 520, 523
EXEHDR (*continued*)
extra paragraphs needed 517
extra stack allocation 518
file
in segment table 520
size 517
system, installable 514
flags (in segment table) 520
full information 515
header
format 515
size 517
heap
allocation (in output) 522
setting 514
size 522
help 514–515
imported names table 522
initial CS, IP 518–519
initial SS, SP 518–519
initial stack location 517
initialization 519
iterated (segment attribute) 523
library 519
linker version 522
magic number 517
mem (in segment table) 520
memory
allocation 514
needed 518
requirement 517–518
module 518, 522
movable entry points 522
name
(in exports table) 521
of program 518–519
new .EXE header address 521
no. (in segment table) 520
non-resident names table 522
offset
(in exports table) 521
(in relocations) 524
operating system 521
options 514–515
ord (in exports table) 521
other module flags 523
output
dynamic-link libraries 518, 521
MS-DOS header 516, 521
segmented executable files 518, 521
verbose output 521, 630
Windows, programs for 518, 521
overview 513

EXEHDR (*continued*)
packed .EXE file
iterated (segment attribute) 523
MS-DOS header 517
pages in file 517
paragraphs in header 517
relocations 517–518, 523, 630
relocs (segment attribute) 523
reserved words (in MS-DOS header) 521
resident names table 522
resource table 522
running 513
seg (in exports table) 521
segment
attributes 519, 523
number (in relocations) 524
sector size 522
table 520, 522–523
stack
address 517–518
setting 515
size 518
starting address 518
syntax 513–515
target (in relocations) 524
type
(in relocations) 524
(in segment table) 520
verbose output 515
word checksum 518
/EXEPACK option, LINK 297, 475
EXETYPE statement
described 498
STUB statement interaction 497
EXIST operator, NMAKE 574
Exit codes
BSCMAKE 623
CVPACK 633
defined 861
LIB 592
LINK 490
makefiles 531, 544–545, 571
NMAKE 580
SBRPACK 626
Exit command
CodeView 332–334
PWB 64, 132
Exit function, PWB 141, 160–161
Exiting
CodeView 334
PWB 41, 160–161, 233–234
EXP
command line 656
options 656
EXP (*continued*)
overview 631, 654
syntax 656
Expanded memory
defined 862
emulator defined 862
manager defined 862
Expanding elements in CodeView 342, 453–454
EXPDEF record 460
Explicit
allocation 509, 862
links, HELPMAKE 602
__export keyword 460, 505
Exported functions
See also Imported functions
EXPORTS statement 505
linking 464–465
name
EXEHDR 521
EXPORTS statement 506
OLD statement 505
ordinal number
EXEHDR 521
EXPORTS statement 506
table, EXEHDR 520, 523
EXPORTS statement
See also Exported functions
described 505
name, EXEHDR 521
ordinal number, EXEHDR 521
Expression evaluators, CodeView
and MASM 380
choosing 380, 430
defined 375
listing 344
numbers 384
operators 381–384
specifying 303
string literals 385
symbol format 385–386
Expressions
address ranges 379, 396–397
addresses 378, 396
C++, in CodeView 386–388
constant, defined 857
defined 862
displaying, CodeView 452–453
editing, CodeView 324–325
line number 376, 394
live, creating 331
overview, CodeView 375
predefined *See* Predefined expressions syntax
preprocessing directives, NMAKE 574–575
regular *See* Regular expressions

Expressions (*continued*)
registers 377, 395
setting breakpoints, CodeView 340
tagged *See* Tagged expressions
watch *See* Watch expressions
Expunging files *See* EXP
Extended ASCII defined 862
Extended dictionaries
defined 862
generally 465, 479
suppressing, LIB 585
Extended memory
browser database 617–618, 621
defined 862
Keepmem switch, PWB 265
Extended memory manager
CodeView 308
defined 862
Extending line *See* Line continuation
Extension switches, PWB 246
Extensions
autoloading, PWB 121–122, 249–250
Curfileext predefined macro, PWB 211
default, PWB 255
defined 862
IMPLIB 653
loading, PWB 266
External references 464–465, 862
EXTERNDEF directive generated by H2INC 640

F

F option, CodeView 310
f, filenames, predefined expression syntax 844, 846, 851
/F option
CodeView 313
LINK 476
NMAKE 530
RM 655
F1 key, getting help 665–666
Factor switch, PWB 245, 261
Far address defined 862
Far calls, optimizing 476, 479, 482
/FARCALL option, LINK 476
/FARCALLTRANSLATION option, LINK 476
_fastcall keyword, symbol format, CodeView 385–386
Fast functions, PWB switches 261
Fastfunc switch, PWB 245, 262
FAT defined 862
Fatal errors
defined 862
simulating, CodeView 360, 440–441
/FBr option and BSCMAKE 616
/FBx option and BSCMAKE 616
.FD files defined 862
FFLAGS macro (NMAKE) 559
.FI files defined 862
Fields *See* specific tool
File allocation table defined 860
File Expunge Utility *See* EXP
File handle defined 860
File Header Utility *See* EXEHDR
File history, setting maximum files 279
File menu
CodeView 332–334
PWB
described 64
predefined macros 132
File Removal Utility *See* RM
File Undelete Utility *See* UNDEL
Filehistory, PWB 64
Filename extensions
autoloading, PWB 249–250
Curfileext predefined macro, PWB 211
default, PWB 255
defined 862
IMPLIB 653
loading, PWB 266
filename! command, HELPMAKE 602
Filename-extension tags, TOOLS.INI file, PWB 123
Filename-Parts Syntax, PWB switches 247
Filenames
base names
Curfilenam predefined macro, PWB 211
defined 856
Shortnames switch, PWB 275
defined 862
long, in makefiles 537
makefiles 546, 555–556
predefined expression syntax 846, 848, 853
specifying, HELPMAKE 596, 598
wildcards 536
Files
adding, PWB 38–41, 44
backing up, PWB 87, 281, 654–656
backup *See* Backup files
.BSC 615, 618
closing, PWB 64, 153, 217–218
CodeView requirements 300–301
command *See* Command files; Response files
compacting for CodeView, CVPACK 631–632
compiling, PWB 218–219
creating, PWB 227
.DEF files *See* Module-definition files
deleting
during debugging session 333
PWB 42

RM 654–655
editing, Editreadonly switch, PWB 256
estimating size, PWB 94
executable *See* Executable files
expunging 656
finding, PWB 64
header *See* Include files
help
See also Help files
rich text format 609–611
include
See also Include files
in browser database *See* Browser database
INCLUDE statement 510
inline, in makefiles *See* Inline files
listing, PWB 86
loading, PWB 132
long names
in makefiles 537
NAME statement 496
/NEWFILES option, EXEHDR 514
make *See* Makefiles
MAKEFILE 529–530
map, LINK 463, 478
module-definition *See* Module-definition files
moving
PWB 42
RM 654–655
opening, PWB 64, 131, 179, 269
printing
CodeView 333
PWB 182–183
project file list, PWB 38
relocatable 459
remote debugging 368–370
response *See* Command files; Response files
restoring, UNDEL 655
saving
Autosave switch, PWB 250
PWB 64, 195, 237–238, 279
.SBR 616
searching, PWB 78–82
segmented executable *See* Segmented executable files
source *See* Source files
source, dot commands 605–608
specifying type, HELPMAKE 597
startup, PWB 127
state *See* State file, CodeView
status, PWB 128–129
temporary, LINK 489–490
time stamps *See* Time stamps
TOOLS.INI, makefiles 534–535, 552, 572
Filetab switch, PWB 118, 245, 262
\fin formatting code, HELPMAKE 610
Find command
CodeView 335–336
PWB 64–65, 79–82
Find Dialog box, CodeView 335–336
Finding
files, PWB 64
symbol definitions, PWB 90–93
text, in PWB 83–85
Mreplace function 173–174
Mreplaceall function 174
Qreplace function 189
Replace function 192–193
FIXED attribute 504
Fixup defined 862
Flags
8087 command, CodeView 448–449
changing values, CodeView 426–428
displaying value, CodeView 329–330
register defined 862
FLAGS macro (NMAKE) 559
Flat memory model defined 863
Flipping screen exchange
CodeView 313, 422–424
defined 863
Flow control statements 102–104
.FOR files defined 863
FOR macro (NMAKE) 559
Foreign makefiles in PWB 55–56
Format
commands, CodeView 326–327, 393
HELPMAKE
described 599
QuickHelp 600–606, 608
rich text format 609–611
memory, changing 331
Formatting
attributes, HELPMAKE, QuickHelp format 602, 604–605
codes, HELPMAKE, rich text format 610
text, HELPMAKE topics 604–605
Forward slash (/)
EXEHDR syntax 514
LIB syntax 584
LINK syntax 471
NMAKE syntax 529
/FR option and BSCMAKE 616
/Fr option and BSCMAKE 616
Frames defined 863
.freeze command, HELPMAKE 606
Friction switch, PWB 245, 263
FULL, /MAP option 478
Full-screen application defined 863
Function calls defined 863
Function Hierarchy command, PWB

described 68
Function Hierarchy command, PWB (*continued*)
function 134
Function-level linking 484, 508
Functions
anonymous 509
assigning 508
calling CodeView expressions 382
defined 863
explicit allocation 509
exported *See* Exported functions
external references 464–465
imported *See* Imported functions
listing, CodeView 411–412
ordered 508
overlaid 508–509
packaged
FUNCTIONS statement 508
/NOPACKF option, LINK 481
/PACKF option, LINK 484
PWB
Arg 86, 96–98
Assign 98, 111–112, 114
Backtab 118
call tree 91–92
closing 206
described 140–141, 146–206
executing 96–98
Linsert 98
listing references 92
mark 78
menu commands 132–135
Meta 97–98
modifying 170
Paste 86, 98
Prompt 106
Psearch 97
tabs 118–119
Tell 98
tracing, CodeView 428
FUNCTIONS statement described 508

G

G command, CodeView 398, 409–410
:g command, HELPMAKE 607
G option, CodeView 310
/G option, CodeView 314
Gigabyte defined 863
Global contexts, help files, linking 603–604
Global
heaps, listing memory objects, CodeView 357, 437–438
memory handles, converting to pointers 439–440
symbols
Global (*continued*)
symbols (*continued*)
defined 863
searching for, CodeView 382,–384
Global Search command, PWB 70, 674–675
GlobalLock function in CodeView 439
GlobalLock routine, locking memory handles 360–361
Go command, CodeView 398, 409–410
Goto command, predefined macros, PWB 134
GOTO Definition command, finding symbols, PWB 90–91
Goto Definition command, PWB function 68, 134
Goto Error command described, PWB 66
Goto Mark command, PWB 65
Goto Match command, PWB
described 65
predefined macros 133
Goto Reference command, PWB 68, 134
Grandparent process defined 863
Graphic function, PWB 141, 161
Greater than operator (>), Redirect Input command, CodeView 312–313, 400–451
Green color value 254
Group defined 863

H

H command, CodeView 398, 410
.H files defined 863
h, hexidecimal number, syntax 846, 848, 853
/H option
CVPACK 633
IMPLIB 653
LIB 585
H2INC
and BASIC langtype specification 648
and C return types 648
and _cdecl langtype specification 648
and _fastcall functions 649
and FORTRAN langtype specification 648
and negative numbers in expressions 640
and non-constant integer expressions 638
and Pascal langtype specification 648
and predefined constants 638
and static function prototypes 648
and user-defined constants 638
C data types (list) 640
C prototype conversion, examples 649
command-line options (lists) 636
converting
C bit fields 645

C enumerations 647
C type definitions 647
comments 635
H2INC (*continued*)
converting (*continued*)
constants 638
function prototypes 648–649
nested structures 644
pointers 642
records 645
structures 642–643
unions 642–643
variables 640
converting function prototypes
and /Mn option 649
syntax 648
naming considerations 642
new features 633–634
overview 634
predefined constants (list) 639
recognized C preprocessor keywords (list) 652
recognized C preprocessor pragmas (list) 652
syntax 635
type definitions 647, 649
/U and /u options and /D option 640
Handlers, symbol, specifying 306–308
/HE option, LINK 477
/HEA option, EXEHDR 514
Header, file
See also Include files
examining and changing *See* EXEHDR
format 515
size 517
/HEAP option, EXEHDR 514
Heaps
global, listing memory objects 357, 437–438
local, listing memory objects 358
setting
.DEF file 500
EXEHDR 514
size
EXEHDR 522
limit 500
HEAPSIZE statement 500
Height switch, PWB 245, 263
/HEL option, EXEHDR 514
Help
See also CodeView; Help files; Microsoft Advisor; Quickhelp
displaying in PWB 8, 185, 221, 223
getting
CodeView 664–673
HELPMAKE 598
index table, PWB 223–224
load state, PWB 219–220
next topic, PWB 186, 220
previous topic, PWB 221
Help (*continued*)
searching, PWB 186–187, 224
structure, CodeView 663
switches 288–290
topic selection, PWB switch 289
topic, PWB 222
Help command
CodeView 347–348, 398, 410
PWB 69
Help database
compressing 595–596
context prefixes 613
creating 595–596
decoding 597–598
decompressing 598
overview 594–595
Help delimiters (> >), HELPMAKE 609–610, 612
Help File Maintenance Utility *See* HELPMAKE
Help files
closing
PWB 200
QuickHelp 677
concatenating 680
creating 595–596
decoding 597–598
decompressing 304
formats
described 599
minimally formatted ASCII 612
QuickHelp 603–606, 608
rich text format 609–611
listing 289, 305, 680
locking 597
managing 679–681
opening
Microsoft Advisor 673
PWB 200
QuickHelp 677
overview 594–595
requirements, CodeView 300–301
specifying 597
splitting 681
topics, defining 600–601
Help menu
CodeView 349, 665
PWB 70, 135, 665
Help on Help command
CodeView 349
PWB 70, 135, 665
HELP option, HELPMAKE 599
/HELP option

