

THE WANG PROFESSIONAL COMPUTER

Program
Development
Guide

Program Development Guide

WANG

The Wang Professional Computer Program Development Guide

1st Edition — April, 1983
Copyright © Wang Laboratories, Inc., 1983
700-8018

WANG

WANG LABORATORIES, INC., ONE INDUSTRIAL AVENUE, LOWELL, MA 01851 • TEL (617) 459-5000, TWX 710-343-6769, Telex 94-7421

Disclaimer of Warranties and Limitation of Liabilities

The staff of Wang Laboratories, Inc., has taken due care in preparing this manual; however, nothing contained herein modifies or alters in any way the standard terms and conditions of the Wang purchase, lease, or license agreement by which this software package was acquired, nor increases in any way Wang's liability to the customer. In no event shall Wang Laboratories, Inc., or its subsidiaries be liable for incidental or consequential damages in connection with or arising from the use of the software package, the accompanying manual, or any related materials.

NOTICE:

All Wang Program Products are licensed to customers in accordance with the terms and conditions of the Wang Laboratories, Inc. Standard Program Products License; no ownership of Wang Software is transferred and any use beyond the terms of the aforesaid License, without the written authorization of Wang Laboratories, Inc., is prohibited.

MS-DOS and XENIX are trademarks of the Microsoft Corporation.

**WANG**

PREFACE

The Wang Professional Computer Program Development Guide is designed to assist intermediate level programmers who will be using the Assembler or the compiled languages on the Wang PC. This guide is intended to be used with The Wang Professional Computer Introductory Guide (700-8020) and with the manuals for the various programming languages. Refer to The Wang Professional Computer Documentation Guide (700-7589) for the names and order numbers of the language manuals.

The guide provides you with two types of information:

- Descriptions of the Wang PC's program development utilities.
- Technical information and specifications of use to the programmer.

Each chapter in this guide discusses one of the program development utilities. The topic of each chapter is summarized below.

Chapter 1 The Editor: Describes the Editor. This utility can create and modify source files for programs in the compiled languages and the Assembler. It can also use create and modify text files such as memos, reports, and lists.

Chapter 2 The Linker: Describes the Linker. This utility combines several separately-compiled object modules into one relocatable run file. The Linker resolves references to symbols external to a particular module and it can search multiple library files for definitions of any external references left unresolved.

Chapter 3 The Library Manager: Describes the Library Manager. This utility creates, modifies, and deletes library files containing program modules that the Linker utility can link with other modules. The Library Manager provides you with a means to create either general libraries for a variety of programs or special libraries for specific programs.

Chapter 4 The Debugger: Describes the Debugger. This utility provides a controlled testing environment for binary files and executable object files. While using the Debugger, you can alter the contents of a file or the contents of a CPU register and immediately reexecute a program to check the validity of the changes.

The appendices present technical information about system architecture, the DOS, file handling and disk I/O, DOS interrupts and function calls, BIOS functions, device drivers, the logical keyboard, screen manipulation, printer control, IBM PC to Wang PC code conversion, and the Wang PC system software diskette.

CONTENTS

CHAPTER 1 THE EDITOR

1.1	Introduction	1-1
1.2	Overview of the Editor	1-1
	The Window	1-1
	Line Numbers	1-2
	Buffer Size	1-2
	Help Screens	1-2
1.3	How to Invoke the Editor	1-2
1.4	Configuration Parameters	1-4
1.5	Editor Commands	1-5
	BACKSPACE	1-5
	CENTER MODE	1-5
	COPY LINES	1-6
	CURSOR EAST and CURSOR WEST	1-6
	CURSOR NORTH and CURSOR SOUTH	1-6
	DELETE CHARACTER	1-6
	DELETE LINE	1-6
	DELETE MANY LINES	1-7
	END SESSION AND SAVE CHANGES	1-7
	ERASE LINE	1-7
	ERASE TO END	1-7
	GO TO FIRST LINE	1-8
	GO TO LAST LINE	1-8
	GO TO LINE	1-8
	INSERT LINE	1-8
	INSERT MODE	1-8
	LOAD PARTIAL FILE	1-9
	LOAD TEXT	1-9
	MOVE LINES	1-9
	NEXT SCREEN	1-10
	PREVIOUS SCREEN	1-10
	PRINT LINES	1-10
	REPLACE	1-10
	SAVE PARTIAL FILE	1-11
	SEARCH	1-11
	SET TABS	1-11
	SET WIDTH	1-12
	SHIFT + CURSOR EAST and SHIFT + CURSOR WEST	1-12
	SHIFT + NEXT and SHIFT + PREV	1-12
	TAB	1-12

CONTENTS (continued)

CHAPTER 2 THE LINKER

2.1	Introduction	2-1
	Overview of Linker Operation	2-1
	Linker System Requirements	2-2
2.2	Segments, Groups, and Classes	2-3
	Definitions	2-3
	How the Linker Combines and Arranges Segments	2-4
2.3	Files the Linker Uses	2-6
	Input Files	2-6
	Output Files	2-7
	VM.TMP File	2-7
2.4	Running the Linker	2-7
	Method 1	2-8
	Method 2	2-8
	Method 3	2-8
	Method 4	2-9
2.5	Command Prompts	2-10
	Object Modules [.OBJ]: Prompt	2-10
	Run File [First-Object-filename.EXE]: Prompt	2-10
	List File [Run-filename.MAP]: Prompt	2-11
	Libraries []: Prompt	2-11
2.6	Command Characters	2-12
2.7	Switches	2-13
	/DSALLOCATE	2-13
	/HIGH	2-14
	/LINENUMBERS	2-14
	/MAP	2-14
	/PAUSE	2-14
	/STACK:<number>	2-15

CHAPTER 3 THE LIBRARY MANAGER

3.1	Introduction	3-1
	Features and Benefits	3-1
	Overview of Library Manager Operation	3-2
	Library Manager System Requirements	3-4
3.2	Running the Library Manager	3-5
	Method 1	3-5
	Method 2	3-5
	Method 3	3-6
	Method 4	3-7

CONTENTS (continued)

3.3	Command Prompts	3-7
	Library file: Prompt	3-8
	Operation: Prompt	3-8
	List file: Prompt	3-9
3.4	Command Characters	3-9
	The Plus Sign	3-10
	The Minus Sign	3-10
	The Asterisk	3-10
	Semicolon	3-11
	Ampersand	3-11
	SHIFT + CANCEL	3-12

CHAPTER 4 THE DEBUGGER

4.1	Introduction	4-1
4.2	How to Invoke the Debugger	4-1
4.3	Initialization	4-2
4.4	Using the Debugger	4-3
4.5	Debugger Command Parameters	4-4
4.6	Debugger Commands	4-7
	A Command (Assemble)	4-8
	C Command (Compare)	4-11
	D Command (Dump)	4-12
	E Command (Enter)	4-14
	F Command (Fill)	4-16
	G Command (Go)	4-17
	H Command (Hex Arithmetic)	4-19
	I Command (Input)	4-20
	L Command (Load)	4-21
	M Command (Move)	4-23
	N Command (Name)	4-25
	O Command (Output)	4-28
	Q Command (Quit)	4-29
	R Command (Register)	4-30
	S Command (Search)	4-33
	T Command (Trace)	4-34
	U Command (Unassemble)	4-36
	W Command (Write)	4-38
	X Command (Change I/O Device)	4-40

CONTENTS (continued)

APPENDICES

Appendix A	PROCESSOR, OPERATING SYSTEM, AND FILE MANAGEGEMENT INFORMATION	
A.1	Introduction	A-1
A.2	Architecture Overview	A-1
	Registers and Flags	A-1
	Memory	A-5
A.3	DOS Technical Information	A-7
	The Contents of a DOS Diskette	A-7
	DOS Initialization	A-8
	The DOS Command Processor	A-10
	DOS Memory Usage	A-11
	Program Initialization	A-13
	Disk Error Handling	A-17
A.4	File and Disk Management	A-17
	File Control Block Definition	A-18
	The Disk Transfer Address	A-20
	Diskette Layouts	A-21
	Disk Space Allocation	A-25
	The Disk Directory	A-25
	The File Allocation Table	A-27
Appendix B	MESSAGES	
B.1	Introduction	B-1
B.2	Linker Messages	B-1
B.3	Library Manager Messages	B-4
B.4	Debugger Messages	B-6
Appendix C	DOS INTERRUPTS AND FUNCTION CALLS	
C.1	Interrupts	C-1
C.2	Function Calls	C-5
	Pre-2.0 Function Calls	C-8
	2.0 Function Calls	C-14
C.3	CP/M Calling Conventions	C-24

CONTENTS (continued)

Appendix D BIOS INFORMATION

D.1	Introduction	D-1
D.2	The Software/BIOS Interface	D-1
D.3	BIOS Trappable Events	D-3
D.4	System Configuration Table	D-4
	Screen Information Block	D-5
	Disk Drive Control Blocks	D-6

Appendix E DEVICE DRIVERS

E.1	Introduction	E-1
E.2	Device Interface	E-1
E.3	Format of a Device Driver	E-2
E.4	Installation of Device Drivers	E-4
E.5	Driver Functions	E-7
E.6	The Static Request Header	E-9
E.7	Data Block Formats	E-11
	Read or Write - ES:BX	E-11
	Nondestructive Read No Wait - ES:BX	E-11
	Media Check - ES:BX	E-12
	Build BPB - ES:BX	E-12
	Status Calls - ES:BX	E-14
	Flush Calls - ES:BX	E-14
	Init - ES:BX	E-14
E.8	Modifying CONFIG.SYS to Install a Driver	E-14

Appendix F LOGICAL KEYBOARD PROTOCOL

F.1	Introduction	F-1
F.2	Simple Characters	F-1
	The WISCII I Character Set	F-1
	Control Characters	F-7
F.3	Special Characters	F-8
F.4	Dead Keys	F-9