BSCMAKE 620
CVPACK 633
EXEHDR 514
/HELP option (*continued*)
EXP 656
IMPLIB 653
LIB 585
LINK 477
NMAKE 530
RM 655
SBRPACK 625
UNDEL 656
using 676
Help window
CodeView
function 332
opening 348
overview 322
using 667, 669
PWB
default key assignments 139
using 667, 669
setting size 288
Helpautosize switch, PWB 288
Helpbuffer entry, TOOLS.INI file, CodeView 302, 304
helpfile! contextstring command, HELPMAKE 602
Helpfiles entry, TOOLS.INI file, CodeView 302, 305
HELPFILES environment variable
defined 863
help file location 679
opening help files 677
restricting global search 675
Helpfiles switch, PWB 289, 675
Helplist switch, PWB 289
HELPMAKE 596
compatibility 593
context prefixes 613
decoding 597–598
defining topics 600–601
dot commands 605–606, 608
encoding 595–597, 611
formats
described 599
minimally formatted ASCII 612–613
QuickHelp 600–603, 605–606, 608
QuickHelp format 602–605
rich text format 609
rich text format 610–611
specifying 597
formatting attributes 602–605
formatting text 604–605
getting help 598
global contexts 603–604
local contexts 603–604
options
decoding 597–598
encoding 596–597
HELPMAKE 596 (*continued*)
overview 594–595
syntax 595–599
Helpwindow switch, PWB 289–290
Hexadecimal
defined 863
numbers, predefined expression syntax 844, 846, 851
/HI option, LINK 477
/HIGH option, LINK 477
High memory defined 863
Highlight defined 863
Highlighting search strings in PWB 196, 197
Hike switch, PWB 245, 264
HIMEM.SYS
CodeView 308
defined 863
.HLP files defined 863
Home function, PWB 141, 161
Horizontal Scrollbars command, CodeView 342, 345
HPFS defined 863
Hscroll switch, PWB 245, 264
Huge memory model defined 863
Hyperlinks, Microsoft Advisor
index screens 672
navigating with 666–668

I

I command, CodeView 398, 410–411
:i command, HELPMAKE 607
I option, CodeView 310
/I option
CodeView 314
LIB 585
LINK 477
NMAKE 531
RM 655
i, C/C++ identifier, syntax 846, 848, 853
\i formatting attribute, HELPMAKE 604–605
\i formatting code, HELPMAKE 610
Identifiers
C/C++, syntax 846, 848, 853
case sensitivity 480
defined 863
searching, PWB 85
IEEE format defined 864
!IF preprocessing directive, NMAKE 573
!IFDEF preprocessing directive, NMAKE 573
!IFNDEF preprocessing directive, NMAKE 573
/IGN option, LIB 585

.IGNORE directive 571
/IGNORECASE option, LIB 585
IMPLIB
  case sensitivity 653
IMPLIB (*continued*)
  command lines 653
  .DEF files 493
  EXPORTS statement 506
  IMPORTS statement 506
  linking import libraries 464
  options 653
  overview 631, 652–653
  syntax 653
Implicit links, HELPMAKE 603
Import libraries
  creating, IMPLIB 652–653
  .DEF files 493
  defined 864
  EXPORTS statement 506
  IMPORTS statement 506
  linking 459, 464
Import Library Manager *See* IMPLIB
Imported functions
  See also Exported functions
  EXEHDR 522
  IMPORTS statement 506
  name 507
  OLD statement 505
IMPORTS statement
  See also Imported functions
  described 506
IMPURE attribute 504
.INC files defined 864
INCLUDE directive, PWB project dependencies 39
INCLUDE environment variable 620
Include files
  defined 864
  finding symbols, PWB 92
  in browser database *See* Browser database
  project dependencies, PWB 39, 41
!INCLUDE preprocessing directive, NMAKE 572
INCLUDE statement described 510
/INCR option, LINK 457
Incremental linking 457
Indenting text
  HELPMAKE 610
  PWB 275
Index
  Microsoft Advisor 670
  screens, Microsoft Advisor 672
Index command
  CodeView 349
  PWB 70, 135, 665
Indirection register, debugging, assembly language 391
Inference rules
  command macros 558–559, 568
  commands 543, 564
  defining 564–567
Inference rules (*continued*)
  dependents, inferred 542, 563, 569–570
  described 563
  displaying 531
  ignoring 532
  macros
    in definition 564
    predefined 558–559, 568
  NMAKE-supplied 567–568
  paths 565, 566
  precedence 570
  predefined 567–568
  priority 564
  recursion 558
  rules 564–570
  .SUFFIXES 532, 563, 565, 568,–571
  syntax 564–567
Inferred dependents 542, 563, 569–570
Infinite loops, terminating execution 362
/INFO option, LINK 477
Infodialog switch, PWB 264–265
Information function, PWB 141, 162
/INFORMATION option, LINK 477
Inheritance, makefile
  described 557–558
  macros 557, 563
  /V option, NMAKE 532
Inheritance, using C++ expressions 386
.INI files defined 864
INIT environment variable 534
  defined 864
  PWB 127
  remote debugging 371
  starting PWB 59
Initialization routine, debugging 356–357
Initialize function, PWB 141, 162
Inixre switch, PWB 246
Inline
  code, debugging 294
  files 547,–550
  functions
    debugging at the source level 294
    source level debugging, workaround 294
Input, redirecting, CodeView 450, 452
Insert
  function, PWB 141, 163
  mode, toggling, in PWB 163, 258
Inserting
  characters, PWB 163
  lines, PWB 166–167

RTF formatting codes, HELPMAKE 610
space, PWB 201–202
Insertmode function, PWB 141, 163
Installable file system
defined 864
EXEHDR /NEWFILES option 514
NAME statement 496
Installing CodeView 299–301
int, searching, PWB 83–85
Integers defined 864
Interoverlay calls
defined 864
/DYNAMIC option, LINK 475
Interrupt call defined 864
Interrupt number, overlays 482
Interrupting CodeView 361, 445, 450
Interrupts, trapping, CodeView 314
Intrinsic functions, calling, CodeView expressions 382
I/O privilege mechanism defined 864
Italics, HELPMAKE formatting
QuickHelp format 604–605
rich text format 610
iterated (segment attribute) 523

J

Justifying tagged expressions 852

K

K command, CodeView 411–412
/K option
HELPMAKE 596–597
NMAKE 531
RM 655
KEEP, inline file 548
Keepmem switch, PWB 245, 265
Key assignment, PWB 97, 109–111, 125–126
Key assignments, PWB
cursor movement commands 144
default 135–140
Graphic function 161
menu commands 132–135
Unassigned function 204
Key Assignments command, PWB 67
Key box, assigning key function, PWB 110
Keyboard
choosing commands 70–71
executing PWB commands 70–71
hyperlinks, activating 667
nagivation in CodeView 323
shortcut keys, PWB 71
Keys
shortcut, PWB 71
TOOLS.INI syntax, PWB 125–126
Keywords

See also Reserved words
Keywords (*continued*)
compressing, HELPMAKE option 596–597
help, getting 669, 671
Kilobyte defined 864

L

L command, CodeView 398, 412–413
:l command, HELPMAKE 606
L option, CodeView 310
/L option
CL 355–357
HELPMAKE 597
LINK 478
/L options, CodeView 314
Label defined 864
Label/Function command, CodeView 335–336
Language command, CodeView 342, 344
Language dialog box, CodeView 344
Language Options command, PWB 67
Large memory model defined 864
Lastproject switch, PWB 131, 245, 265–266
Lastselect function, PWB 141, 164
Lasttext function, PWB 141, 164–165
Ldelete function, PWB 141, 165–166
Leaving
CodeView 334
PWB 160–161, 234
Left function, PWB 141, 166
.length command, HELPMAKE 606, 609
Less than operator (<), Redirect Output command, CodeView 312–313, 450
LIB
See also LIB; Libraries
adding a module 587–588
case sensitivity 585
combining libraries 588
command line 582
commands
Add (+) 587–588
Copy (*) 589
Delete (-) 588
Move (-*) 589
Replace (-+) 589
specifying 586
consistency check 584
copying a module 589
copyright message 585
creating a library 584, 587
cross-reference listing 590
defaults
See also specific option or field
command line 582
listing filename 590
LIB (*continued*)
deleting a module 588
error codes 592
exit codes 592
extended dictionary, suppressing 585
extending a line 586
extracting a module 589
fields
commands 586–587
described 583
listfile 590
newlibrary 588, 590
oldlibrary 584, 588
options 584–586
specifying 582–583
file types, input 581
help 585–586
input 584
limits
command line 583
library size 586
line continuation 586
listing 584, 590
memory 585
modules list 590
moving a module 589
naming a library 584, 590
options
/? 586
/HELP 585
/IGN 585
/NOE 585
/NOI 585
/NOL 585
/PAGE 585–586
described 584
rules 584
output 584
output library 590
overview 581
page size 585–586
prompts 582–584
removing a module 588–589
replacing a module 589
response file 583
return codes 592
running 582
saving a library 590
symbols list 590
syntax 582–583
updating a module 589
LIB environment variable 466, 864

.LIB files defined 864
LibEntry routine, debugging 356–357
Libraries
  See also LIB
  combining *See* LIB
  comment pragma 465
  creating *See* LIB
  default 464–465, 479, 858
  defined 864
  dynamic-link *See* Dynamic-link libraries
  extended dictionaries 465, 479
  formats 581
  import *See* Import libraries
  linking 463–466
  linking, LIB 581
  load 462
  load defined 863
  search order 465–466
  size, LIB 586
  standard 581
  standard, defined 868
  static, defined 868
Library files, PWB 38
Library Manager *See* LIB
LIBRARY statement
  described 496
  initialization, EXEHDR 519
  name, EXEHDR 519
LIM EMS defined 864
\lin formatting code, HELPMAKE 610
.line command, HELPMAKE 606
Line continuation
  command, makefile 543
  dependency, makefile 538
  LIB command line 586
  LINK command line 461, 463, 469–470
  macro definition, makefile 551
Line continuation character (\), PWB 105, 107, 126
\line formatting code, HELPMAKE 610
Line Mode command, PWB 64, 133
Line numbers
  CodeView expressions 376, 394
  /LINE option, LINK 478
/LINE option, LINK 478
Line selection mode, setting in PWB 224–225
Line-display mode, setting, CodeView 311
/LINENUMBERS option, LINK 478
Lines
  deleting, PWB 165–166
  inserting, PWB 166–167
  multiple statements, debugging 294
  trailing, display mode, in PWB 280
LINK
  alignment 472
  application type 485
  case sensitivity 480
LINK (*continued*)
  .COM file 487
  command line 460–466, 468
  copyright message 480
  CVPACK 457
  .DBG file 473, 487
  debugging 473
  .DEF file
    See also Module-definition files
    specifying 466
  default libraries 464–465, 479
  defaults
    See also specific option or field
    input 462
    output 459
    prompts 469
  directories 463
  DLLs
    See also Dynamic-link libraries
    creating 459
    .DEF file 466–467
  environment variable 862
  environment variables
    LIB 466
    LINK 488–489
    TMP 490
  error
    bit 481
    codes 490
  errors
    symbol defined more than once 465
    symbol multiply defined 465
    unresolved external 465, 479
  exit codes 490
  extending a line 463
  external references 464–465
  far calls 476, 479, 482
  fields
    deffile 466
    exefile 462
    libraries 463–465, 466
    mapfile 463
    objfiles 461–462
  function-level linking 484, 508
  functions
    ordered 508
    packaged *See* Packaged functions
  halting 461, 468
  help 477, 488
  ILINK 457
  import libraries 464
  information 477–478