CONTENTS (continued)

Appendix G SCREEN MANIPULATION

G.1	Introduction	G-1
G.2	Control Codes	G-1
G.3	Escape Code Sequences	G-1
	Set Graphics Rendition	G-2
	Switch Default Screen to Specified Screen	G-3
	Set or Clear Selected Modes and LED Control Codes	G-4
	Clear All or Part of the Screen	G-5
	Clear All or Part of Current Line	G-5
	Position Cursor at Row and Column	G-6
	Move Cursor Up, Down, Right, or Left	G-6
	Save Cursor Position/Restore Cursor to Saved Position ...	G-7
	Insert Line(s) at Current Line	G-7
	Delete Line(s) at Current Line	G-8
	Scroll Portion of the Screen	G-8
	Delete Characters from Current Line	G-9
	Reset to Startup Configuration	G-9
	Set Palette Entry Attributes	G-9

Appendix H PRINTER CHARACTER SETS, CODES, AND ESCAPE SEQUENCES

H.1	Introduction	H-1
H.2	The Daisy Wheel Printer	H-1
	The Daisy Printer Character Set	H-1
	Daisy Printer Control Codes and Escape Sequences	H-3
	Daisy Printer Return Codes	H-5
H.3	The Matrix Printer	H-5
	The Matrix Printer Character Set	H-5
	Matrix Printer Control Codes and Escape Sequences	H-8
	Matrix Printer Return Codes	H-9

Appendix I IBM TO WANG CODE CONVERSION

I.1	Introduction	I-1
I.2	Keyboard, Printer, and Screen Protocols	I-1
	Keyboard Values	I-2
I.3	Transporting Text Files from the IBM PC to the Wang PC	I-3

CONTENTS (continued)

Appendix J	CONTENTS OF THE WANG PC SYSTEM SOFTWARE DISKETTE	J-1
Appendix K	ABBREVIATIONS GLOSSARY	K-1
INDEX		Index-1

FIGURES

Figure 1-1	The Program Development Menu	1-3
Figure 2-1	Linker Input and Output	2-2
Figure 3-1	Library Manager Operations	3-3
Figure A-1	General Registers	A-2
Figure A-2	The Relations of Logical Segments to Physical Memory	A-3
Figure A-3	Flags	A-4
Figure A-4	Storage of Word Variables	A-5
Figure A-5	Storage of an Address in a Doubleword	A-5
Figure A-6	Logical to Physical Address Generation	A-6
Figure A-7	DOS and Its Resources	A-8
Figure A-8	DOS Memory Usage	A-12
Figure A-9	The Program Segment Header	A-14
Figure A-10	File Control Block	A-20
Figure A-11	Startup Sector BIOS Parameter Block Variables (512-byte Sectors)	A-23
Figure A-12	Startup Sector BIOS Parameter Block Variables (256-byte Sectors)	A-24

TABLES

Table 4-1	Debugger Commands	4-4
Table 4-2	Flags and Flag Codes	4-31
Table A-1	Interrupt Vectors	A-13
Table A-2	Directory Entries	A-26
Table A-3	File Attribute Codes	A-26
Table F-1	The WISCII I Character Set	F-2
Table F-2	Control Characters	F-7
Table F-3	Special Character Keys	F-8
Table G-1	Graphics Rendition Escape Sequences	G-2
Table G-2	Screen Modes and LED Control Codes	G-4
Table G-3	Combinations of Palette Attributes	G-11
Table H-1	WISCII I Character Set	H-2
Table H-2	Daisy Printer Escape Sequences	H-3
Table H-3	Daisy Printer Return Codes	H-5
Table H-4	Matrix Printer Characters and Hexadecimal Codes	H-6
Table H-5	Matrix Printer Character Names and Corresponding Codes	H-7
Table H-6	Matrix Printer Escape Sequences	H-8
Table H-7	Matrix Printer Return Codes	H-9

CHAPTER 1 THE EDITOR

1.1	Introduction	1-1
1.2	Overview of the Editor	1-1
	The Window	1-1
	Line Numbers	1-2
	Buffer Size	1-2
	Help Screens	1-2
1.3	How to Invoke the Editor	1-2
1.4	Configuration Parameters	1-4
1.5	Editor Commands	1-5
	BACKSPACE	1-5
	CENTER MODE	1-5
	COPY LINES	1-6
	CURSOR EAST and CURSOR WEST	1-6
	CURSOR NORTH and CURSOR SOUTH	1-6
	DELETE CHARACTER	1-6
	DELETE LINE	1-6
	DELETE MANY LINES	1-7
	END SESSION AND SAVE CHANGES	1-7
	ERASE LINE	1-7
	ERASE TO END	1-7
	GO TO FIRST LINE	1-8
	GO TO LAST LINE	1-8
	GO TO LINE	1-8
	INSERT LINE	1-8
	INSERT MODE	1-8
	LOAD PARTIAL FILE	1-9
	LOAD TEXT	1-9
	MOVE LINES	1-9
	NEXT SCREEN	1-10
	PREVIOUS SCREEN	1-10
	PRINT LINES	1-10
	REPLACE	1-10
	SAVE PARTIAL FILE	1-11
	SEARCH	1-11
	SET TABS	1-11
	SET WIDTH	1-12
	SHIFT + CURSOR EAST and SHIFT + CURSOR WEST	1-12
	SHIFT + NEXT and SHIFT + PREV	1-12
	TAB	1-12

CHAPTER 1 THE EDITOR

1.1 INTRODUCTION

This chapter describes the Wang PC Editor (PCEDIT.EXE). You use the Editor to create and modify source files for programs in the compiled languages and the Assembler. However, you can also use the Editor to create and modify text files, such as memos, reports, and lists.

When you run the Editor, the screen contains an area called a "window." To use the Editor, move a section of text into the window with the cursor control, PREV, or NEXT keys. Then, you edit the text within the window by moving the cursor to the desired position and using the Editor commands.

1.2 OVERVIEW OF THE EDITOR

This section gives an overview of the Editor. It describes the window, line numbering, the Editor's use of memory, and Editor Help screens.

1.2.1 The Window

The Editor's window is 21 lines long. The top four lines are reserved for header, status, prompt, and tab lines. Two modes of editing within the window exist: center mode and floating cursor mode. In center mode, the cursor line is always in the center of the screen. Therefore, you can only edit the text in the screen's center. To scroll the screen up or down and move a new line to the center of the screen, press the North or South cursor control key. Center mode is the default mode when you begin editing a file.

In floating cursor mode, you can move the cursor to any line in the window and edit the text in that line. To scroll one new line at a time into the window, position the cursor at the top line and press the North cursor control key, or position the cursor at the bottom line and press the South cursor control key. In either mode, you can scroll the preceding or succeeding 21 lines into the window by pressing PREV or NEXT, respectively.

1.2.2 Line Numbers

The Editor always numbers lines sequentially from 1. If you insert or delete a line, the Editor renumbers all lines. Line numbers are not part of the text. They are only used as a means to reference a particular line or a group of lines. The line number of the cursor line appears at the top of the screen. Line numbers do not appear beside the lines on the screen.

1.2.3 Buffer Size

When you edit a file, the Editor maintains the entire file in a buffer in memory. The amount of memory available with the operating system and the Editor program resident determines the size of this buffer. The available memory space varies depending on the configuration of your system. If a file is too large to fit in available memory, you can create a new file and use the LOAD PARTIAL FILE command (refer to Section 1.5) to insert a section of an old file into the new.

1.2.4 Help Screens

You issue Editor commands by single keystrokes or shifted keystrokes. To find the keystroke required for a command, press HELP. A Help screen that defines the keystroke for each command appears on the top three lines of the screen. If the command you are looking for is not on this screen, press HELP again. Another Help screen appears. Pressing HELP a number of times displays successive screens and finally brings you back to the original screen. While a Help screen is displayed, the window remains in view, and you can invoke any command.

1.3 HOW TO INVOKE THE EDITOR

You can invoke the Editor from the Program Development Menu or from the DOS Command Processor. To use the first method, select the Program Development option from the Main System Menu. Figure 1-1 is the Program Development Menu.

mm/dd/yy	Wang Professional Computer	hh:mm:ss
P R O G R A M D E V E L O P M E N T M E N U		
Release 1.0		
Select an Item and Proceed		
☒	Basic	
☐	Debugger	
☐	Editor	
☐	Linker	
☐	Library Manager	
☐	Other	
		SPACE BAR - Item select
		EXECUTE - Proceed
		CANCEL - Previous Menu

Figure 1-1. The Program Development Menu

From the Program Development Menu, select the Editor option. A file specification prompt then appears. Enter the specification of the file you want to edit.

The other method of invoking the Editor is to select the DOS Command Processor option from the Main System Menu. The DOS prompt then appears. Respond to the prompt by typing:

```
PCEDIT [<filespec>]
```

If you supply the optional file specification, the Editor automatically loads that file. If you do not supply the file specification, the Editor prompts you for it.

To create a file, invoke the Editor and enter a specification for a file that does not exist on the disk you are accessing. The Editor then creates the file. (You can also create a file using the COPY A FILE System utility or the COPY DOS utility. Refer to The Wang Professional Computer Introductory Guide.)

1.4 CONFIGURATION PARAMETERS

When you enter the file specification, the Editor searches the current and alternate directories for a file named CONFIG.EDT. This file contains default tab and width settings for several different programming languages. The extension to the file name identifies the file's language. For example, the Editor supplies .ASM files with the default tab settings and line length for Assembler. If CONFIG.EDT does not define values for an extension, the Editor uses defaults of 8, 8, 8, 8, 8, 8, 8, 8 for tabs and 80 for width.

NOTE:

If the document you are editing was created with a width greater than 80, you must use the SET WIDTH command (refer to Section 1.5) to override the default width. If you do not reset the width, lines you change are shortened to a width of 80 characters. Valid widths are 1 to 255.