input 468
interoverlay calls 475
LINK (*continued*)
interrupting 461, 468
libraries 459, 462–466, 479
library search order 465–466
library search record 465
limits
command line 469
interoverlay calls 475
libraries 463
program size 459, 472
segments 483, 486
stack size 487
line numbers 478
mapfile 463, 478
memory
allocation 473
loading 475, 477
requirement 486
module-definition file
See also Module-definition files
specifying 466
MS-DOS programs
See also MS-DOS programs
creating 459
name
executable file 462
mapfile 463
new features 457–458
NOEXE 481
NUL.DEF 467
null bytes 474, 480
object files 461–462
operating system requirements 457
optimizing
far calls 476, 479, 482
load time 476
relocations 475
options
/? 488
/ALIGN 472
/BATCH 472
/CO 473
/CPARM 473
debugging considerations 295–297
described 471
/DOSSEG 474
/DSALLOC 475
/DYNAMIC 475
environment variable 488
/EXEPACK 475
/FARCALL 476
/HELP 477
/HIGH 477
/INFO 477
LINK (*continued*)
options (*continued*)
/LINE 478
/MAP 478
/NOD 479
/NOE 479
/NOFARCALL 479
/NOGROUP 480
/NOI 480
/NOLOGO 480
/NONULLS 480
/NOPACKC 481
/NOPACKF 481
/OLDOVERLAY 481
/ONERROR 481
/OV 482
/PACKC 482
/PACKD 483
/PACKF 484
/PAUSE 484
/PCODE 485
/PM 485
/Q 485
rules 471, 472
/SEG 486
/STACK 487
/TINY 487
/W 488
ordered functions 508
output files 459–460
overlays
See also Overlays
creating 459
deffile field 466
/DYNAMIC 475
/OLDOVERLAY 481
/OV 482
overview 458–459
p-code 485
packaged functions 481, 484
packing
code 481–482
data 483
executable file 475
paragraphs 473
path 463
program size 472, 476
prompts 469, 472
public symbols 478
PWB menu commands 67
references, resolving 464–465
relocatable files 459

relocations 475–476, 479
requirements 457
LINK (*continued*)
response file 469–470, 472
return codes 490
rules for output 459
running 457, 468
segmented-executable files
See also Segmented-executable files
creating 459, 466
segments
aligning 472
limit 486
loading 475
ordering 474, 480
stack
/STACK option, EXEHDR 515
/STACK option, LINK 487
STACKSIZE statement 500
suppressing messages 472
syntax 460–466, 468–469, 471
system requirements 457
temporary files 489–490
Windows, programs for
See also Windows, programs for
creating 459, 466
LINK environment variable 488–489, 864
LINK Options command, PWB 67
Linker *See* LINK
Linking
debugging considerations 295–297
defined 864, 870
libraries, LIB 581
topics, HELPMAKE 601–603
Linsert function, PWB 98, 141, 166–167
List box, PWB 73
List files defined 865
List References command, PWB
described 68
function 134
List Watch command, CodeView 399
List Watch Expressions command, CodeView 441
Listing
.ASM files, PWB 86
breakpoints, CodeView 405
consistency check, LIB 584
cross references, LIB 590
defined 865
expression evaluators, CodeView 344
functions, CodeView 411–412
help files 680
help files, CodeView 305
mapfile, LINK 463
Microsoft Advisor topics 673
modules, CodeView 439
project files, PWB 38
Listing (*continued*)
references, PWB 92
watch expressions, CodeView 339, 441
Literal characters, searching, PWB 83
Live expressions, creating, CodeView 331
.LNK files defined 865
Load command, CodeView 336–338
Load dialog box, CodeView 337–338
Load libraries 462, 865
Load Other Files option, CodeView 314
Load switch, PWB 245, 266
Loader, Windows 498
Loading
source files, CodeView 333
symbolic information, CodeView 314
LOADONCALL attribute 504
Local
contexts, help files, linking 603–604
heaps, listing memory objects, CodeView 358, 438
memory handles, converting to pointers 441–442
symbols defined 865
variables, listing, CodeView 328–329, 344
Local command, CodeView 347–348
Local Options command, CodeView 342, 344
Local Options dialog box 344
Local window
CodeView
function 328–329
opening 348
overview 322
defined 865
LocalLock routine, locking memory handles 360
Locking help files 597
Log command, PWB
described 65
predefined macros 133
Log Search Complete dialog box, PWB 80
Logged search, PWB 78, 167, 225, 259
Logical segment defined 865
Logsearch function, PWB 141, 167
Long integer defined 865
LONGNAMES keyword, NAME statement 496
Loops, infinite, terminating execution 362
Low memory defined 865
.LRF files defined 865
.LST files defined 865
/lu option, BCSMAKE 621
l-value defined 865

M

M option, CodeView 310
:m command, HELPMAKE 607
/M option
CL, setting breakpoints 405–408
CodeView 315
CVPACK 633
LINK 478
NMAKE 527, 531
PWB 132
m. context prefix, HELPMAKE 613
/MA option, EXEHDR 514
Machine code defined 865
Macros
changing key assignment, PWB 109–111, 125–126
debugging
at the source level 294
programming considerations 294
defined 865
defining, PWB 147–149
executing, PWB 96–98, 160
flow control statements, PWB 102–104
key assignments, PWB 135–140
overview, PWB 99
predefined, PWB 132–135, 207–244
recording, PWB 99–102, 190–191, 234–235
.SBR files, PWB 94
shortcut keys, PWB 71
source level debugging, workarounds 294
TOOLS.INI syntax, PWB 125
undefined, PWB 210–212
user input statements, PWB 104–106
Macros (NMAKE)
assembler 558–559, 568
command 558–559, 568
comments 551
compiler 558–559, 568
defining 551–553
dependent path 542
dependents 555–556
described 550
displaying 531
environment variables 530, 561–562
escaped characters 551
extending a line 551
filename 555–556
ignoring 532
inference rules 564
inheriting 532, 557–558, 563
literal characters 551
Microsoft tools 558–559, 568
nesting 551, 560
newline character 535, 551–552, 561
NMAKE-supplied 554–559, 561–562, 568
null 551, 553–554
precedence rules 563
predefined 554, 556–559, 561–562, 568

Macros (NMAKE) (*continued*)
preprocessing 553
recursion 557–558, 563
replacing strings 560–561
rules 551–554, 560–561, 563
substitution 560–561
syntax 551–554, 560–561
targets 555–556
time stamps 555
TOOLS.INI 552
!UNDEF 553
undefined 551–554, 559, 568
using 554, 560–561
Magenta color value 254
Magic number, in file header 517
.MAK files
See also Makefiles
defined 865
MAKE macro (NMAKE) 557
MAKEDIR macro (NMAKE) 557
MAKEFILE file 529–530
Makefiles
See also NMAKE
association with .PIF files 59
build process 51–52
building browser database, non-PWB projects 94–96
characters, literal 535
command files, inline 547
command modifiers
! (repeat command) 544, 555
- (ignore error) 544
@ (suppress echo) 544
commands
comments 536
dependents 544, 546
described 543
in dependency 544
inference rules 564
inline files 547–550
macros, predefined 558–559, 568
repeating 544, 564
rules 543
wildcards 536
comments 536
customizing 52–55
debugging 530–531
defined 528
dependency *See* Dependency
dependency tree 538–539, 542, 563
dependents
commands 544
described 542
filenames 555–556
inferred 542, 563, 569–570
Makefiles (*continued*)
dependents (*continued*)
macros, predefined 555–556
paths 542
pseudotargets 541
described 535
description blocks *See* Description blocks
directives
dot 570
preprocessing 572
error codes from commands 531, 544–545, 571
exit codes from commands 531, 544–545, 571
filenames
dependents 546
long 537
macros 555–556
wildcards 536
inference rules *See* Inference rules
inline files 547–550
literal characters 535–536
loading, PWB 132
macros *See* Macros (NMAKE)
non-PWB 55–56
opening 131
preprocessing
See also Preprocessing, makefile
error codes 545
exit codes 545
macros 553
return codes 545
suppressing builds 531–532
pseudotargets 540–541
recursion 557–558, 563
response files, inline 547
return codes from commands 531, 544–545, 571
rules 535
sample 578–579
sequence of operations 576–578
SET command 562–563
specifying 530
targets
accumulating 539
build rules 538–542
described 538–539
filenames 555–556
keeping 571
macros, predefined 555–556
multiple description blocks 539–540
pseudotargets 540–541
time stamps *See* Time stamps
TOOLS.INI 534–535, 552, 572
wildcards 536
MAKEFLAGS macro (NMAKE) 557
.MAP files 865

Map files defined 865
/MAP option, LINK 478
Mapfile, LINK 463, 478–479
.mark command, HELPMAKE 607
Mark file, PWB menu commands 65
Mark function, PWB 78, 142, 167–168
Markfile switch, PWB 112, 245, 266–267
Marks
  manipulating, in PWB 167–168
  saving, in PWB 266–267
MASM
  and CodeView 389
  and CodeView expression evaluators 380
  debugging assembly language 389–392
  Options command, PWB 67
  radix 420–421
Match Case command, PWB 68
Match case, search option, CodeView 335–336
Matches, searching, PWB 79–82
Matching
  characters, syntax 844
  regular expressions 284–285, 853–854
Math coprocessors
  defined 865
  displaying registers, CodeView 330
  dumping register contents 448–449
/MAX option, EXEHDR 514
Maximize command
  CodeView 347–348
  described, PWB 69
  predefined macros, PWB 135
Maximize function, PWB 142, 168
Maximizing windows, PWB 168, 226
MAXVAL keyword, HEAPSIZE statement 500
MC command, CodeView 398, 413–414
MD command, CodeView 398, 414–415, 420–421
MDC command, CodeView 415
MDI defined 865
ME command, CodeView 398, 416–418
  and Restart command 412–413
  input radix 420–421
Medium memory model defined 865
Megabyte defined 865
Memory
  allocation
    EXEHDR 514
    LINK 473
  CodeView
    comparing 413–414
    displaying 330–332
    dumping data 414–415
    entering data 416–418
    filling 418–419
    moving data 419
Memory (*continued*)
  CodeView (*continued*)
    searching 419– 420
    viewing 431–433
  expression evaluator requirements 380
  extended
    defined 860
    Keepmem switch, PWB 265
    LINK options 486
  format, changing 331
  high, defined 861
  LIB requirement 585
  LINK requirement 486
  loading 475, 477
  managing, DOS 617–618
  managing, CodeView 308
  NMAKE, running 531
Memory 1 command, CodeView 347–348
Memory 2 command, CodeView 347–348
Memory Compare command, CodeView 398, 413–414
Memory Dump Code command, CodeView 415
Memory Dump command, CodeView 398, 414–415, 420–421
Memory Enter command, CodeView
  and Restart command 412–413
  generally 398, 416–418
  input radix 420–421
Memory Fill command, CodeView
  and Restart command 412–413
  generally 398, 418–419
Memory handles
  converting to pointers 439–442
  dereferencing, CodeView 360
Memory models defined 865
Memory Move command, CodeView 399, 419
Memory objects, listing, CodeView 357–358, 437–438
Memory operators
  CodeView 381
  debugging assembly language 390–392
Memory Search command, CodeView 399
Memory Window command, CodeView 344
Memory Window Options dialog box, CodeView 344
Memory windows, CodeView
  changing values 299
  described 299
  function 330–332
  initializing values 299
  opening 348
  overview 322
  saving addresses 317
  specifying 431–433
Memory-resident program defined 865
Menu bars
  activating, PWB 169

Menu bars (*continued*)
CodeView, overview 320
defined 866
Menu commands
adding, PWB 115, 117
Browse menu, PWB 68, 134, 187
choosing, PWB 70–71
Data menu, CodeView 338–342
Edit menu
CodeView 335–336
PWB 64
executing, PWB 70–71
File menu
CodeView 332–334
PWB 64, 132
Help menu
CodeView 349
PWB 70, 135
help, getting 671
Options menu
CodeView 342–346
PWB 67
predefined macros, PWB 132–135
Project menu, PWB 66, 134
Run menu
CodeView 336–338
PWB 66, 134
Search menu, PWB 65, 133
Window menu
CodeView 347
PWB 69, 135
Menu items
adding, PWB Run menu 282–283
custom, PWB 241–242
help, getting 666
HELPMAKE context prefix 613
Menukey function, PWB 142, 169
Menus
Browse menu, PWB 68, 134, 187
Calls menu, CodeView 346–347
closing, PWB 71
Data menu, CodeView 338–342
Edit menu
CodeView 334–335
PWB 64, 133
File menu
CodeView 332–334
PWB 64, 132
Help menu
CodeView 349, 665
PWB 70, 135, 665
menu bars, CodeView 320
Options menu
CodeView 342–346
Menus (*continued*)
Options menu (*continued*)
PWB 67
Project menu, PWB 66, 134
Run menu
CodeView 336–338
PWB 66, 134, 241–242, 282–283
Search menu
CodeView 335–336
PWB 65, 133
Window menu
PWB 69, 135
CodeView 347
Merge command, PWB 64
Message classes, CodeView options 405–408
!MESSAGE directive 572
Message function, PWB 142, 169
Message numbers, HELPMAKE context prefix 613
Messages, Windows types and class 358–359
Meta function, PWB 97–98, 142, 170
Metacharacters, searching, PWB 83
MF command, CodeView 398, 412–413, 418–419
Mgrep function, PWB 142, 170–171
Mgreplist macro, PWB 210–212
/MI option, EXEHDR 514
Microsoft Advisor 8
copying text 668
error help 671–672
global searches 674–675
help files
concatenating 680
listing 680
managing 679–681
opening 673
splitting 681
Help menus 665
help, getting 664–673
hyperlinks 666–668
index 670
keyword help 669, 671
menu items 665
mouse functions 665
pasting text 668–669
Pwbhelp function 185
structure 663
Microsoft Browse Information Compactor *See* SBRPACK
Microsoft Browser Database Maintenance Utility *See* BSCMAKE
Microsoft Debugging Information Compactor *See* CVPACK
Microsoft EXE File Header Utility *See* EXEHDR
Microsoft File Expunge Utility *See* EXP
Microsoft File Removal Utility *See* RM
Microsoft File Undelete Utility *See* UNDEL