CONFIG.EDT also specifies whether the Editor saves the file in compressed format or uncompressed ASCII format and whether the Editor adds CONTROL + Z to the file as the end-of-file indicator. Compressed format uses less disk space, because a tab replaces every group of contiguous blank spaces preceding an 8-column boundary. CONFIG.EDT specifies uncompressed format as the default.

Some programs that run on the Wang PC require CONTROL + Z as the end-of-file indicator. Because most programs do not require it, the default specified by CONFIG.EDT is for the Editor not to add CONTROL + Z to a file.

You can change the CONFIG.EDT defaults or add defaults for other file name extensions by editing CONFIG.EDT. A CONFIG.EDT command is a text string that consists of one to three characters (the file name extension without the period), followed by a space, a keyword, an equal sign, and then a value or values separated by commas. The keywords are TABS, WIDTH, COMPRESS, and CTRLZ. The values for TAB and WIDTH are decimal numbers. The values for COMPRESS and CTRLZ are TRUE or FALSE.

A sample CONFIG.EDT file would be as follows:

```
ASM TABS=8,8,8,8
ASM WIDTH=120
ASM COMPRESS=FALSE
ASM CTRLZ=FALSE
```

All the defaults defined for various extensions must be contained in one CONFIG.EDT file. If CONFIG.EDT contains more than one definition of a particular value for an extension, the Editor uses the last one.

1.5 EDITOR COMMANDS

The Editor acts on commands that require no input parameters as soon as you press the required key. When you select a command that requires you to supply parameters, a command line with fields for the parameter values appears at the top of the screen. You can edit these fields in the same manner that you edit a text line. Use TAB, BACK TAB, and RETURN to move between fields. When you have entered the values you want, press the EXEC. To abort a command, press CANCEL.

Special Function key 15 serves as a recall key for parameter fields and text lines. Press SF 15 after you invoke a command to update a parameter field with the value the field had the last time you invoked the command during that session. For example, if you invoke the SEARCH command and press SF 15, the last string that you searched for appears in the search field. Press SF 15 after you have typed over text already existing on a line to restore the previous text.

A description of each of the Editor's commands follows.

BACKSPACE

The cursor moves left one column and a space replaces the character at the new cursor position.

Press BACK SPACE.

CENTER MODE

Center mode causes the line you are editing to be fixed at the vertical center of the screen. In this mode, the screen scrolls every time you position the cursor to a new line. Center mode allows you to see the text lines that appear directly above and directly below the line you are currently editing.

Floating cursor mode allows you to edit at any position on the screen. In floating cursor mode, the screen scrolls when you try to position the cursor above the top display line or below the bottom display line. With floating cursor mode, you can view a small portion of text without losing view of the top lines, for example, as you edit the bottom lines. Floating cursor mode is also helpful on systems with a low resolution video board, because scrolling is slower with a low resolution board.

In center mode, the center line of the screen is highlighted. In floating cursor mode, the entire screen is highlighted. Therefore, which mode you are in is always clear from viewing the screen.

Press CENTER while in floating cursor mode to invoke center mode.
Press CENTER while in center mode to invoke floating cursor mode.

COPY LINES

This command copies a block of lines from one specified point in the text to a second specified point. The original lines remain intact at the first location. The command prompts you for a starting line, an ending line, and a target line. The copied lines are inserted after the target line. The default value for each of the three fields is the current line number. To change the line number in any of the fields, use the North or South cursor control key (shifted or unshifted), or NEXT or PREV to position the cursor at the desired line. The line number in the field automatically changes to that of the cursor line. Use TAB, BACKTAB, and RETURN to move between fields. When the values you want appear in the prompt fields, press EXEC, and the copying takes place.

Press COPY.

CURSOR EAST and CURSOR WEST

The CURSOR EAST command repositions the cursor one position to the right. The CURSOR WEST command repositions the cursor one position to the left. The cursor line does not change. If the line length is greater than 80, the screen scrolls left by one position or right by one position when the cursor is at the right or left edge of the screen. Scrolling ceases when the cursor reaches the start or end of the line.

Press the East or West cursor control key.

CURSOR NORTH and CURSOR SOUTH

The CURSOR NORTH command repositions the cursor one line above the current line. The CURSOR SOUTH command repositions the cursor one line below the current line. The cursor column does not change. The screen scrolls according to whether center mode or floating cursor mode is in effect (refer to the CENTER MODE command).

Press the North or South cursor control key.

DELETE CHARACTER

This command deletes the character at the current cursor position. The characters on the line to the right of the cursor move one position left. A space is placed in the last column.

Press DELETE.

DELETE LINE

This command deletes the line at the current cursor position.

Press SHIFT + DELETE.

DELETE MANY LINES

This command deletes a block of text lines. The command prompts you to enter the starting line and the ending line. The default for both fields is the current line number. To change the line number in either field, use the North or South cursor control key (shifted or unshifted), or NEXT or PREV to position the cursor at the desired line. The line number in the field automatically changes to that of the cursor line. Use the TAB, BACK TAB, and RETURN keys to move between fields. When the values you want appear in the prompt fields, press the EXEC key to delete the lines you have specified.

Press STOP.

END SESSION AND SAVE CHANGES

This command returns you to the Program Development Menu and gives you the option of saving your changes. The command displays the prompt PRESS EXECUTE AND TEXT WILL BE SAVED, TYPE N AND CHANGES WILL BE LOST. The name of the file being edited also appears on the monitor to remind you which file will be written if you press EXEC. If you press the N or n, all changes are lost.

If you respond to the prompt by typing C or c, the Editor saves the text in compressed format. If you press EXEC or type A or a, the Editor saves the text in uncompressed format (the default). Refer to Section 1.4 for an explanation of compressed and uncompressed formats.

If you invoke this command by mistake, you can abort the command by pressing CANCEL.

Press SHIFT + CANCEL or CONTROL + C.

ERASE LINE

This command erases all characters on the cursor line, replaces them with blanks, and positions the cursor at the beginning of the line.

Press SHIFT + ERASE.

ERASE TO END

This command erases the characters from the current cursor position to the end of the line.

Press ERASE.

GO TO FIRST LINE

This command moves the cursor to the first line of the file. The cursor column does not change.

Press SHIFT + PREV.

GO TO LAST LINE

This command moves the cursor to the last line in the file. The cursor column does not change.

Press SHIFT + NEXT.

GO TO LINE

This prompts you to enter a line number and positions the cursor at the line you specify. The column does not change.

Press GO TO.

INSERT LINE

This command inserts one blank line directly below the current cursor line.

Press EXEC.

INSERT MODE

In insert mode, the Editor inserts the characters that you type into the text at the current cursor position. When the Editor is not in insert mode, the characters you type replace the characters in the positions where they are typed. Inserting causes existing text to move to the right. If you press BACKSPACE while in insert mode, the character to the left of the cursor is deleted.

While insert mode is in effect, the letters Ins appear highlighted in the status line at the top of the screen, and the LED at the far left is on.

Press INSERT while insert mode is off to select insert mode.
Press INSERT while insert mode is on to turn off insert mode.

LOAD PARTIAL FILE

This command reads a range of lines from a file into the text buffer and inserts them into your existing text file after the current cursor line. The command prompts you to enter a file specification. The command also prompts you to enter the starting line and ending line. The default values for these fields are 1 and 9999, respectively. Type the values specifying the range of lines you want to load and press EXEC.

Press SHIFT + INDENT.

LOAD TEXT

This command loads a specified text file into memory from a disk. The command prompts you to enter a file specification. After you enter the specification you want, press EXEC to load the file. If you have made changes to the current text file, a prompt appears asking if the current text area should be saved. The file specification of the current text also appears. Press EXEC to save the changes. If you do not want to save the changes, enter N before pressing EXEC.

If you respond to the prompt by typing C or c, the Editor saves the text in compressed format. Press EXEC or type A or a to save the text in uncompressed format (the default). Refer to Section 1.4 for an explanation of compressed and uncompressed formats.

Press INDENT.

MOVE LINES

This command moves a block of lines from one specified point in the text and inserts them at a second specified point. The original lines are deleted. The command prompts you for a starting line, an ending line and a target line. The lines being moved are inserted after the target line. The default value for each of the three fields is the current line number. To change the line number in any field, use the North or South cursor control key (shifted or unshifted), or NEXT or PREV to position the cursor at the desired line. The line number in the field automatically changes to that of the cursor line. Use TAB, BACKTAB, and RETURN to move between fields. When the values you want appear in the prompt fields, press EXEC, and the lines are moved.

Press MOVE.

NEXT SCREEN

This command moves the entire next screen into the window. The cursor position remains the same, unless the cursor would be positioned beyond the last line of text on the new screen. In that case, the cursor is positioned on the last line of text with the column unchanged.

Press NEXT.

PREVIOUS SCREEN

This command moves the entire previous screen into the window. The cursor position remains the same, unless the cursor would be positioned before the first line of text on the screen. In that case, the cursor is positioned on the first line with the column unchanged.

Press PREV.

PRINT LINES

This command sends a block of text lines to the printer. The command displays a prompt for the starting line and the ending line. The default for both fields is the current line number. To change the line number in either field, use the North or South cursor control key (shifted or unshifted), or NEXT or PREV to position the cursor at the desired line. The line number in the field automatically changes to that of the cursor line. Use TAB, BACK TAB, and RETURN to move between fields. When the values you want appear in the prompt fields, press EXEC to print the lines.

Press PRINT.

REPLACE

This command searches for a string of characters and replaces it with another string. First, the command prompts you to enter a search string and a replace string. Then, the Editor searches for the first string, beginning at column 1 of the current cursor line. The search only recognizes characters that have the same case as the characters in the search string you entered. If the Editor finds the string, it displays the prompt "Replace Y or N" with the default response of Y. To replace the string, press EXEC. If you do not want to replace this occurrence of the string, enter N and press EXEC. In either case, the search continues for another occurrence of the search string. To terminate the command, press CANCEL.

NOTE:

If the replacement string is longer than the search string, characters at the end of the line can be lost. A message appears informing you that the replacement will result in a loss of characters.

If you want to replace all matches without receiving a prompt for each occurrence of the search string, invoke this command by pressing SHIFT + REPLACE.

Press REPLACE or SHIFT + REPLACE.

SAVE PARTIAL FILE

This command writes a range of lines from memory to disk. The command prompts you to enter a file specification. It also prompts you for the starting line, and the ending line. The default for both fields is the current line number. To change the line number in either field, use the North or South cursor control key (shifted or unshifted), or NEXT or PREV to position the cursor at the desired line. The line number in the field automatically changes to that of the cursor line. Use TAB, BACKTAB, and RETURN to move between fields. When you press EXEC, the lines are written to the specified file.

NOTE:

If the specified file already exists, this command deletes the previous version.

Press SHIFT + GO TO.

SEARCH

This command prompts you to enter a string of characters and searches for the first occurrence of that string, beginning at the line following the current cursor line. If the search finds the string, the line in which the text occurs becomes the new cursor line, and the line is highlighted in the center of the screen. To find the next occurrence of the string, press EXEC. If the search does not find the string, a message is displayed, and the cursor position does not change. If you want the search to find both uppercase and lowercase occurrences of a string, invoke this command by pressing SHIFT + SEARCH.

Press SEARCH or SHIFT + SEARCH.

SET TABS

This command allows you to set and modify tab positions. When you invoke the command, the cursor is positioned at column zero in the tab line. Use the East and West cursor control keys to move the cursor to the desired tab position. To set a tab, press TAB or T (shifted or unshifted). A tab indicator then appears at the cursor position in the tab line. To erase a previously set tab, type over the tab indicator with the space bar or a hyphen (-).

Press DEC TAB.

SET WIDTH

This command allows you to change the line width. When you invoke the command, it prompts you to enter the new line width. Type a number between 1 and 255 and then press EXEC.

NOTE:

This command does not reformat the text. If you shorten the line width, existing lines can be truncated. However, the text is not lost. You can recover it by resetting the line width to the previous value.

Press FORMAT.

SHIFT + CURSOR EAST and SHIFT + CURSOR WEST

The SHIFT + CURSOR EAST command repositions the cursor to the last character on the line. The SHIFT + CURSOR WEST command repositions the cursor to column one. The cursor line does not change. The screen scrolls left or right if the new cursor position is not on the currently displayed screen.

Press SHIFT + the East or West cursor control key.

SHIFT + NEXT and SHIFT + PREV

The SHIFT + NEXT command repositions the cursor to the last line of the file being edited. The SHIFT + PREV command repositions the cursor to the first line of the file being edited. The cursor column does not change. The screen scrolls according to whether center mode or floating cursor mode is in effect (refer to the CENTER MODE command).

Press SHIFT + NEXT or SHIFT + PREV.

TAB

This command moves the cursor to the next tab stop. If there are no tab stops to the right of the cursor, the cursor moves one space to the right. Tabs do not occur in the actual text; therefore, you cannot use the Search command to find them.

Press TAB.

CHAPTER 2 THE LINKER

2.1	Introduction	2-1
	Overview of Linker Operation	2-1
	Linker System Requirements	2-2
2.2	Segments, Groups, and Classes	2-3
	Definitions	2-3
	How the Linker Combines and Arranges Segments	2-4
2.3	Files the Linker Uses	2-6
	Input Files	2-6
	Output Files	2-7
	VM.TMP File	2-7
2.4	Running the Linker	2-7
	Method 1	2-8
	Method 2	2-8
	Method 3	2-8
	Method 4	2-9
2.5	Command Prompts	2-10
	Object Modules [.OBJ]: Prompt	2-10
	Run File [First-Object-filename.EXE]: Prompt	2-10
	List File [Run-filename.MAP]: Prompt	2-11
	Libraries []: Prompt	2-11
2.6	Command Characters	2-12
2.7	Switches	2-13
	-DALLOCATE	2-13
	-HIGH	2-14
	-LINENUMBERS	2-14
	-MAP	2-14
	-PAUSE	2-14
	-STACK:<number>	2-15

CHAPTER 2 THE LINKER

2.1 INTRODUCTION

The Linker utility program (LINK.EXE) is a relocatable linker designed to link separately produced modules of object code. The object code modules must be 8086 files only.

The Linker prompts you for all input. Your answers to the prompts are the commands for the Linker.

The Linker uses a dictionary-indexed library search method, which substantially reduces link time for sessions involving library searches.

2.1.1 Overview of Linker Operation

The Linker combines several separately-compiled object modules into one relocatable load module or run file. All addresses in the run file are relative to one of the four 8086 segment registers. (Refer to Section A.1.2.) The operating system can load the run file at any convenient 16-byte boundary.

In assigning the object modules their relative addresses, the Linker resolves references to symbols external to a particular module. The Linker can search multiple library files for definitions of any external references left unresolved. Also, the Linker produces a list file that shows resolved external references and any error messages.

The Linker establishes the relative positions of the modules it links so that the linked program uses as little memory as possible. When it exhausts available memory, the Linker creates a temporary disk file (VM.TMP) and acts as a virtual linker. The total size of the linked files, however, cannot be more than 384KB.

Figure 2-1 illustrates the operation of the Linker in terms of its input and output.

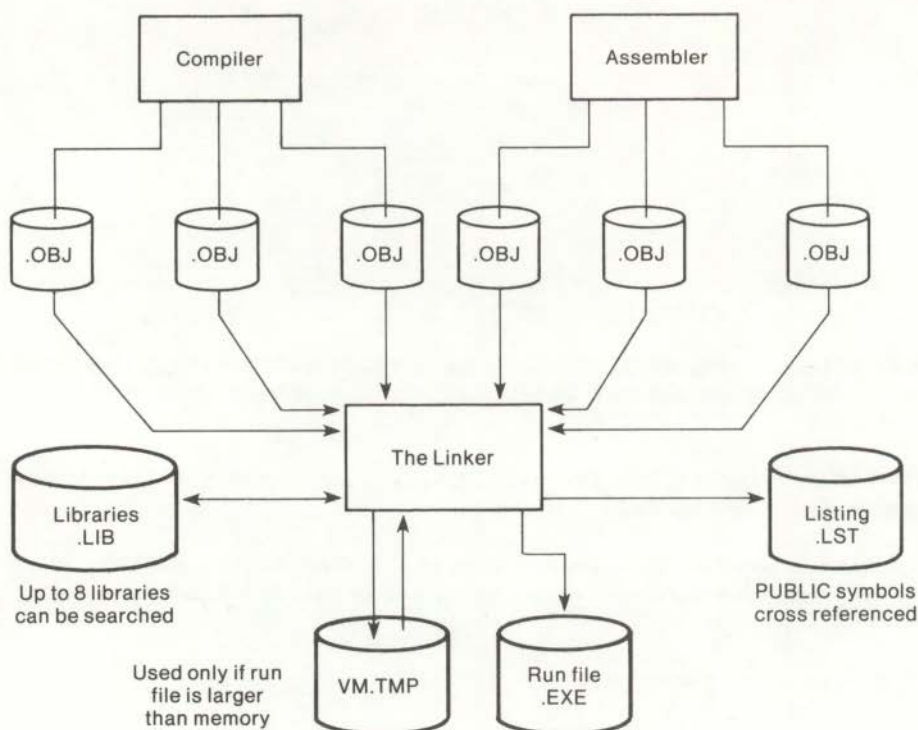


Figure 2-1. Linker Input and Output

2.1.2 Linker System Requirements

The Linker utility requires a minimum of 50KB of memory, 40KB for code and data, and 10KB for run space.

Systems with a single drive can use the Linker, if and only if output from the Linker is sent to the same physical disk from which the input is taken. The Linker does not allow you to swap disks during operation on a single-drive configuration. Therefore, a dual-drive system is a more practical configuration. However, if an input file is not on the current disk, the Linker prompts you to insert a new diskette before it continues its processing.

2.2 SEGMENTS, GROUPS, AND CLASSES

This section describes the underlying functioning of the Linker. It explains segments, groups, and classes, which are the units that the Linker manipulates. And it explains how the Linker combines and arranges segments.

2.2.1 Definitions

A segment is a program module or a collection of modules each of which has the same segment name. A segment can be all or a part of a program and is the smallest unit the Linker manipulates.

In Assembler, you assign modules to a segment by giving them the same segment name. In the compiled languages, the compiler assigns the segment name.

The contents of a segment occupy a contiguous area of memory, up to 64KB long. The segment contents are addressed by a single segment register plus an offset. A segment can begin on a 16-byte boundary anywhere in 8086 memory.

A group is a collection of segments that have different segment names but which are addressed from the same segment register. The segments within a group can occupy noncontiguous areas of memory.

In Assembler, you assign the segments to a group by giving them the same group name. In the compiled languages, the compiler assigns the group name.

Since all segments in the group are addressed by a single segment register plus an offset, the maximum size of the group is 64KB. The Linker ensures that the object modules of a group meet the 64K-byte constraint.

A class, like a group, is a collection of segments with different segment names. Unlike a group, the segments of a class do occupy a contiguous area of memory. However, placing segments in the same class does not affect their addressing. That is, segments belonging to the same class are not automatically addressed from a single segment register.

Segments are named to a class at compile time or assembly time. In Assembler, you give the segments the same class name. In the compiled languages, the compiler assigns the class name.

The naming of segments to a class controls the order and the relative placement of segments in memory. One class precedes another in memory if a segment of the first class precedes all segments of the second class in the input to the Linker. Thus, once the Linker accepts a segment from a class as input, it places all other segments of the same class contiguously in memory before placing segments of any other classes in memory. The Linker loads the segments of a class in the order in which it encounters the segments in the object files. Classes can be loaded across 64K-byte boundaries.