Microsoft Import Library Manager *See* IMPLIB
Microsoft Library Manager *See* LIB
Microsoft Program Maintenance Utility *See* NMAKE
Microsoft Relocatable Object-Module Format (OMF) 458
Microsoft Segmented Executable Linker *See* LINK
Microsoft Symbolic Debugging Information 473
Microsoft Windows
  debugging 351–357
  default key assignments, PWB 140
Microsoft Word, rich text format, HELPMAKE 609
/MIN option, EXEHDR 514
Minimally formatted ASCII
  HELPMAKE 600, 612
  specifying, HELPMAKE 597
/MINIMUM option, CVPACK 633
Minimize command
  CodeView 347–348
  PWB 69, 135
Minimize function, PWB 142, 171
Minimizing windows in PWB 226
Minus sign (–) *See* Dash (–)
Mixed mode defined 864
ML options, debugging considerations 295–296
Mlines function, PWB 142, 171
MM command, CodeView 399, 419
Mnemonics, assembling 400–402
Mode *See* Protected mode; Real mode
Model entry, TOOLS.INI file
  CodeView 299–302, 305
  debugging p-code 364
  remote debugging 368–369
Module Outline command, PWB
  described 68
  function 134
Module-definition files
  application name, type 495
  attributes 503
  case sensitivity 494
  class, segment 502
  code segments 501
  comments 494, 497
  creating 491
  custom loader 498
  data segments 501
  defined 858, 864
  DLLs 505
  export ordinals 505
  exporting 505
  functions
    exported 505
    FUNCTIONS 508
    imported 506
  heap 500
  importing 506
Module-definition files (*continued*)
  include file 510
  inserting text 496
  library name 496
  linking 466
  MS-DOS programs 499
  MS-DOS stub 497
  name of program 495–496
  new features 491–492
  numeric arguments 494
  operating system 498
  overlays 502
  overview 492
  private library 496
  protected mode 499
  PWB 38
  real mode 500
  reserved words 494, 510
  segments
    attributes 501–503
    class 502
    name 502
  stack 500
  statements
    APPLOADER 498
    CODE 501
    DATA 501
    DESCRIPTION 496
    EXETYPE 498
    EXPORTS 505
    FUNCTIONS 508
    HEAPSIZE 500
    IMPORTS 506
    INCLUDE 510
    LIBRARY 496
    NAME 495
    OLD 505
    PROTMODE 499
    REALMODE 500
    SEGMENTS 502
    STACKSIZE 500
    STUB 497
    summaries 493
  syntax rules 494
  version 499
  Windows, programs for
    See also Windows, programs for
    EXETYPE statement 499
    NAME statement 495
Modules
  configuring, CodeView 337–338
  defined 866
  listing, CodeView 358, 439

Monitors
CodeView
black-and-white display 312
redrawing 454
screen exchange 313, 315, 345, 422–424, 454
specifying 310–311
suppressing snow 314
line-display mode, CodeView 311
PWB, specifying color 252, 254
remote debugging 371
Monochrome adapter defined 866
Mouse
choosing commands 70
disabling, CodeView option 315
enabling, PWB 267
executing PWB commands 70–71
help, getting 664–665
hyperlinks, activating 667
pointer defined 866
Mousemode switch, PWB 245, 267
Move command
CodeView 347–348
LIB 589
PWB 69, 135
MOVEABLE attribute
.DEF file 504
EXEHDR 522
Movewindow function, PWB 142, 172
Moving
files
PWB 42
RM 654–655
memory blocks, CodeView 419
windows, PWB 172, 227
Mpage function, PWB 142, 173
Mpara function, PWB 142, 173
MPC, debugging p-code 365
Mreplace function, PWB 142, 173–174, 269
Mreplaceall function, PWB 142, 174
MS command, CodeView 419–420
MS-DOS header format 515
MS-DOS programs
See also Executable files
creating, LINK 459
.DEF files 491
EXETYPE statement 499
header
See also EXEHDR
format 515
information 516
interrupts 482
loading 475, 477
memory allocation 473
MS32EM87.DLL defined 866
MS32KRNL.DLL defined 866
Msearch function, PWB 142, 174–175
Msgdialog switch, PWB 245, 268
Msgflush switch, PWB 245, 268
Multi-level arrays or structures, watching in CodeView 325
Multimodule programs, PWB
assembler options 47, 49–50
building 40
editing 41–42, 49
extending projects 52–55
non-PWB makefiles 55–56
opening projects 36
overview 35
project contents 38
project dependencies 39, 41
using existing projects 42–43
Multiple applications, debugging 354–357
MULTIPLE attribute 504
Multitasking operating system defined 866
Mword function, PWB 142, 175

N

N command, CodeView 399, 420–421
:n command, HELPMAKE 608
n. context prefix,HELPMAKE 613
n, unsigned number, predefined expression syntax 846, 848, 853
/N option
CodeView 310, 314–315
CVPACK 633
NMAKE 531
/n option, BSCMAKE 621
\n, tagged expression reference, syntax 846, 849
Name decorations, debugging considerations 296, 385–386
NAME statement
application type, EXEHDR 514, 523
described 495
name, EXEHDR 518
Named tags, TOOLS.INI file, PWB 124
Naming segments, debugging considerations 294
NAN defined 866
Native command, CodeView 342, 346, 366–367
Native entry, TOOLS.INI file
CodeView 299–300, 302, 305
remote debugging 368–369
Native execution model, specifying, CodeView 305
Navigation
CodeView windows 323
cursor movement commands, PWB 144
Microsoft Advisor 664–673

QuickSearch 678
Navigation (*continued*)
windows, menu commands, PWB 69
/NE option, EXEHDR 514
Near address defined 866
Negated set, regular expression syntax 845
negative numbers in expressions and H2INC 640
Nested structures, expanding, contracting 453
New .EXE header
address 521
format 515
information 521
New command, PWB 64, 69, 132, 135
New line, starting, PWB 158
New Project command, PWB 66
Newfile function, PWB 142, 175–176
NEWFILES keyword, NAME statement 496
/NEWFILES option, EXEHDR 514
Newline character
.DEF file syntax 494
defined 866
makefile syntax 535, 551–552, 561, 564
Newline function, PWB 142, 176
Newwindow switch, PWB 245, 269
.next command, HELPMAKE 607
Next command, PWB 64, 68, 70, 132, 135
Next Error command, PWB 66
multimodule builds 40
predefined macros 134
Next Match command, PWB
described 65
searching 81
Nextmsg function, PWB 142, 177, 269
Nextsearch function, PWB 142, 178, 269
nfodialog switch, PWB 245
NMAKE
See also Makefiles
batch processing 532
building projects, PWB 39, 52–54
builds
conditional 572
errors 572
forcing builds 530
halting 572
ignoring errors 531, 544–545, 571
keeping targets 571
managing projects 528
matching time stamps 530
suppressing 531–532
targets 538
command file 533–534
command files, inline 547
command line 529, 533
command modifiers
! (repeat command) 544–555
NMAKE (*continued*)
command modifiers (*continued*)
- (ignore error) 544
@ (suppress echo) 544
commands
comments 536
dependency line 544
dependents 544, 546
described 543
displaying 531
error codes 531, 544–545, 571
exit codes 531, 544–545, 571
inference rules 564
inline files 547–550
macros, predefined 558–559, 568
modifiers 544
repeating 544, 564
return codes 531, 544–545, 571
rules 543
suppressing display 532, 544, 571
wildcards 536
comments 534, 536, 551
copyright message 530–531
debugging makefiles 530–531
dependency *See* Dependency
dependency tree 538–539, 542, 563
dependents
commands 544
defined 530
described 542
filenames 555–556
inferred 542, 563, 569–570
macros, predefined 555–556
paths 542
pseudotargets 541
description blocks *See* Description blocks
directives
!CMDSWITCHES 534, 572
!dot 570
ERROR 572
.IGNORE 571
!MESSAGE 572
.PRECIOUS 558, 571
preprocessing 572–575
.SILENT 571
.SUFFIXES 531–532, 546, 558, 563, 565, 568–571
!UNDEF 553, 563
environment variables
/E option 530
for NMAKE 534
INIT 534
TMP 548

error codes from commands 531, 544–545, 571
NMAKE (*continued*)
errors
displaying 532
!ERROR 572
suppressing 530
extending a line
command 543
dependency 538
macro 551
fields
macros 529
options 529
targets 529
filenames
dependents 546
long 537
macros 555–556
wildcards 536
forcing builds 540–541
help 530, 532
inference rules
See also Inference rules
displaying 531
ignoring 532
predefined 567–568
information
additional 531
displaying 530–531
!ERROR 572
!MESSAGE 572
suppressing command echo 532, 544, 571
suppressing messages 530
!UNDEF 553, 563
INIT environment variable 534
inline files 547–550
input 530
KEEP 548
limits
command line 533, 547
macro length 551
target length 538, 540
macros
See also Macros (NMAKE)
assembler 558–559, 568
command 558–559, 568
compiler 558–559, 568
dependent path 542
described 550
displaying 531
environment variables 530, 561–562
filename 555–556
ignoring 532
inheriting 532, 557–558, 563
Microsoft tools 558–559, 568
NMAKE (*continued*)
macros (*continued*)
precedence rules 563
predefined 554–562, 568
recursion 557–558, 563
replacing strings 560–561
rules 563
specifying 529
substitution 560–561
syntax 535
MAKEFILE file 529–530
makefiles
See also Makefiles
debugging 530–531
defined 528
described 535
PWB 56
specifying 529–530
standard input 530
messages
displaying 530
!ERROR 572
!MESSAGE 572
suppressing 530
new features 527
NOKEEP 548
operating system
requirements 527
time 528
option macros 559, 568
options
!CMDSWITCHES 572
/? 532
/A 530
/B 530
/C 530
/D 530
/E 530, 561, 563
/F 529–530
/HELP 530
/I 531
/K 531
/N 531
/NOLOGO 531
/P 531
/Q 532
rules 529
/R 532
specifying 529–530
/S 532
/T 532
/V 532, 563
/X 532

NMAKE (*continued*)
output
additional 530–531
!ERROR 572
errors from NMAKE 532
preprocessing 531
suppressing command echo 532, 544, 571
suppressing messages 530
overview 528
preprocessing
See also Preprocessing, makefile
error codes 545
exit codes 545
macros 553
return codes 545
suppressing builds 531–532
pseudotargets 540–541
recursing 532
recursion 557–558, 563
response files, inline 547
return codes from commands 531, 544–545, 571
running 527, 529, 533–535
sample makefile 578–579
sequence of operations 576–578
standard input 530
syntax 529
system requirements 527
targets
accumulating 539
build rules 538–542
building all 530
checking timestamps 532
defined 528
described 538
filenames 555–556
keeping 571
macros, predefined 555–556
multiple description blocks 539–540
pseudotargets 540–541
specifying 529
time stamps
2-second resolution 530
changing 532
checking 532
defined 528
dependencies 528
displaying 530
macros, predefined 555
pseudotargets 541
TMP environment variable 548
TOOLS.INI
!CMDSWITCHES directive 572
described 534–535
ignoring 532
NMAKE (*continued*)
TOOLS.INI (*continued*)
macros 552
touch 532
NMAKE Options command, PWB 67
NMD1PCD.DCC 364
NMWOPCD.DCC 364
/NO option, EXEHDR 514
/NOD option, LINK 479
NODATA keyword, EXPORTS statement 506
/NODEFAULTLIBRARYSEARCH option, LINK 479
/NOE option
LIB 585
LINK 465, 479
Noedit function, PWB 142, 178–179
NOEXE, /ONERROR option 481
/NOEXTDICTIONARY option, LIB 585
/NOEXTDICTIONARY option, LINK 465, 479
/NOF option, LINK 479
/NOFARCALL option, LINK 479
/NOFARCALLTRANSLATION option, LINK 479
/NOG option, LINK 480
/NOGROUP option, LINK 480
/NOGROUPASSOCIATION option, LINK 480
/NOI option
IMPLIB 653
LIB 585
LINK 480
/NOIGNORECASE option
IMPLIB 653
LIB 585
LINK 480
Noise switch, PWB 245, 270
NOKEEP, inline file 548
/NOL option
IMPLIB 654
LIB 585
LINK 480, 489
/NOLOGO option
BSCMAKE 621
CVPACK 633
EXEHDR 514
HELPMAKE 599
IMPLIB 654
LIB 585
LINK 480, 489
NMAKE 531
SBRPACK 625
/NON option, LINK 480
Non-constant integer expressions and H2INC 638
/NONULLS option, LINK 480
/NONULLSDOSSEG option, LINK 480
Non-UNIX
predefined expressions syntax 846

Non-UNIX (*continued*)
regular expressions
matching method 855–856
syntax 847–848, 854–855
syntax, setting, in PWB 281–282
NONAME keyword, EXPORTS statement 506
NONAME keyword, OLD statement 505
NONDISCARDABLE attribute 503
NONE attribute 503
NONE keyword, STUB statement 497
Nonmaskable-interrupt, CodeView 314
Nonmaskable-Interrupt Trapping option, CodeView 315
Nonresident names
NONAME keyword, EXPORTS statement 506
table, EXEHDR 522
NONSHARED attribute 504
/NOP option, LINK 458
/NOPACKC option, LINK 481
/NOPACKCODE option, LINK 481
/NOPACKF option, LINK 481
/NOPACKFUNCTIONS option, LINK 481
NOTWINDOWCOMPAT keyword, NAME statement 495
NOVIO, /PM option 485
Null
bytes 474, 480
character defined 868
pointer defined 868
Number sign (#)
custom builds 53
HELPMAKE syntax 596–597
makefile syntax 536, 551–552, 564
Tab Set command, CodeView 400, 445
TOOLS.INI syntax 534
Numbers, predefined expression syntax 844, 851
Numeric
constants, CodeView expression evaluators 384
switches, PWB 112

O

O command, CodeView 399, 422–424
O3 command, CodeView 422–424
/O option
HELPMAKE 596, 598
LINK 458
/o option, BSCMAKE 621
OA command, CodeView 321, 422–424
OB command, CodeView 422–424
.OBJ files defined 868
Object files
defined 868
in libraries, LIB 581
linking *See* LINK
PWB 38
Object module format defined 868
Object modules defined 868
Object-Module Format 458
OC command, CodeView 422–424
/Od option, ML, debugging considerations 296
OF command, CodeView 422–424
/Of option, CL, debugging p-code 365
/Of- option, CL, debugging p-code 365
Offset defined 868
OFFSET operator, MASM 391
OH command, CodeView 422–424
OL command, CodeView 422–424
/OL option, LINK 481
Old header format 515
OLD statement 505
OLDNAMES.LIB 479
/OLDOVERLAY option, LINK 481
OMF
defined 866
LINK object-file format 458
ON command, CodeView 422–424
/ON option, LINK 481
/ONERROR option, LINK 481
Open command, PWB 64
Open Custom command, PWB 68
Open Module command, CodeView 332–333
Open Module dialog box, CodeView 333
Open Project command, PWB 66
Open Source command, CodeView 332–, 333
Open source File dialog box, CodeView 333
Openfile function, PWB 142, 179, 269
Opening
files, PWB 64, 131, 179, 269
help files
Microsoft Advisor 673
PWB 200
QuickHelp 677
projects
automatically 265–266
PWB 36, 43–44, 183
source files, CodeView 333
Source window, CodeView 324
windows
CodeView 347
PWB 188, 206, 228
Operating system
EXETYPE statement 498
MS-DOS stub 497
Windows *See* Windows, programs for
Operating system prompt, DOS Shell command 333
Operating system tags, TOOLS.INI file, PWB 122
Operations
LIB *See* LIB

regular expressions, PWB 86
Operators
flow control, PWB 102–104
functions, using C++ expressions 388
regular expressions, PWB 83–85, 87
Opt 315
Optimizing
assembler options, PWB 47
debugging considerations 296
far calls, LINK 476, 479, 482
load time 476
relocations, LINK 475
Option button, PWB 73
OPTION directive
CASEMAP 638
EXPR32 *See* EXPR32 argument, OPTION directive
Options
See also specific tool
assembler, changing in PWB 47, 49–50
BSCMAKE 620–622
CL, debugging considerations 296
CodeView
described 310–315, 422–424
remote debugging 370–371
setting 317
compiler 295–296
CVPACK 633
EXEHDR 514–515
EXP 656
HELPMAKE
decoding 597–598
encoding 596–597
IMPLIB 653
in makefiles 559, 568, 572
LIB 584–586
LINK 295–297, 471–488
ML, debugging considerations 295–296
NMAKE 529–532
PWB 131–132
RM 654
SBRPACK 625
UNDEL 656
Options command, CodeView 399, 422–424
Options menu
CodeView 342–346
PWB 67
Ordered functions 508
Ordering segments 294
Ordinal number, export
EXEHDR 521
EXPORTS statement 506
OLD statement 505
OS command, CodeView 422–424
Output
redirecting, CodeView 450–452
Output (*continued*)
screen defined 869
viewing, CodeView 349
/Ov option, CL, debugging p-code 365
/Ov- option, CL 365
/OV option, LINK 482
OV command, CodeView 422–424
OVERLAY keyword, SEGMENTS statement 502
/OVERLAYINTERRUPT option, LINK 482
Overlays
creating 459, 466
defined 867
FUNCTIONS statement 509
interoverlay calls 475
interrupt number 482
LINK 457
overlay number 502
SEGMENTS statement 502
static overlays 481
Overloaded functions using C++ expressions 388
OVL keyword, SEGMENTS statement 502

P

P command, CodeView 399, 425, 428
:p command, HELPMAKE 607
\p formatting attribute, HELPMAKE 603–605
/P option
CodeView 370–371
CVPACK 633
EXEHDR 514
LIB 585–586
NMAKE 531
P register, CodeView syntax 395, 426
p, path, predefined expression syntax 846, 848, 853
P-code
debugging 346, 363–367
linking 485
defined 865
FUNCTIONS statement 508
/NOPACKF option, LINK 481
ordered 508
/PACKF option, LINK 484
/PACKC option, LINK 482
/PACKCODE option, LINK 482
/PACKD option, LINK 483
/PACKDATA option, LINK 483
/PACKF option, LINK
/PACKFUNCTIONS option, 484
registers, displaying 329
Packaged function
defined 867

LINK 484

Packing
code 481–482
data 483
executable file
determining, EXEHDR 517
/EXEPACK option, LINK 475
iterated (segment attribute) 523
files, CVPACK 631–632
.SBR files 616, 623–624
/PADC option, LINK 458
/PADD option, LINK 458
/PAGE option, LIB 585–586
/PAGESIZE option, LIB 585–586
\par formatting code, HELPMAKE 610
\pard formatting code, HELPMAKE 610
Parameters defined 869
Parent process defined 869
Parentheses ( )
balancing, in PWB 180–181
makefile syntax 554, 556, 560
.PAS files defined 869
PASCAL macro (NMAKE) 559
.paste command, HELPMAKE 607
Paste command
CodeView 335
PWB
described 64
predefined macros 133
Paste function, PWB 86, 98, 142, 179–180
Pasting text
Microsoft Advisor 668
QuickHelp 679
PATH environment variable
CodeView, installing 299
starting PWB 59
Paths
Curfile predefined macro, PWB 210
defined 867
predefined expression syntax 846, 848, 853
Patterns *See* Regular expressions
/PAU option, LINK 484
Pause command, CodeView 400, 445
/PAUSE option, LINK 484
Pausing Trace Speed command, CodeView 344
Pbal function, PWB 142, 180–181
/PC option, LINK 485
/PCODE option, LINK 485
.PCH files defined 869
Percent sign (%)
Filename-Parts Syntax, PWB 247
makefile syntax 536
Period (.)
Current Location command, CodeView 400, 446
line number specifier, CodeView 339

Period (.) (*continued*)
LINK syntax 461, 466, 469
makefile syntax 564, 570
match character, syntax 844
wildcard character, syntax 843, 846
/PF option, PWB 131
PFLAGS macro (NMAKE) 559
Physical segments defined 867
PID defined 867
PIF files, association with Makefiles 59
/PL option, PWB 131
\plain formatting code, HELPMAKE 610
Playback macro, PWB 210
Plines function, PWB 142, 181
Plus sign (+)
concatenating help files 680
LIB syntax 587–588
LINK syntax 461, 463, 469–470
makefile syntax 572
searching, PWB 84
/PM option, LINK 485, 495, 514, 523
PM, /PM option 485
/PMTYPE option
EXEHDR 514
LINK 485, 495, 514, 523
Pointers
converting global memory handles 439–440
converting local memory handles 441–442
defined 867
expanding and contracting, CodeView 342, 453–454
H2INC, translated by 642
null 474, 480
.popup command, HELPMAKE 607
Pop-up menu defined 867
Port defined 867
Port Input command, CodeView 398, 410–411
Port Output command, CodeView 399, 424
port: option, CodeView 370
Postfix operator, CodeView precedence 382
Pound sign (#)
custom builds 53
HELPMAKE syntax 596–597
makefile syntax 536, 551–552, 564
Tab Set command, CodeView 400, 445
TOOLS.INI syntax 534
Power, regular expression syntax 845, 853
/PP option, PWB 131
Ppage function, PWB 142, 181–182
Ppara function, PWB 142, 182
PQ register, CodeView 395, 426
Pragmas
alloc_text 509
comment 465
Precedence defined 867
.PRECIOUS directive
described 571
recursion 558
Predefined constants
and H2INC 638
converting 660
Predefined expressions syntax
non-UNIX 848
UNIX 846, 853
_ predefined macro 226
Predefined macros, PWB 132–244
Prefixes
context, HELPMAKE 613
program segments, debugging considerations 295
PRELOAD attribute 504
Preprocessing directives, NMAKE 573–575
Preprocessing, makefile
conditional 572
directives 572
error codes 545
errors, forcing 572
exit codes 545
halting builds 572
indenting 572
macros 553
return codes 545
suppressing builds 531–532
.previous command, HELPMAKE 607
Previous command, PWB
described 68
function 135
Previous Error command, PWB
described 66
predefined macros 134
Previous match command, PWB
described 65
predefined macros 133
searching 81
Print command
CodeView 332–333
PWB 64
Print dialog box, CodeView 333
Print dialog box, CodeView 333
Print function, PWB 142, 182–183
Print Results command, PWB 69
Printcmd switch, PWB 245, 247, 270
Printfile entry, TOOLS.INI file, CodeView 302, 306
Printing
cancelling, _pwbcancelprint macro 215
directly to a printer in CodeView 333
files
CodeView 333
PWB 182–183
specifying program, PWB 270

Printing (*continued*)
to a file, CodeView 333
PRIVATE keyword, EXPORTS statement 506
PRIVATELIB keyword, LIBRARY statement 496
Privileged mode defined 867
Procedure call defined 867
Procedure defined 867
Process defined 867
Process Descriptor Block command block, CodeView 357
Process identification number, defined 867
Program Arguments command, PWB 66
Program build errors 23
Program Item, adding, PWB 58
Program Maintenance Utility *See* NMAKE
Program segment prefixes 295
Program Step command, CodeView
generally 399, 425, 428
controlling execution 361
Program step defined 867
Programs
building 51–52
debugging, preparing for 293–297
PWB
building 40
debugging 26
editing 41–42, 49
multimodule 35
non-PWB makefiles 55–56
project dependencies 39, 41
running 40–41
Project function, PWB 142, 183
Project menu described, PWB 66
Project menu, predefined macros, PWB 134
Project Templates command, PWB 67
Projects
building *See* NMAKE
opening automatically 265–266
PWB
adding files 38–39, 41, 44
closing 218
contents 38
defined 35
deleting files 42
dependencies 39, 41
editing 41–42
extending 52–55
makefiles 51–52
menu commands 66
moving files 42
opening 36, 183
status files 129
using 42–43
Prompt function, PWB 106, 142, 183–184
Prompts
Askexit switch, PWB 248
Askrtn switch, PWB 249
LIB 582–583
LINK 469, 472
Protected mode
defined 855, 868
PROTMODE statement 499
PROTMODE statement 499
PROTO directive generated by H2INC 648
Prototypes converted by H2INC 648
Psearch function, PWB 97, 142, 184–185
Pseudofiles
creating, in PWB 175–176, 227–228
Saveall function, PWB 195
Pseudotargets
dependents 541
described 540–541
time stamps 541
PTR operator, debugging assembly language 391
Public symbols
browser database 616
/CO option, LINK 473
in a library, LIB 590
/MAP option, LINK 478
searching, CodeView 382–384
PURE attribute 504
PWB
Browse menu
described 68
functions 134, 187
build errors 23
build options 18
closing 7
command line 131–132
commands
choosing 70–71
cursor movement 144
executing 70–74, 132, 160, 205
configuration
autoloading 121–122
environment variables 127–128
overview 120–121
customizing colors 114–115
Edit menu
described 64
predefined macros 133
File menu
described 64
predefined macros 132
files
adding 41, 44
deleting 42
estimating size 94