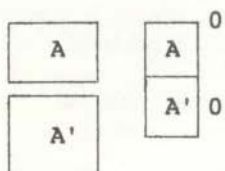
For example, assume class CODE contains segments A and D and class DATA contains segments B and C. If the first segment found by the Linker is A, the Linker places the segments belonging to class CODE in a contiguous area of memory, preceding the segments of class DATA. But if segments A and C belong to group G, their addresses have the same segment register, even though they occupy noncontiguous areas of memory.

2.2.2 How the Linker Combines and Arranges Segments

The Linker works with four "combine types," which you declare in the source module for the Assembler or compiler: private, public, stack, and common. (The memory combine type available in Microsoft's MACRO-86 is treated the same as the public combine type. The Linker does not automatically place memory combine type segments at the highest addresses.)

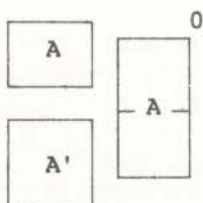
The Linker combines segments for these combine types as follows:

Private



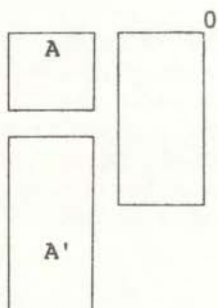
Private segments are loaded separately and remain separate. They can be physically contiguous, but not logically, even if the segments have the same name. Each private segment has its own base address.

Public



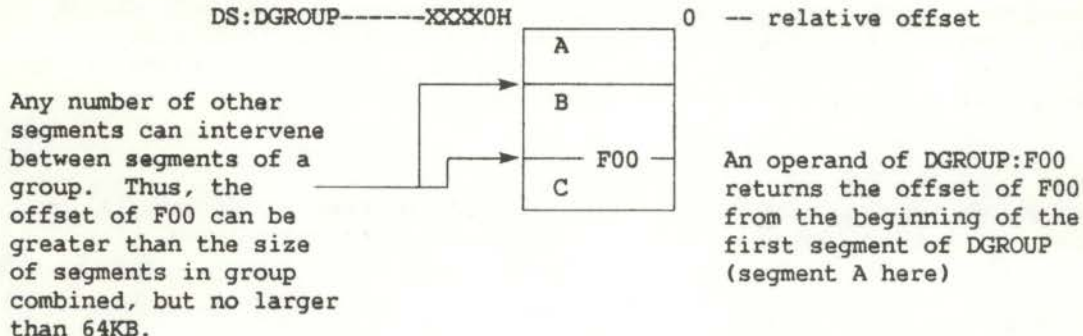
Public segments of the same name and class name are loaded contiguously. The offset is from beginning of the first segment loaded through last segment loaded. Only one base address exists for all public segments of the same name and class name. The stack and memory combine types are treated the same as public. However, the Stack Pointer is set to the first address of the first stack segment. (Public segments of the same segment name, but different class names, may or may not be loaded contiguously.)

Common



Common segments of the same name and class name are loaded overlapping one another. That is, one base address exists for all common segments of the same name. Only one of the segments is in memory at any time. When the system loads another segment at the same address, the previous segment is overwritten. The length of the common area is the length of the longest segment. (The initial values of variables in a common segment may be indeterminate.)

Placing segments in a group in the Assembler provides offset addressing of items from a single base address for all segments in that group.



The Linker loads segments by declared class names. It loads all the segments belonging to the first class name encountered, then loads all the segments of the next class name encountered, and so on until it has loaded all the classes.

If an Assembler program contains:

```
A SEGMENT 'F00'
B SEGMENT 'BAZ'
C SEGMENT 'BAZ'
D SEGMENT 'Z00'
E SEGMENT 'F00'
```

They are loaded as:

```
'F00
A
E
'BAZ'
B
C
'Z00'
D
```

If you are writing Assembler programs, you can exercise control over the ordering of classes in memory by writing a dummy module and listing it first in response to the Linker's Object Modules prompt. The dummy module declares segments into classes in the order you want the classes loaded. For example:

```
A SEGMENT 'CODE'
A ENDS
B SEGMENT 'CONST'
B ENDS
C SEGMENT 'DATA'
C ENDS
D SEGMENT STACK 'STACK'
D ENDS
E SEGMENT 'MEMORY'
E ENDS
```

Be careful to declare all classes you use in your program in this module. If you do not, you lose absolute control over the ordering of classes.

Also, if you want to load segments with the memory combine type as the last segments of your program, simply add MEMORY between SEGMENT and 'MEMORY' in the E segment line above. Note, however, that the Linker loads these segments last only because you impose this control on them, not because of any inherent capability in the Linker or Assembler operations.

CAUTION:

Do not use this method with BASIC, COBOL, FORTRAN, or Pascal programs. Allow the compiler and the Linker to perform their tasks in the normal way.

2.3 FILES THE LINKER USES

The Linker works with one or more input files, produces two output files, can create a virtual memory file, and can be directed to search one to eight library files. For each file type, you can give a 3-part file specification. The format for the Linker file specifications is as follows:

[<d:>]<filename>[.<ext>]

<d:> is the drive designation. Permissible drive designations for the Linker are A: through O:. The colon is always required as part of the drive designation. If you do not enter a drive designation, the default drive is used.

<filename> is any legal file name 1- to 8-characters long.

.<ext> is a 1- to 3-character extension to the file name. The period is always required as part of the extension.

2.3.1 Input Files

If you do not give extensions in the input (object and library) file specifications, the Linker recognizes the following by default: .OBJ for object files and .LIB for library files.

2.3.2 Output Files

The Linker appends the following default extensions to the output (run and list) files: .EXE to run files and .MAP to list files. You cannot override the .EXE default.

2.3.3 VM.TMP File

The Linker uses available memory for the link session. If the linked files create an output file that exceeds available memory, the Linker creates a temporary file and names it VM.TMP. If the Linker needs to create VM.TMP, it displays the message:

VM.TMP has been created.

Do not change diskette in drive, <d:>

Once this message appears, you must not remove the disk from the default drive until the link session ends. If you remove the disk, the operation of the Linker is unpredictable, and the Linker might return the error message:

Unexpected end of file on VM.TMP

The Linker uses VM.TMP as virtual memory. The contents of VM.TMP are subsequently written to the file named following the Run File prompt. (Refer to Section 2.5.2.) VM.TMP is a working file only, and the Linker deletes it at the end of the link session.

CAUTION:

Do not use VM.TMP as a file name for any file. If you have a file named VM.TMP on the default drive and the Linker requires the VM.TMP file, the Linker deletes the VM.TMP on disk and creates a new VM.TMP. Thus, you lose the contents of the previous VM.TMP.

2.4 RUNNING THE LINKER

Running the Linker requires you to use two types of commands: a command to invoke the Linker and answers to command prompts. In addition, six switches control optional Linker features. Usually, you enter all the commands to the Linker on the keyboard. Alternatively, you can create a response file containing answers to the command prompts and switches. The Linker supports some special command characters to assist you while entering commands.

You can invoke the Linker in four ways. When using the first two methods, you enter the remaining commands as responses to prompts. When using the third method, you enter all commands on the line used to invoke the Linker. The fourth method uses a response file containing all the necessary commands.

2.4.1 Method 1

To use Method 1, select the Program Development option from the Main System Menu. Select the Linker option from the Program Development Menu (Figure 1-1). The Linker then displays four prompts, one at a time. Your answers to the prompts are commands to the Linker to perform specific tasks.

2.4.2 Method 2

To use Method 2, select the DOS Command Processor or Other option from the Main System Menu. The DOS prompt or a file specification prompt then appears. Respond to the prompt by entering the following:

LINK

This response loads the Linker into memory. Then, the Linker returns four prompts, one at a time.

2.4.3 Method 3

To use Method 3, select the DOS Command Processor option from the Main System Menu. The DOS prompt or a file specification prompt then appears. Respond to the prompt by entering the following:

LINK <object-list>,<runfile>,<listfile>,<lib-list>[/switch...]

The entries following LINK are responses to the command prompts described in Section 2.5.

<object list> is a list of object modules, separated by plus signs.

<runfile> is the name of the executable output file.

<listfile> is the name of the file to contain the listing.

<lib-list> is a list of library modules you want to search.

</switch...> are optional switches, which you can place following any of the response entries (before any commas or after the <lib-list>, as shown).

You must use commas to separate the entry fields for the different prompts. To select the default for a field, simply enter a second comma without spaces. For example:

```
LINK FUN+TEXT+TABLE+CARE/P/M,,FUNLIST,COBLIB.LIB
```

This example loads the Linker, then loads the object modules FUN.OBJ, TEXT.OBJ, TABLE.OBJ, and CARE.OBJ. The Linker then pauses (due to the /P switch). When you press any key, the Linker links the object modules, produces a global symbol map (due to the /M switch), produces the default run file FUN.EXE, creates a list file named FUNLIST.MAP, and searches the library file COBLIB.LIB.

2.4.4 Method 4

To use Method 4, select the DOS Command Processor or Other option from the Main System Menu. The DOS prompt then appears. Respond to the prompt by entering the following:

```
LINK @<filespec>
```

<filespec> is the name of a response file. A response file contains answers to the Linker prompts, and can also contain switches. Method 4 permits you to conduct the link session without interactive (direct) responses to Linker prompts.

Before using Method 4 to invoke the Linker, you must first create the response file. A response file consists of text lines, one for each prompt. Responses must appear in the same order as the command prompts appear. Use switches and command characters in the response file the same way as you use them for responses you enter on the keyboard.

When the link session begins, each prompt appears with the responses from the response file. If the response file does not contain answers for all the prompts, either in the form of file names, the semicolon special character, or RETURNS, the Linker, after displaying a prompt that does not have a response, waits for you to enter a legal response. When you have entered a legal response, the Linker continues the link session.