PWB (*continued*)
files (*continued*)
moving 42
functions 96–98, 106, 111–112, 114, 118, 140–141, 146–206
help
copying and pasting 668, 679
getting 8, 664–673
global searches 674–675
keywords 669
managing files 680–681
opening files 673
structure 663
Help menu 70, 135, 665
HELPMAKE, restrictions 593
key assignments 135, 140
macros
changing key assignments 109–111, 125–126
executing 96–98, 160
flow control statements 102–104
overview 99
recording 99–102
user input statements 104–106
makefiles
loading 132
opening 131
multimodule programs
project dependencies 39
using existing projects 42–43
options 18, 131–132
Options menu 67
predefined macros 132–135, 207, 209–244
programs
adding files 38–39, 44
assembler options 47, 49–50
build process 22, 51–52
building 40
editing 41–42, 49
extending projects 52–55
non-PWB makefiles 55–56
opening projects 36
overview 35
project contents 38
project dependencies 41
running 25, 40–41
project file list 38
Project menu 66, 134
prompt
Askexit switch 248
Askrtn switch 249
quitting 41, 160–161, 233–234
regular expressions syntax 281–282
Run menu
adding commands 115, 117

PWB (*continued*)
Run menu (*continued*)
described 66
predefined macros 134
Search menu
described 65
predefined macros 133
searching
find command 79–82
mark function 78
overview 77
regular expressions 82–84
single-module programs, building 18
single-module programs, debugging 26
source browser
browser database 55
building database 92
call tree, showing 91–92
combined database 96
creating database 89–90
estimating file size 94
finding symbols 90, 92–93
non-PWB project database 94–96
Pwbrowse functions 187
starting 57–59
status files 128–129
switches 112–114, 244, 247–275, 277, 279, 280–290
syntax 131–132, 247–248
undefined macros 210–212
View menu 678
Window menutabs 118–119, 123
text, replacing 85–88
TOOLS.INI file
line continuation 126
macros 103–105
switch syntax 126
tags 122–124
tutorial 7
described 69
predefined macros 135
_pwb predefined macro 207
PWB Windows command, PWB 69
_pwbarrange predefined macro 207, 212
_pwbboxmode predefined macro 207, 213–214
_pwbbuild predefined macro 207, 214
Pwbcancelbuild predefined macro 214–215
_pwbcancelbuild predefined macro 207
_pwbcancelprint predefined macro 207, 215
_pwbcancelsearch predefined macro 215–216
_pwbcascade predefined macro 207, 216
_pwbclear predefined macro 207, 216–217
_pwbclose predefined macro 207
_pwbcloseall predefined macro 207, 217
_pwbclosefile predefined macro 207, 217–218

_pwbcloseproject predefined macro 207, 218
_pwbcompile predefined macro 207, 218–219
_pwbfilen predefined macro 207
_pwbgotomatch predefined macro 207, 219
Pwbhelp function 142, 185
_pwbhelp_again predefined macro 207, 220
_pwbhelp_back predefined macro 207, 221
_pwbhelp_contents predefined macro 207, 221
_pwbhelp_context predefined macro 207, 222
_pwbhelp_general predefined macro 207, 223
_pwbhelp_index predefined macro 207, 223–224
Pwbhelpnext function 143, 186
_pwbhelpn1 predefined macro 207
_pwbhelpnl predefined macro 219–220
Pwbhelpsearch function 143, 186–187
_pwbhelp_searchres predefined macro 207, 224
_pwblinemode predefined macro 207, 224–225
_pwblogsearch predefined macro 207, 225
_pwbmaximize predefined macro 207, 226
_pwbminimize predefined macro 209
_pwbmove predefined macro 209, 227
_pwbnewfile predefined macro 209, 227–228
_pwbnewwindow predefined macro 209, 228
_pwbnextfile predefined macro 209, 229
_pwbnextlogmatch predefined macro 209, 229–230
_pwbnextmatch predefined macro 209, 230
_pwbnextmsg predefined macro 209, 231
_pwbpreviouslogmatch predefined macro 209, 231
_pwbpreviousmatch predefined macro 209, 232
_pwbprevmsg predefined macro 209, 232–233
_pwbprevwindow predefined macro 209, 233
_pwbquit predefined macro 209, 233–234
_pwbrebuild predefined macro 209, 234
_pwbrecord predefined macro 209, 234–235
_pwbredo predefined macro 209, 235–236
_pwbrepeat predefined macro 209
_pwbresize predefined macro 209, 236–237
_pwbrestore predefined macro 209, 237
PWBRMAKE.EXE 615
Pwbrosweviewrel function 143
Pwbrowse1stdef function 143, 187
Pwbrowse1stref function 143, 187
Pwbrowsecalltree function 143, 187
Pwbrowseclhier function 143, 187
Pwbrowsecltree function 143, 187
Pwbrowsefuhier function 143, 187
Pwbrowsegotodef function 143, 187
Pwbrowsegotoref function 143, 187
Pwbrowselistref function 143, 187
Pwbrowsenext function 143, 187
Pwbrowseoutline function 143, 187
Pwbrowsepop function 143, 187
Pwbrowseprev function 143, 187
Pwbrowseviewrel function 187
Pwbrowsewhref function 143, 187
_pwbsaveall predefined macro 209, 237–238
_pwbsavefile predefined macro 209, 238
_pwbsetmsg predefined macro 209, 238
_pwbshell predefined macro 209, 239
_pwbstreammode predefined macro 209, 239–240
_pwbtile predefined macro 209, 240
_pwbundo predefined macro 209, 241
_pwbusern predefined macro 209, 241
PWBUTILS, PWB Options menu 67
_pwbviewbuildresults predefined macro 209, 242
_pwbviewsearchresults predefined macro 209, 243
Pwbwindow function 143, 188
_pwbwindow predefined macro 244
_pwbwindown predefined macro 209
Pword function, PWB 143, 188

Q

Q command, CodeView 399, 425
/Q option
  EXP 656
  LINK 485
  NMAKE 532
q, quoted string, predefined expression syntax 846, 848, 853
QH command, MS-DOS 676
.QLB files defined 868
Qreplace function, PWB 143, 189
Question mark (?)
  call tree, PWB 91
  decorated names, C++ 385–386
  Display Expression command, CodeView 400, 452–453
  makefile syntax 555–556
  Quick Watch command, CodeView 400, 453–454
  wildcard character
    regular expression syntax 847, 848, 850, 855
    UNDEL 655
  wildcard operator, HELPMAKE syntax 595
  wildcards 536
Quick Watch command, CodeView 338, 342, 400, 453–454
Quick Watch dialog box
  CodeView
    described 342
    displaying 453–454
    exploring watch expressions 299
QuickHelp
  commands 678
  copying text 679
  CVPACK option 633
  EXP option 656

help files, opening, closing 677
QuickHelp (*continued*)
/HELP option 676
HELPMAKE option 599
IMPLIB option 653
pasting text 679
QH command, MS-DOS 676
specifying format, HELPMAKE 597
topics
displaying 677
navigation 678
selecting 670
UNDEL option 656
QuickHelp format
defining topics 600–601
described, HELPMAKE 599
dot commands 605–606, 608
formatting attributes 602–605
global contexts 603–604
linking topics 601–603
local contexts 603–604
/QUICKLIBRARY option, LINK 485
Quit command, CodeView 399, 425
Quitting
CodeView 334
PWB 41, 160–161, 234
Quotation marks, double (")
character strings 857
CodeView syntax 312–313
.DEF file syntax 494
LINK syntax 461
makefile syntax 537, 553
Pause command, CodeView 400, 445
Quotation marks, single ('), LINK syntax 461
Quote function, PWB 143, 190
Quoted string, predefined expression syntax 846, 848, 853

R

R command, CodeView 399
:r command, HELPMAKE 607
/R option
CodeView 371
EXEHDR 515
EXP 656
NMAKE 532
PWB 132
RM 655
/r option
BSCMAKE 621
LINK 458
Radix
changing in CodeView 420–421
CodeView expression evaluators 384
defined 868
Radix command, CodeView 399, 420–421
RAM defined 868
Random access memory defined 866
rate option, CodeView 370
.RC files defined 868
RC macro (NMAKE) 559
RC.HLP file 679
RCVCOM option, CodeView 370–371
RCVCOM.EXE file, remote debugging 370
RCVWCOM option, CodeView 370–371
Read Only command, PWB 64
READONLY attribute 504
Readonly switch, PWB 112, 245, 247, 270–271
READWRITE attribute 504
Real mode
defined 868
REALMODE statement 499–500
REALMODE statement 500
Realtabs switch, PWB 245, 271
Realtabs switch, white space, PWB 118–119
Rebuild All command, predefined macros, PWB 134
Rebuilding, _pwbrebuild macro 234
Record function, PWB 143, 190–191
Record On command, PWB
described 64
predefined macros 133
Record Results, PWB 69
Recording macros, PWB 99–102, 190–191, 234–235
Records generated by H2INC 645
Recursion, makefile
described 557–558
macros 557, 563
/V option, NMAKE 532
Red color value 254
Redirect Input and Output command, CodeView 400, 452
Redirect Input command, CodeView 312–313, 450
Redirect Output command, CodeView 312–313, 400, 450–451
Redirection defined 868
Redo command, PWB 64, 133
Redraw command, CodeView 400, 454
.ref command, HELPMAKE 607
Refresh function, PWB 143, 191
Register command, CodeView 347–348, 366–367, 399
Register indirection, debugging assembly language 391
Register names, CodeView recognition 377, 395
Register window
CodeView
debugging p-code 366–367
described 299
function 329–330

opening 348
overview 322
defined 868
Registers
changing values, CodeView 426–428
CodeView expressions 377, 395
defined 868
display radix 420–421
displaying value, CodeView 329–330
flags, defined 862
math coprocessors, dumping 448–449
Regular expressions
defined 868
finding, CodeView 335–336
global searches, in Microsoft Advisor 674
matching
non-UNIX 854–855
PWB 284–285
predefined *See* Predefined expressions
replacing text, PWB 85–88
searching for, CodeView 447–448
searching, PWB 77, 82–84
syntax
CodeView 847
non-UNIX 847–848, 854–855
PWB 84
UNIX 845–846, 848–849
tagged *See* Tagged expressions
Relocatable defined 868
Relocatable Object-Module Format, LINK 458
Relocations
descriptions, EXEHDR 523
EXEHDR 630
far calls 476, 479
number, EXEHDR 517, 524
optimizing, LINK 475
table offset, EXEHDR 518
relocs (segment attribute) 523
Remote debugging
bit rate 370
options 370–371
overview 367
requirements 368–370
starting a session 371–373
syntax 370
with Windows-based target files 370
Remove Custom Project Templates command, PWB 67
Removing
breakpoints, CodeView 341–342
status bar, CodeView 321
Repeat
command, PWB 64, 133
function, PWB 143, 192
regular expression syntax 845–849, 854–855
Repeat Last Find command, CodeView 335–336
Repeating function actions in PWB 262

Replace command
LIB 589
PWB 65
Replace function, PWB 143, 192–193
Replacing text
Mreplace function, PWB 173–174
Mreplaceall function, PWB 174
Qreplace function, PWB 189
Replace function, PWB 192–193
.RES files defined 868
Reserved words
.DEF files 494, 510
in MS-DOS header 521
Resetting
CodeView command 412–413
PWB 131
Resident names
RESIDENTNAME keyword, EXPORTS statement 506
table, EXEHDR 522
Resident option, CodeView 371
RESIDENTNAME keyword, EXPORTS statement 506
Resize function, PWB 143
Resizing windows, PWB 236–237
Resource-assembler source file, PWB 38
Response files
See also Command files
BSCMAKE 622
defined 868
inline, in makefiles 547
LIB 583
LINK 469–470
/RESERERROR option, EXEHDR 515
Restart command, CodeView 336–337, 398, 412–413
Restart macro, PWB 210, 212
Restcur function, PWB 143, 194
Restore command
CodeView 347–348
PWB
described 69
predefined macros 135
Restorelayout switch, PWB 245, 272
Restoring
files, UNDEL 655
status bar, CodeView 321
windows, PWB 237
Return codes
BSCMAKE 623
CVPACK 633
defined 861
LIB 592
LINK 490
makefiles 531, 544–545, 571
NMAKE 580
Return codes (*continued*)
described 600, 609
encoding 611
formatting codes 610
specifying 597
Right function, PWB 143, 195
RM
command line 654–655
options 654
overview 631, 654
SBRPACK 626
RFLAGS macro (NMAKE) 559
Rich text format, HELPMAKE
syntax 654–655
Rmargin switch, PWB 245, 272
RND.ASM sample program 14
ROM defined 868
Root defined 868
Routines
defined 869
listing in CodeView 346–347
.RSP files defined 869
RTF *See* Rich text format, HELPMAKE
Rules, inference *See* Inference rules
Run DOS Command command, PWB 66
Run menu
CodeView 336–338
PWB
adding menu items 115, 117, 282–283
custom items 241–242
described 66
predefined macros 134
Run OS/2 Command command, PWB 66
Run-time error defined 869
Running programs, PWB 40–41