The following is an example of a Linker response file:

```
FUN TEXT TABLE CARE <RETURN>
/PAUSE/MAP <RETURN>
FUNLIST <RETURN>
COBLIB.LIB <RETURN>
```

This response file causes the Linker to load the four object modules. The Linker pauses to permit you to swap disks before creating and producing a public symbol map. (Refer to Section 2.7.5, /PAUSE, before using this feature.) When you press any key, the Linker creates output files named FUN.EXE and FUNLIST.MAP, searches the library file COBLIB.LIB, and uses the default settings for the flags.

2.5 COMMAND PROMPTS

As already explained, you give commands to the Linker by entering responses to four text prompts. When you enter a response to one prompt, the next appears. When you answer the last prompt, the Linker begins linking automatically. When the link session is finished, the Linker returns you to the place from which you invoked it. When the operating system prompt appears, the Linker has finished successfully. If the link session is unsuccessful, the Linker returns an error message.

The Linker prompts you for the names of object, run, list files, and libraries. The prompts are listed in the order they appear. For prompts that can default to preset responses, the default response is shown in square brackets ([]) following the prompt.

2.5.1 Object Modules [.OBJ]: Prompt

The Linker displays the following prompt:

Object Modules [.OBJ]:

Enter a list of object modules you want to link. You must separate the modules by a space or a plus sign. If you enter a plus sign as the last character, the prompt reappears. This prompt has no default file name. You must enter the name of an object file.

The default for the file name extension is .OBJ. If an object module has any other file name extension, you must enter the extension. Otherwise, you can omit the extension. This prompt is followed only by a default extension because it has no default file name and requires you to enter a file name.

When determining the order in which you enter the object modules, remember that the Linker loads segments into classes in the order encountered.

2.5.2 Run File [First-Object-filename.EXE]: Prompt

The Linker displays the following prompt:

Run File [<Object-file>.EXE]:

Enter the file name for the file to receive the executable object code. If you do not enter a response to this prompt, the Linker uses the first file name entered in response to the Object Modules prompt as the run file name with the extension .EXE.

The Linker uses the file name entered here to store the run (executable) file that results from the link session. All run files receive the file name extension .EXE. If you specify a different extension, the Linker ignores it. For example:

```
Run File [FUN.EXE]:  B:PAYROLL.COM/P
```

This response directs the Linker to create the run file PAYROLL.EXE on Drive B:.. Also, the Linker pauses (/P), which allows you to insert a new disk to receive the run file.

2.5.3 List File [Run-Filename.MAP]: Prompt

The Linker displays the following prompt:

```
List File [<Run-file>.MAP]:
```

Enter the file name for the listing file. If you do not enter a response to this prompt, the Linker uses the run file name with the default extension .MAP.

The list file contains an entry for each segment in the input (object and library) modules. Each entry also shows the offset (addressing) in the run file.

2.5.4 Libraries []: Prompt

The Linker displays the following prompt:

```
Libraries [ ]:
```

Enter 1 to 8 library file names you want to search, separated by blank spaces or plus signs (+). If you enter a plus sign as the last character, the prompt reappears. You may also respond by pressing RETURN. RETURN alone means no library search.

Library files are created by the Library Manager. (Refer to Chapter 3, The Library Manager.) The default file name extension for library files is .LIB.

The Linker searches the library files in the order listed to resolve external references. When it finds the module that defines the external symbol, the Linker processes the module as another object module. For multiply defined symbols, the search ceases when it finds the first instance.

If the Linker cannot find a library file on the disk in the disk drives, the following prompt appears:

```
Cannot find library <library-name>
Enter new drive letter:
```

Press the letter for the drive designation.

The Linker does not search within each library file sequentially. Instead, the Linker uses a method called dictionary indexed library search. For each library, the Library Manager creates a table indexing the publicly defined symbols of the modules within the library. The Linker finds definitions for external references by index access rather than searching from the beginning of the file to the end for each reference. This indexed search reduces substantially the link time for any sessions involving library searches.

2.6 COMMAND CHARACTERS

The Linker provides three command characters to aid you in entering commands.

Use the plus sign (+) to separate entries and to extend the current logical line following the Object Modules and Libraries prompts. (You can also use a blank space to separate object modules.)

To enter a large number of responses (each of which can also be very long), enter plus sign/RETURN at the end of the physical line (thus extending the logical line). If the plus sign/RETURN is the last entry following these two prompts, the Linker prompts you to enter more module names. When the Object Modules or Libraries prompt appears again, continue to enter responses. When you have listed all the modules to be linked, be sure the response line ends with a module name and a RETURN and not a plus sign/RETURN.

Example:

```
Object Modules [.OBJ]: FUN TEXT TABLE ARE+<RETURN>
Object Modules [.OBJ]: F00+FLIPFLOP+JUNQUE+<RETURN>
Object Modules [.OBJ]: CORSAIR<RETURN>
```

Use a single semicolon (;) followed immediately by RETURN at any time after the first prompt (from "Run File" on) to select default responses to the remaining prompts. This feature saves time and overrides the need to enter a series of RETURNS.

CAUTION:

Once you have entered the semicolon, you can no longer respond to any prompt for that link session. Therefore, do not use the semicolon to skip some prompts. To skip some prompts, press RETURN.

For example:

```
Object Modules [.OBJ]: FUN TEST TABLE CARE<RETURN>
Run File [FUN.EXE]: ;<RETURN>
```

The remaining prompts do not appear, and the Linker uses the default values (including FUN.MAP for the List File.)

Use SHIFT + CANCEL at any time to abort the link session. If you enter an erroneous response, such as the wrong file name or a misspelled file name, you must press SHIFT + CANCEL to exit the Linker. Then you must invoke the Linker and start the procedure again. If you have typed the error but not entered it, you can retype the characters, but for that line only.

2.7 SWITCHES

At the end of each prompt line, you can enter one or more switches that control optional Linker functions. You can group switches at the end of any one of the prompt responses, or you can scatter them the end of several responses. You must precede each switch with a slash. If you do not include a particular switch, the Linker defaults to not performing the function defined for that switch.

For all switches, you can use an abbreviation, from a single letter through the whole switch name. The only restriction is that an abbreviation must be a sequential substring from the first letter through the last entered; no gaps or transpositions are allowed. For example:

<u>Legal</u>	<u>Illegal</u>
/D	/DSL
/DS	/DAL
/DSA	/DLC
/DSALLOC	/DSALLOCT

2.7.1 /DSALLOCATE

The /DSALLOCATE switch directs the Linker to load all data at the high end of the Data Segment (DS). Otherwise, the Linker loads all data at the low end of the Data Segment. At run time, the DS pointer contains the lowest possible address and allows the entire DS segment to be used. The compilers assign the data portions of programs to a group named DGROUP. The /DSALLOCATE switch locates this group in memory.

The presence of the /DSALLOCATE switch and the absence of the /HIGH switch (refer to Section 2.7.2) permit your program to allocate dynamically any available memory below the area specifically allocated within DGroup, yet to address that memory by the same DS pointer. The application program can dynamically allocate up to 64KB (or the actual amount available) less the amount allocated within DGROUPE. Pascal and FORTRAN programs need this dynamic allocation. Therefore, this switch is required for Pascal and FORTRAN programs.

2.7.2 /HIGH

The /HIGH switch causes the Linker to place the run file as high as possible in memory. Otherwise, the Linker places the run file as low as possible.

CAUTION:

Do not use the /HIGH switch with Pascal or FORTRAN programs.

2.7.3 /LINENUMBERS

The /LINENUMBERS switch directs the Linker to include in the list file the line numbers and addresses of the source statements in the input modules. Otherwise, the Linker does not include line numbers in the list file.

Not all compilers produce object modules that contain line number information. In these cases, of course, the Linker cannot include line numbers.

2.7.4 /MAP

/MAP directs the Linker to list all public (global) symbols defined in the input modules, together with their definitions. If you do not enter /MAP, the Linker only lists errors (with undefined globals included).

The Linker lists the symbols alphabetically. For each symbol, the Linker lists its value and its segment:offset location in the run file. The symbols appear at the end of the list file.

2.7.5 /PAUSE

The /PAUSE switch causes the Linker to pause in the link session and wait for you to press RETURN. Normally, the Linker performs the linking session without stopping from beginning to end. This switch allows you to swap the disks before the Linker produces an output file.

When the Linker encounters the /PAUSE switch, it displays the message:

```
About to generate .EXE file
Change disks <hit any key>
```

The Linker resumes processing when you press any key.

CAUTION:

Do not swap the disk that receives the list file or the disk used for the VM.TMP file, if created.

2.7.6 /STACK:<number>

/STACK:<number> directs the Linker to allocate a stack of the specified size for the run file. <number> represents any positive numeric value (in hexadecimal radix) up to 65536 bytes. If you enter a value from 1 to 511, the Linker uses 512.

If you do not use the /STACK switch for a link session, the Linker calculates the necessary stack size automatically. The compilers and Assemblers provide information in the object modules that allow the Linker to compute the required stack size.

At least one object (input) module must contain a stack allocation statement. If not, the Linker returns the following error message:

```
WARNING: NO STACK STATEMENT
```


CHAPTER 3 THE LIBRARY MANAGER

3.1	Introduction	3-1
	Features and Benefits	3-1
	Overview of Library Manager Operation	3-2
	Library Manager System Requirements	3-4
3.2	Running the Library Manager	3-5
	Method 1	3-5
	Method 2	3-5
	Method 3	3-6
	Method 4	3-7
3.3	Command Prompts	3-7
	Library file: Prompt	3-8
	Operation: Prompt	3-8
	List file: Prompt	3-9
3.4	Command Characters	3-9
	The Plus Sign	3-10
	The Minus Sign	3-10
	The Asterisk	3-10
	Semicolon	3-11
	Ampersand	3-11
	SHIFT + CANCEL	3-12



CHAPTER 3 THE LIBRARY MANAGER

3.1 INTRODUCTION

The Library Manager (LIB.EXE) is a utility program that creates, modifies, and deletes library files used with the Linker. This chapter explains the features and operation of the Library Manager.

3.1.1 Features and Benefits

The Library Manager creates files containing program modules that the Linker utility can link with other modules. (Refer to Chapter 2, The Linker.) Once it creates a library file, the Library Manager can add modules to it, delete modules from it, or create separate object files from library file modules.

The distinction between an object file and a module (or object module) is that a module is part of a file. Therefore, while you must give a file a drive designation, at least by default, and can give it a file name extension, you can give neither of these to object modules. However, the drive designation and extension are used with the library file to which a module belongs.

The Library Manager provides you with a means to create general libraries for a variety of programs or special libraries for specific programs. For example, with the Library Manager you can create a general library for a language compiler. Or, you can create a special library that permits fast linking and possibly more efficient execution, for one program only.

You can modify individual modules within a library by extracting the modules, making changes, and then adding the modules to the library again. You can replace an existing module with a different module or with a new version of an existing module.

3.1.2 Overview of Library Manager Operation

The Library Manager performs two basic actions. First, it changes object files into modules and appends them to a library file. Second, it deletes modules from a library file. These two actions are the basis for the five Library Manager functions:

- Create a library file
- Append an object file as a module of a library
- Delete a module
- Make a separate object file by copying a module
- Replace a module in the library file with a new module

During each library session, the Library Manager first deletes or extracts modules, then appends new ones. This order of execution prevents confusion when a new version of a module replaces an old version in the library file. (The replace function is simply the delete-append functions in succession.) You can specify delete, append, or extract functions in any order.

In a single operation, the Library Manager loads each module into memory, checks it for consistency, and writes it back into the file. The consistency check determines whether or not the module's syntactical form is correct and that its file name is unique. If you delete a module, the Library Manager loads that module, but does not write it back into the file.

When the Library Manager writes back the next module to be retained, it places that module at the end of the last module written. This procedure effectively closes up the file space to keep the library file from growing larger than necessary. When the Library Manager has read through the whole library file, it appends any new modules to the end of the file. The Library Manager also creates an index, which the Linker utility uses to find modules and symbols in the library file. Finally, the Library Manager produces a cross-reference listing of the PUBLIC symbols in the library, if you request such a listing. The Library Manager may take some time to build the library index (up to 20 seconds in some cases).

Figure 3-1 illustrates the various Library Manager operations.

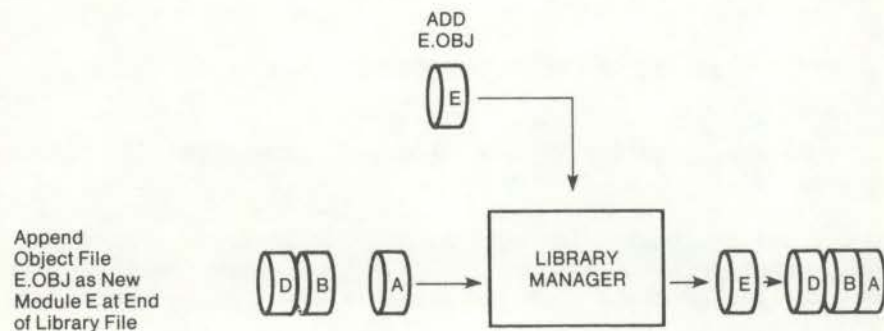
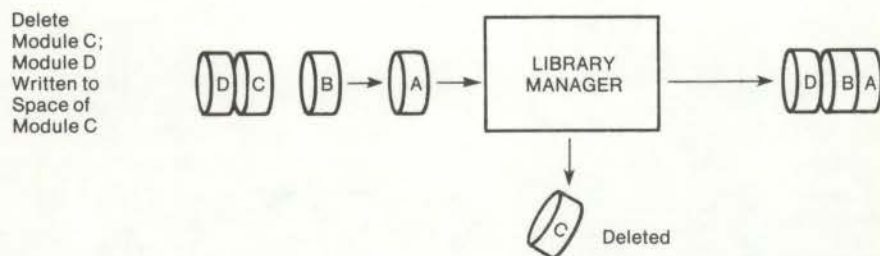
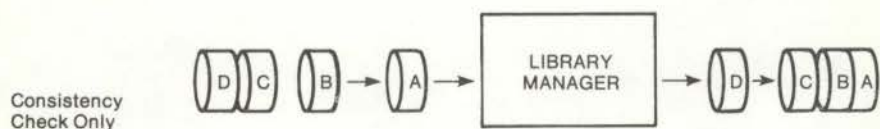
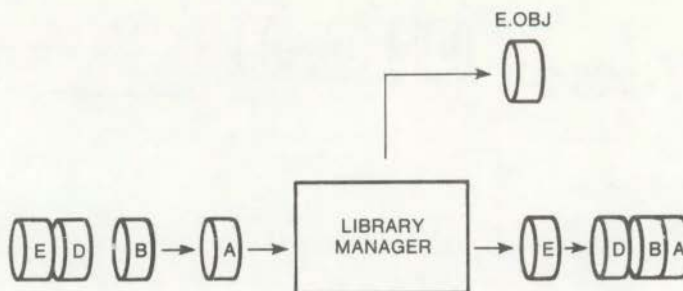


Figure 3-1. Library Manager Operations

Extract
Module E;
Place in a
Separate
Object File;
Return to Library



Consistency
Check; Then
Output a
Cross-Reference
Listing of
Public Symbols

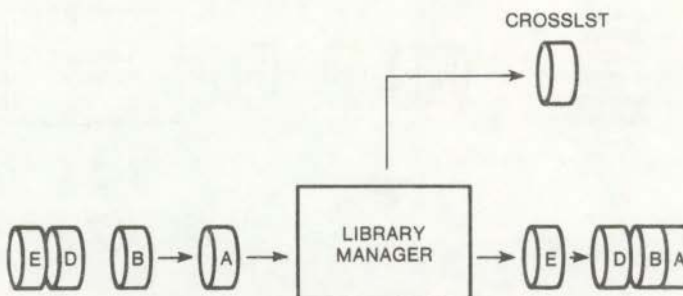


Figure 3-1. Library Manager Operations (continued)

3.1.3 Library Manager System Requirements

The Library Manager requires 38KB of memory minimum, 28KB for code and 10KB for run space.

You can use the Library Manager with a single-drive system if and only if you send output to the same physical disk from which the input was taken. None of the utility programs in this package allow time to swap disks during operation on a single-drive system.

3.2 RUNNING THE LIBRARY MANAGER

Running the Library Manager requires two types of commands: a command to invoke the Library Manager and answers to command prompts. Usually, you enter all the commands on the keyboard. As an option, you can create a response file containing answers to the command prompts. The Library Manager supports special command characters for entering the commands.

You can invoke the Library Manager in four ways. When you use the first two methods, you enter the remaining commands as answers to individual prompts. The third method allows you to enter all commands on the line used to invoke the Library Manager. The fourth method uses a response file containing all the necessary commands.

3.2.1 Method 1

To use Method 1, select the Program Development option from the Main System Menu. Select the Library Manager option from the Program Development Menu (Figure 1-1). The Library Manager then displays a series of four prompts, one at a time. Your answers to the prompts are commands to the Library Manager to perform specific tasks.

3.2.2 Method 2

To use Method 2, select the DOS Command Processor or Other option from the Main System Menu. The DOS prompt or a file specification prompt then appears. Respond to the prompt by entering the following:

```
>LIB
```

This response loads the Library Manager into memory. Then, the Library Manager returns a series of four prompts that appear one at a time. Your answers to the prompts are commands to the Library Manager to perform specific tasks.

The following is an example of responses to Methods 1 and 2 prompts:

```
Library file:pascal  
Operation:-heap+heap;  
List file:
```

These responses delete the old library module HEAP from the library file, then add a new file HEAP.OBJ as the last module in the library.

3.2.3 Method 3

To use Method 3, select the DOS Command Processor or Other option from the Main System Menu. The DOS prompt or a file specification prompt then appears. Respond to the prompt by entering the following:

LIB <library><operations>,<listing>

The entries following LIB are responses to the command prompts.

<library> is the file specification of a library file.

<operations> is a series of module or object file names, each preceded by one of three command characters: the plus sign (+), the minus sign (-), and the asterisk (*).

<listing> is the file specification for the file to contain the cross-reference listing of PUBLIC symbols in the modules in the library.

A command character must precede all operations entries. If you want a cross-reference listing, you must use a comma to separate the list file specification from the last operations entry. If you have many operations to perform during a library session, use the ampersand (&) command character to repeat the command prompt so that you can enter additional object file specifications and module names. (Refer to Section 3.4.5, Ampersand.)

The following are examples of Method 3.

LIB PASCAL-HEAP+HEAP;

This command causes the Library Manager to delete the module HEAP from the library file PASCAL.LIB, then append the object file HEAP.OBJ as the last module of PASCAL.LIB (the module is given the name HEAP).

LIB PASCAL

This command causes the Library Manager to perform a consistency check of the library file PASCAL.LIB. No other action is performed.

LIB PASCAL,PASCROSS.PUB

This command causes the Library Manager to perform a consistency check of the library file PASCAL.LIB, then produce a cross-reference listing file named PASCROSS.PUB.

3.2.4 Method 4

To use Method 4, select the DOS Command Processor or Other option from the Main System Menu. The DOS prompt or a file specification prompt then appears. Respond to the prompt by entering the following:

```
LIB @<filespec>
```

Method 4 permits you to conduct the Library Manager session without interactive responses to the Library Manager prompts. The @<filespec> designates a response file. A response file contains answers to the Library Manager prompts. Before using Method 4 to invoke the Library Manager, you must first create the response file.