S

S option, CodeView 310
/S option
BSCMAKE 621
CodeView 313, 315
EXEHDR 515
NMAKE 532
Sample programs
ONEOF.ASM 11
PWB, SHOW 35–36, 38–43, 47, 49–50, 52–56, 89–93
RND.ASM 14
Save All command, PWB 64, 132
Save As command, PWB 64
Save command, PWB
described 64

predefined macros 132
Save Custom Project Template command, PWB 67
Saveall function, PWB 143, 195
Savecur function, PWB 143, 195–196
Savescreen switch, PWB 245, 273
Saving
CodeView environment 334
files
Autosave switch, PWB 250
PWB 64, 195, 237–238, 279
macros, PWB 102
marks, PWB 266–267
.SBR file 616
.SBR files
building browser database
non-PWB 95
PWB 89–90
defined 869
estimating size 94
SBRPACK
See also Browser database; BSCMAKE
and CL 616
command line 624
copyright message 625
creating .SBR files 616
error codes 626
exit codes 626
fatal errors 626
help 625
include files, excluding 624
operating system 624
options 625
overview 615–616, 623–624
return codes 626
rules 624
running requirements 624
.SBR file 616
syntax 624–625
system requirements 624
Scope
defined 869
specifying, searching for symbols 443
Scope operator (::), CodeView precedence 382
Screen exchange
CodeView options 313, 315, 345, 422–424
defined 869
Screen Exchange command, CodeView 400, 454
Screen Swap command, CodeView 342, 345
Scroll bars
CodeView
options 422–424
toggling options 345
PWB, window styles 207, 256
Scrolling
defined 869
Scrolling (*continued*)
Mlines function, PWB 171
Plines function 181
switches, PWB 264
Vscroll switch, PWB 284
Sdelete function, PWB 143, 196
/SE option, LINK 486
Search command
CodeView 400, 447–448
QuickHelp 677
Search logging, PWB 167, 225
Search Memory command, CodeView 419–420
Search menu
CodeView 335–336
PWB
described 65
predefined macros 133
Search Results command, PWB
described 69–70
predefined macros 135
Search Results dialog box, PWB 81
Search Results window
PWB
clearing 152
described 243
Mgrep function 170–171
Nextsearch function 178
Searchall function, PWB 143, 196–197
Searchdialog switch, PWB 246, 273
Searchflush switch, PWB 246, 274
Searching
backwards, PWB 174–175
cancelling
_pwbcascade macro 215–216
PWB 151
Find command, PWB 79–82
global, Microsoft Advisor 674–675
help system, in PWB 186–187
highlighting search strings, PWB 196–197
in CodeView, overview 335–336
logging searches in PWB 167, 225, 259
mark function, PWB 78
memory, CodeView 419–420
Mgrep function, PWB 170–171
Mgreplist macro, PWB 211–212
overview, PWB 77
regular expressions
CodeView 447–448
PWB 82–84
symbol definitions, PWB 90–93
symbols, CodeView 382–384, 397–398, 443
text, PWB 184–185
Searchwrap switch, PWB 246, 274

Section tags, TOOLS.INI file, PWB 122
/SEG option, LINK 486
Segmented-executable files
creating, LINK 459, 466
defined 869
header *See* EXEHDR
header format 515
MS-DOS stub 497
Segmented-executable Linker *See* LINK
Segments
aligning 472
attributes
code segments 501
data segments 501
EXEHDR 519–520, 523
keywords 503
SEGMENTS statement 502
class 502
code *See* Code segments
data *See* Data segments
defined 869
definitions 502
discardable 503
functions
assigning 508
explicit allocation 509
ordered 508
information 477–478
limit 483, 486, 502
listing 463
loading 504
moving 504
name 502
null bytes 474, 480
ordering 294, 474, 480
overlays 502
packing
code 481–482
data 483
permissions 503–504
sharing 503–504
table, EXEHDR 520, 522–523
/SEGMENTS option, LINK 486
SEGMENTS statement described 502
Selcur function, PWB 143, 197
Select function, PWB 144, 197
Select To Anchor command, PWB
described 64
predefined macros 133
Selected Text command, CodeView 336
Selected text command, CodeView 335
Selecting in PWB
selection mode 198, 224–225, 239–240, 259
text 197
windows 198
Selection modes, PWB
changing 198, 224–225, 239–240, 259
setting 64
Selmode function, PWB 144, 198
Selwindow function, PWB 144, 198
Semaphores defined 869
Semicolon (;)
command separator, CodeView 312–313, 326–327
comments, PWB 126, 127
.DEF file syntax 494
LIB syntax 582–584
LINK syntax 461, 467, 469–470
makefile syntax 542, 544, 564
TOOLS.INI file syntax 301, 534
Separator, custom
predefined macros 133
Set Breakpoint command, CodeView
described 338–341
line numbers 376
Set Breakpoint dialog box, CodeView 339–341
SET command, environment variables 562–563
Set Line-Display Mode option, CodeView 53
Sequence, NMAKE operations 576–578
Sessions
defined 858
remote debugging, starting 371–373
Set Anchor command, PWB
described 64
predefined macros 133
Set Mark File command, PWB 65
Set Project Templates command, PWB 67
Set Record command, PWB 64
Set Runtime Arguments command, CodeView 336–337
Set Screen Swapping option, CodeView 315
Set Screen-Exchange Method option, CodeView 313, 422–424
Set Switch function, changing settings, PWB 114
Setfile function, PWB 144, 199, 269
Sethelp function, PWB 144, 200, 673
SETUP program
CodeView, installing 299–301
help files, installing 679
Setwindow function, PWB 144, 200
SHARED attribute 504
Shell Escape command, CodeView 399, 443–445
Shell function, PWB 144, 201
Shells
defined 869
DOS Shell command 333
Shortcut keys
CodeView 320–321
PWB 71
Shortnames switch, PWB 246, 275

SHOW sample program, PWB 35–36, 38–43, 47, 49–50, 52–56, 89–93
Showing call tree, PWB 91–92
SI register, CodeView syntax 395, 426
Sign-on banner *See* /NOLOGO option
.SILENT directive 571
SINGLE attribute 503
Single precision defined 869
Sinsert function, PWB 144, 201–202
Size command, CodeView 347–348
Size command, PWB
  described 69
  predefined macros 135
Slash (/)
  EXEHDR syntax 514
  HELPMAKE options 595
  LIB syntax 584
  LINK syntax 471
  NMAKE syntax 529
  Search command, CodeView 335–336, 400, 447–448
Slow motion, CodeView execution 337, 344
Small memory model defined 869
SMARTDRV.SYS defined 869
/Sn option, HELPMAKE 597
Snow, surpressing, CodeView option 314
Softcr switch, PWB 246, 275
Source 92
Source 1 command, CodeView 347–348
Source 2 command, CodeView 347–348
Source browser
  See also Browser database; BSCMAKE
  browser database, PWB
    building 92–93, 96
    case sensitivity 286
    combined 96
    creating 89–90
    estimating file size 94
    finding symbols 93
    makefiles 55
    non-PWB projects 94–96
    specifying 287
  makefiles, PWB 55
  menu commands, PWB 68
  Pwbrowse functions 187
  searching, PWB 78
  switches 286–287
Source code, displaying, CodeView 324, 433–436
.source command, HELPMAKE 608
Source files
  decoding, HELPMAKE 597–598
  defined 869
  HELPMAKE formats
    minimally formatted ASCII 612
    QuickHelp 599–608
Source files (*continued*)
  HELPMAKE formats (*continued*)
    rich text format 609–610
  loading, CodeView 333
  opening, CodeView 333
  PWB project file list 38
  specifying type, HELPMAKE 597
Source mode defined 869
Source window
  CodeView
    arranging display 299
    displaying 433–436
    function 324
    getting help 664
    opening 348
    overview 321–322
    setting mode 429–430
  options and view source (VS) command 343
Source1 Window command, CodeView 342
Source2 Window command, CodeView 342
SP register, CodeView syntax 395, 426
Space
  inserting, PWB 201–202
  optimizing, PWB 47
Spaces
  CodeView expression evaluators 382
  .DEF file syntax 494
  LINK syntax 461, 463, 469
  makefile syntax 551–552, 560, 564–565, 570
  trailing, display mode 279–280
Specifiers
  CodeView Options command 422–424
  displaying source code 433–436
  memory format
    dumping memory 414–415
    entering data 416–418
    viewing memory 431–433
  scope, searching for symbols 443
Specify Interrupt Trapping option, CodeView 314
Specifying
  color, PWB display 252, 254
  execution model, CodeView 305
  expression evaluators, CodeView 303
  file type, HELPMAKE 597
  filename, HELPMAKE 596, 598
  interrupt trapping, CodeView 314
  symbol handlers, CodeView 306
Speed of execution
  CodeView 429
  PWB switches 261, 263
Splitting help files 681
SS register, CodeView syntax 395, 426
/ST option, LINK 487

Stack
address 517–518
defined 869
frame defined 869
machine, debugging p-code 363
setting
.DEF file 500
/STACK option, EXEHDR 515
/STACK option, LINK 487
size, EXEHDR 518
trace defined 869
/STACK option
EXEHDR 515
LINK 487
Stack Trace command, CodeView 398, 411–412
STACKSIZE statement 500
Standard
error, defined 870
input, defined 870
library, defined 870
mode, defined 870
output, defined 870
Starting PWB
command line 57–58
Windows Program Manager 58
Startup
code defined 870
files, PWB configuration 127
STARTUP.CMD, PWB configuration 127
State file, CodeView
overview 316–317
toggling status 315
Statefileread entry, TOOLS.INI file, CodeView 302, 306, 315
Statements
.DEF file *See* Module-definition files
flow control, PWB 102–104
multiple, debugging 294
TOOLS.INI syntax, PWB 124–126
user input, PWB 104–106
Static
library defined 870
linking defined 870
Status Bar
overview, CodeView 321
showing, CodeView option 422–424
Status bar defined 870
Status Bar command, CodeView 342, 345
Status files, PWB 128–129
Stream Mode command, PWB 64, 133
Stream selection mode, setting in PWB 239–240
String literals, CodeView expression evaluators 385
String literals, CodeView expression operators 385
Strings
CodeView expression evaluators 385
debugging assembly language 392
defined 870
null terminated
viewing in memory window 392
viewing with memory dump ASCII (MDA) command 392
searching, PWB 77–82
Structure member defined 870
Structures
debugging assembly language 392
defined 870
expanding and contracting, CodeView 342, 453–454
H2INC, generated by 643
nested, expanding and contracting 453
Stub file defined 870
STUB statement
described 497
EXETYPE statement interaction 498–499
Subdirectories, copying file to, PWB 87
Subroutine defined 870
.SUFFIXES directive
dependents 546
described 571
inference rules 563, 568, 570
inferred dependents 569
/P option, NMAKE 531
priority 564–565
/R option 532
recursion 558
Suppress Snow option, CodeView 314
Swapping
defined 870
screen exchange, CodeView 313, 315, 345, 422–424
Switches
PWB 83, 112–114, 118–119, 244, 247–275, 277, 279, 280–282, 284–290
syntax, TOOLS.INI file, PWB 126
Switching, Window function, PWB 206
Symbol handler, specifying, CodeView 306
Symbolhandler entry, TOOLS.INI file
CodeView 299–300, 302, 306–307
remote debugging 368–369
Symbolic Debugging Information 473
compressing 296–297
defined 295, 870
loading 314
memory requirements 296
preserving, with CVPACK 633
searching 336
Symbolic information *See* Symbolic Debugging Information

Symbols
defined 90–93, 857–858, 863
format, CodeView 385–386
.SBR files, PWB 94
searching for, CodeView 397–398, 443
searching order with C, C++ expression evaluators 382
Syntax
See also specific tool, command, or statement
CodeView
command line 308–316
commands 357–360
context operator 382–384, 397–398
expressions 376,–379, 394, 395–397
regular expressions 844
TOOLS.INI file entries 302–308
CVPACK 632
EXP 656
Filename-Parts Syntax, PWB switches 247
HELPMAKE 599
decoding 597–598
encoding 595–597
overview 595
IMPLIB 653
macros, TOOLS.INI file, PWB 125
non-UNIX
predefined expressions 846
regular expressions 845–846, 852–853
PWB
Boolean switches 248
options 131–132
regular expressions 84, 281–282
remote debugging 370
RM 654–655
TOOLS.INI file
switches, PWB 126
tags, PWB 122, 124
UNDEL 655
UNIX
predefined expressions 846, 848, 853
regular expressions 845–846, 848–849
.SYS files defined 870
SYSTEM environment variable defined 870
System include files, finding symbols, PWB 92

T

T command, CodeView 399, 408, 428–429
/T option
HELPMAKE 597–598, 605
LINK 487
NMAKE 532
PWB 132
\tab formatting code, HELPMAKE 610
Tab function, PWB 118, 144, 202
Tab Set command, CodeView 400, 445
Tabalign switch, PWB 118–119, 246, 276
Tabdisp switch, PWB 118, 246, 276
Tabs
.DEF file syntax 494
HELPMAKE syntax 595
hyperlinks, navigating with 666–668
makefile syntax 551–552, 564–565, 570
PWB 118–119, 123, 126, 149, 262, 271, 276–277
regular expressions, PWB 85
setting, CodeView 445
Tabstops switch, PWB 113, 118, 246, 277
Tagged expressions
Build:message switch 852
justifying 852
overview 840–851
regular expression syntax 845–848, 853–855
replacing text, PWB 85–88
Tags, TOOLS.INI file, PWB 122–124, 301
Targets
accumulating 539
build rules 538–542
checking timestamps 532
compiling, PWB 153–154
defined 528
described 538
filenames 555–556
forcing builds 530, 540–541
function, PWB 52–55
keeping 571
length limit 538, 540
macros, predefined 555–556
makefiles, PWB 56
multiple description blocks 539–540
pseudotargets 540–541
Tell dialog box, PWB 202–204
Tell function, PWB 98, 112, 144, 202–204
TEMP environment variable defined 871
Temporary files
defined 871
LINK 489–490
Terminate-and-stay-resident programs
defined 871
DOS Shell command 333
Shell function, PWB 201
Terminating CodeView execution 362
Ternary operator defined 871
Text
Arg function, PWB 96–98
argument, Arg function, PWB 96–98
box, PWB 73
copying

CodeView commands 327–328
Text (*continued*)
copying (*continued*)
Microsoft Advisor 668
QuickHelp 679
defined 871
deleting, PWB 156, 216–217
editing menu commands, PWB 64
files *See* Files
finding, PWB 83–85
formatting, HELPMAKE topics 604–605
indenting, PWB 275
pasting
Microsoft Advisor 668
QuickHelp 679
replacing
PWB 85–88
Qreplace function, PWB 189
Replace function, PWB 192–193
searching, PWB 77–82, 184–185
selecting, PWB 197
strings, searching, PWB 77–82
switches, PWB 112
Text Argument dialog box, PWB
default key assignments 139
Lasttext function 164–165
Prompt function 183, 184
TEXTEQU directive generated by H2INC 638
TH register, CodeView syntax 395, 426
Thread defined 871
Thread ID defined 871
Thread of execution defined 871
Thunk defined 871
Tilde (~) menu command, PWB 116
Tile command
CodeView 347–348
PWB
described 69
predefined macros 135
Tilemode switch, PWB 246, 278
Tiling windows, PWB 240, 278
Time
current, PWB 156
stamps
2-second resolution 530
changing, NMAKE 532
checking 532
checking targets 531
defined 528, 871
dependency 538
displaying, NMAKE 530
macros, NMAKE 555
pseudotargets 541
Timersave switch, PWB 246, 279
Tiny memory model defined 871
/TINY option, LINK 487
TL register, CodeView syntax 395, 426
.TMP files defined 871
TMP environment variable 490, 548
defined 871
starting PWB 59
Tmpsav switch, PWB 246, 279
Toggle State-File Reading option, CodeView 315
Toggling defined 871
TOOLS.INI file
CodeView
configuring 301
entries 302–308
installing 299–301
remote debugging 368–369
setting options 317
comments 534
defined 871
makefiles 534–535, 552, 572
PWB
autoloading extensions 121–122
comments 126–127
extension switches 246
filename-extension tags 123
Initialize function 162
line continuation 126
macros 103–105
named tags 124
operating-system tags 122
sections tags 122
switch syntax 126
TOOLS.PRE file, installing, CodeView 299–300
.topic command, HELPMAKE 608
Topic command
CodeView 349
PWB 70, 135
Topic lists, Microsoft Advisor 673
Topic: command described, PWB 665
Topics, help files, linking 600–603
Touch (time stamps), NMAKE 532
Trace command, CodeView 361, 399, 428
Trace speed command, CodeView 342, 344, 399, 408, 429
Tracepoint defined 871
Tracing
defined 871
functions, CodeView 428
Traildisp switch, PWB 246, 279
Trailing
lines, display mode, in PWB 280
spaces, display mode, in PWB 279–280
Traillines switch, PWB 246, 280
Traillinesdisp switch, PWB 246, 280

Trailspace switch, PWB 246, 280
Translating white space, PWB 119
Transport entry, TOOLS.INI file
  CodeView 299–302, 307–308
  remote debugging 368–369
Transport layer, specifying, CodeView 307–308
Trapping, interrupting, CodeView 314
TSF option, CodeView 310
/TSF option, CodeView 315
TSR
  defined 871
  programs *See* Terminate-and-stay resident programs
Tutorial
  conventions 5
  PWB 7
Twips, defined 611
.TXT files defined 871
Type casting defined 871
TYPEDEF directive generated by H2INC 647, 649

U

U command, CodeView 399, 429–430
\u formatting attribute, HELPMAKE 604–605
\ul formatting code, HELPMAKE 610
Unary operators
  CodeView precedence 382
  defined 871
  preprocessing directives, NMAKE 574
Unassemble command, CodeView 399, 429–430
Unassembling
  defined 858, 871
  p-code 367
Unassigned function, PWB 144, 204
!UNDEF directive
  macros 553
  precedence rules 563
!UNDEF preprocessing directive, NMAKE 573
Undefined macros, PWB 210
UNDEL
  command line 655
  options 656
  overview 631, 654
  syntax 655
Undelcount switch, PWB 246, 281
Underlining, HELPMAKE code 610
Underscore (_), symbol format, CodeView 385–386
Undo command
  CodeView 334
  PWB
    described 64
    predefined macros 133
Undo function, PWB 144, 204
Undocount switch, PWB 112, 246, 281
Unions generated by H2INC 642
UNIX
  predefined expression syntax 846, 853
  regular expression syntax 281–282, 843, 845–846, 848–849
Unixre switch 83, 281–282, 843
UNKNOWN keyword, EXETYPE statement 499
Unresolved external defined 872
Unsigned integer, predefined expression syntax 846, 848, 853
Unsigned numbers, predefined expression syntax 846, 848, 853
Up function, PWB 144, 205
Use 8514 Displays option, CodeView 311
Use Black-and-White Display option, CodeView 312
USE command, CodeView 399, 430
Use Language command, CodeView 399, 430
Use Two Displays option, CodeView 310
Use VGA Displays option, CodeView 311
User input statements 104–106
User switch, PWB 246–247, 282–283
User-defined constants
  and H2INC 660
  converting 660
User-defined type defined 872
Usercmd function, PWB 144, 205
UTILERR.HLP file 679
Utilities
  extension, PWB Options menu 67
  H2INC 634
UTILS.HELP file 679
UTILS.HLP file 681

V

\v formatting attribute, HELPMAKE 602, 604–605
\v formatting code, HELPMAKE 610
/V option
  EXEHDR 515, 521, 630
  HELPMAKE 599
  NMAKE 532, 563
/v option, BSCMAKE 621
Values, entering, CodeView 416–418
Variables
  addresses, debugging assembly code 391
  defined 872
  editing, CodeView 324–325
  environment *See* Environment variables
  local, CodeView 328–329
  .SBR files, PWB 94
  scope, CodeView 422–424
VCPI server

See also Virtual Control Program Interface server
defined 872
/VERBOSE option, EXEHDR 515, 521, 630
Verbose output, HELPMAKE option, HELPMAKE option 599
Vertical Scrollbars command, CodeView 342, 345
VGA
and EGA displays, with CodeView 311
defined 872
display, specifying, CodeView 311
Video graphics adapter defined 872
View Back command, QuickHelp 678
View History command, QuickHelp 678
View Last command, QuickHelp 678
View Memory command, CodeView 399, 412–413, 431–433
View menu, PWB 678
View Next command, QuickHelp 678
View Output command, CodeView 347–349
View References command, searching, PWB 78
View Relationship command, PWB 68, 134
View Source Command, CodeView 376
View Source command, CodeView 399, 433–436
VIO, /PM option 485
Virtual Control Program Interface server, memory management, CodeView 308
Virtual memory
browser database 617–618
defined 872
VM command, CodeView 399, 412–413, 431–433
VM.TMP 490
VS command, CodeView 376, 399, 433–436
Vscroll switch, PWB 246, 284

W

W? command, CodeView 399, 436
w, English word, predefined expression syntax 846, 848, 853
/W option
HELPMAKE 597, 605
LINK 488
/WARNFIXUP option, LINK 488
Watch command, CodeView 347–348
Watch expressions
adding 339, 436
deleting 437
listing 339, 441
saving 317
setting 298–299
Watch window
CodeView
exploring watch expressions 299
function 324–325
opening 348
overview 322

Watch window (*continued*)
expressions
adding, in CodeView 325
changing contents of, in CodeView 325
Watchpoint defined 872
WC command, CodeView 399, 437
WDG command, CodeView 351, 357, 362, 399, 437–438
WDL command, CodeView 351, 358, 399, 438
WDM command, CodeView 358, 399, 439
WGH command, CodeView 351, 360, 399, 439–440
Which Reference? command, PWB
described 68
function 134
White
color value 254
space
converting, PWB 257, 258
predefined expression syntax 846, 848, 853
searching, PWB 83, 85
tab switches, PWB 118–119
translating, PWB 271
Width
HELPMAKE text 597
switch, PWB 246, 284
Wildcards
defined 872
HELPMAKE syntax 595
makefiles 536
regular expression syntax 845, 847–848
Window function, PWB 144, 206
Window menu, PWB
described 69
predefined macros 135
WINDOWAPI keyword, NAME statement 495
WINDOWCOMPAT keyword, NAME statement 495
Windows
CodeView
8087 window 330
Command window 326–328, 393
Help window 332
Local window 328–329
Memory windows 330–332
navigation 323
opening 347
overview 320–322
Register window 329–330
Source windows 324
Watch window 324–325
PWB
activating 244
cascade arrangement 216
closing 217
maximizing 168, 226
minimizing 171, 226
Windows (*continued*)
PWB (*continued*)
moving 172, 227
opening 188, 206, 228
resizing 236–237
restoring 237
selecting 198
styles in 207, 256
tiling 240, 278
Windows Dereference Global Handle command, CodeView 351, 360, 399
Windows Dereference Local Handle command, CodeView 351, 360, 399
Windows Display Global Heap command, CodeView 351, 357, 399, 437–438
Windows Display Local Heap command, CodeView 351, 358, 399, 438
Windows Display Modules command, CodeView 351, 358, 399, 439
Windows File Manager, starting PWB 59
WINDOWS keyword, EXETYPE statement 499
Windows Kill Application command, CodeView 360, 362, 399, 440–441
Windows menu, CodeView 347
Windows Program Manager
help, getting 676
starting, in PWB 58
Windows, programs for
application type 514, 523
character-mode application 485, 495
creating, LINK 459, 466
custom loader 498
EXETYPE statement 499
FORTRAN 498
full-screen application 485, 495
inserting text 496
LIBRARY statement 496
loader 498
module-definition files *See* Module-definition files
MS-DOS stub 497
NAME statement 495
/PM option, LINK 485
private library 496
protected mode 499
real mode 499–500
text window 485, 495
version 499
Winstyle function, PWB 144, 207
WKA command, CodeView 351, 360, 362, 440–441
WL command, CodeView 399, 441
WLH command, CodeView 351, 360, 399, 441–442
WO operator, CodeView 381, 390–392
Word
processor formatting, HELPMAKE text 610

Word (*continued*)
rich text format, HELPMAKE 609
switch, PWB 246, 284–285
wrapping, PWB switches 246, 272, 286
Words
English, regular expression syntax 846, 848, 853
finding in CodeView 335–336
Wordwrap switch, PWB 246, 286
WX
/? 659
/A option 659
/B option 659
environment variable 660
/H option 659
/N option 659
running 659
running synchronously with the MS-DOS prompt 659
using 659
/W option 659
WX/WXServer
about dialog box 658
overview 657
setting request check interval with timer delay 658
WXServer
icon, hiding 658
running 657
using 657

X

(x), grouping 847, 854
x, repeat, regular expression syntax 845, 847–849
/x option, LINK 486
/X option, NMAKE 532
X command, CodeView 399, 443
:x command, HELPMAKE 606
\{x\}, grouping, regular expression syntax 847
\(x), tagged expression 843
\(x\), tagged expression 847
x#, repeat, syntax 845, 852
x*, repeat, syntax 843, 845, 847, 852
x+, repeat, syntax 843, 845, 852
x@, repeat, syntax 845
x^n, 845, 853
XMS
See also Extended memory manager
defined 872
Keepmem switch, PWB 265
server defined 872
(x!y!z!) alternation 854
xyz thing, alternation, syntax 843, 845, 847

Y

:y command, HELPMAKE 606
Yellow color value 254

Z

:z command, HELPMAKE 606
z, unsigned integer, predefined expression syntax 844, 846, 851
/Zd option
CL 296, 385–386
ML 296
/Zd option, CL
and /CO option, LINK 473
and /LINE option, LINK 478
/Zi option
CL 296, 385–386
ML 296
/Zi option, CL
and /CO option, LINK 473
and /LINE option, LINK 478
/Zn option and BSCMAKE 616
/Zs option and BSCMAKE 616