A response file contains text lines, one for each prompt. You must arrange the responses in the same order as the command prompts appear. Use command characters in the response file the same way you use them for responses you enter on the keyboard.

When the library session begins, the Library Manager displays each prompt in turn with the responses from the response file. If the response file does not contain responses to all the prompts, the Library Manager uses the default responses.

The following is an example of a response file for Method 4:

```
PASCAL<RETURN>
-HEAP*FOIBLES+CROSSREF+HEAP<RETURN>
CROSSREF<RETURN>
```

This response file causes the Library Manager to delete the module HEAP from the PASCAL.LIB library file, extract the module FOIBLES and place it in an object file named FOIBLES.OBJ, then append the object files CURSOR.OBJ and HEAP.OBJ as the last two modules in the library. Then, the Library Manager creates a cross-reference file named CROSSREF.

3.3 COMMAND PROMPTS

As explained already, you can give commands to the Library Manager by entering responses to three text prompts. When you enter your response to the current prompt, the next prompt appears. When you answer the last prompt, the Library Manager proceeds with its operations. When the library session finishes successfully, the Library Manager returns you to the place from which you invoked it. If the library session is unsuccessful, the Library Manager returns an error message. (Refer to Section B.3, Library Manager Messages.)

The Library Manager prompts you to enter the name of the library file, the operation you want to perform, and the specification you want to give for a cross-reference listing file, if any. The following sections explain each of these prompts.

3.3.1 Library File: Prompt

The Library Manager displays the following prompt:

Library file:

Enter the file specification of the library file that you want to use. The Library Manager assumes that the file name extension is .LIB. You can override this extension by giving a file name extension when you enter the library file name. Because the Library Manager can manage only one library file at a time, you cannot enter more than one file name in response to this prompt. Additional responses, except the semicolon command character, are ignored.

If you enter a library file name followed immediately by a semicolon command character, the Library Manager performs a consistency check only, then returns to the operating system. The Library Manager reports any errors in the file.

If the file name you enter does not exist, the Library Manager returns the prompt:

Library file does not exist. Create?

The Library Manager checks the response for the letter Y as the first character. If you enter any other character first, the Library Manager terminates and returns to the operating system.

3.3.2 Operation: Prompt

The Library Manager displays the following prompt:

Operation:

Enter one or more command characters for manipulating modules (+, -, and *), each followed immediately (no space) by a module name or an object file specification. The plus sign appends an object file as the last module in the library file. The minus sign deletes a module from the library file. The asterisk extracts a module from the library and places it in a separate object file with the file name taken from the module name and a file name extension .OBJ. There is no default action.

The Library Manager allows you to enter operations on modules and object files in any order you want. Be sure to always include one of the command characters for operations before the name of each module or object file.

More information about order of execution and what the Library Manager does with each module is given in Section 3.4, Command Characters.

NOTE:

The Library Manager allows you to enter your Operation responses in any order. The order of execution is always that described in Section 3.1.2, Overview of Library Manager Operation.

3.3.3 List File: Prompt

The Library Manager displays the following prompt:

List file:

Enter a file specification for a cross-reference listing file if you want a cross-reference list of the PUBLIC symbols in the modules in the library file. If you do not enter a file specification, the Library Manager does not generate a cross-reference list.

Because the response to the list file prompt is a file specification, you can specify, along with the file name, a drive (or device) designation, and a file name extension. The Library Manager does not give the List file a default file name extension.

The cross-reference listing file contains two lists. The first list is an alphabetical listing of all PUBLIC symbols. Each symbol name is followed by the name of its module. The second list is an alphabetical list of the modules in the library. Under each module name is an alphabetical listing of the PUBLIC symbols in that module. The Library Manager produces the list after all other module operations take place.

If you enter a library file name followed immediately by a comma and a list file specification, the Library Manager performs its consistency check of the library file, then produces the cross-reference listing file.

3.4 COMMAND CHARACTERS

The Library Manager provides six command characters, three for required responses to prompts and three for optional responses. The first three are the plus sign (+), the minus sign (-), and the asterisk (*). Responses to the Operation prompt require you to use of one or more of these three characters. The characters for optional commands are the semicolon (;), the ampersand (&), and SHIFT + CANCEL.

3.4.1 The Plus Sign

In response to the Operation command, the plus sign (+) followed by an object file specification appends the object file as the last module in the library. When the Library Manager encounters the plus sign, it assumes that the file name extension is .OBJ. You can override this extension by specifying a different one.

The Library Manager removes the drive designation and the extension from the object file specification, leaving only the file name. For example, if the object file to be appended as a module to a library is B:CURSOR.OBJ, you enter the following response to the Operation prompt:

```
+B:CURSOR.OBJ
```

This response causes the Library Manager to remove the B: and the .OBJ, leaving only CURSOR, which becomes a module named CURSOR in the library.

3.4.2 The Minus Sign

The minus sign (-) followed by a module name deletes that module from the library file. The Library Manager then closes up the file space left empty by the deletion. Thus, the Library Manager prevents the file from growing larger than necessary with empty space. (New modules, even replacement modules, are added to the end of the file.)

3.4.3 The Asterisk

The asterisk (*) followed by a module name copies that module from the library file to a separate object file. The module still exists in the library. The module name is used as the file name. The Library Manager adds the default drive designation and the file name extension .OBJ.

For example, assume the module you want extracted is CURSOR, and the default disk drive is A. If your response to the Operation prompt is *CURSOR, the Library Manager copies the module named CURSOR from the library file into an object file. The file specification of the object file is A:CURSOR.OBJ.

You cannot override the drive designation and file name extension. You can, however, rename the file, giving a new file name extension, and/or copy the file to a new disk drive, giving a new file name and/or file name extension.

3.4.4 Semicolon

Use a semicolon (;) followed immediately by RETURN at any time after you respond to the first prompt (Library file:) to select default responses to the remaining prompts. This feature saves time and overrides the need for you to answer additional prompts. For example:

```
Library file: FUN
Operation: +CURSOR;
```

The remaining prompt does not appear, and the Library Manager uses the default value (no cross-reference file).

If you enter a library file name followed immediately by a semicolon, the Library Manager reads through the library file and performs a consistency check. No changes are made to the modules in the library file.

NOTE:

Once you enter the semicolon, you can no longer respond to any of the prompts for that library session. Therefore, do not use the semicolon to skip over some prompts. If you want to ignore some prompts, use RETURN.

3.4.5 Ampersand

You use this command character with the Operation prompt only. The line length for a response to any prompt is 255 characters. For a large number of responses to the Operation prompt, place an ampersand at the end of a line. The Library Manager displays the Operation prompt again on the next line. You can then enter more responses.

Use the ampersand character as many times as needed. The Library Manager can perform many functions during a single library session. The number of modules you can append is limited only by disk space. The number of modules you can replace or copy is also limited only by disk space. The number of modules you can delete is limited only by the number of modules in the library file. For example:

```
Library file: FUN
Operation: -HEAP*FOIBLES*INIT+CURSOR&
Operation: +HEAP+ASSUME+RIDE;
```

The Library Manager deletes the module HEAP, copies the modules FOIBLES and INIT (creating two files, FOIBLES.OBJ and INIT.OBJ), then appends the object files CURSOR, HEAP, ASSUME, and RIDE.

3.4.6 SHIFT + CANCEL

Use SHIFT + CANCEL at any time to abort the library session. If you enter an erroneous response, such as the wrong file name or module name, or a misspelled file name or module name, you must press SHIFT + CANCEL to exit the Library Manager, then invoke the Library Manager again. If you have typed the error, but not entered it, you can delete the erroneous characters, but for that line only.

4.1	Introduction	4-1
4.2	How to Invoke the Debugger	4-1
4.3	Initialization	4-2
4.4	Using the Debugger	4-3
4.5	Debugger Command Parameters	4-4
4.6	Debugger Commands	4-7
	A Command (Assemble)	4-8
	C Command (Compare)	4-11
	D Command (Dump)	4-12
	E Command (Enter)	4-14
	F Command (Fill)	4-16
	G Command (Go)	4-17
	H Command (Hex Arithmetic)	4-19
	I Command (Input)	4-20
	L Command (Load)	4-21
	M Command (Move)	4-23
	N Command (Name)	4-25
	O Command (Output)	4-28
	Q Command (Quit)	4-29
	R Command (Register)	4-30
	S Command (Search)	4-33
	T Command (Trace)	4-34
	U Command (Unassemble)	4-36
	W Command (Write)	4-38
	X Command (Change I/O Device)	4-40



CHAPTER 4 THE DEBUGGER

4.1 INTRODUCTION

The Debugger (DEBUG.COM) provides a controlled testing environment for binary files and executable object files. The Assembler and the compilers produce binary files. The Linker produces executable (.EXE) files.

Although you can use the Debugger for programs written in any compiled language, many of the Debugger functions require that you understand Assembler concepts. Many Debugger functions also require that you are familiar with the information on system architecture, the operating system, and definitions contained in Appendix A.

Note that you use the Editor (refer to Chapter 1) to alter source files. The Debugger is the Editor's counterpart for binary files. The Debugger eliminates the need to reassemble a program to see if a minor change has fixed a problem. While using the Debugger, you can alter the contents of a file or the contents of a CPU register and immediately reexecute a program to check the validity of the changes.

You can abort the Debugger commands at any time by pressing CANCEL. Pressing CONTROL + S suspends the display, so that you can read the display before it scrolls off the screen. To restart the display, press CONTROL + Q. Additionally, the control character and the special editing functions available at the DOS command level apply when you are using the Debugger.

4.2 HOW TO INVOKE THE DEBUGGER

You can invoke the Debugger in three ways. The first way uses the Wang PC menus. From the Main System Menu, select Program Development. The next screen is the Program Development Menu, Figure 1-1.

From the Program Development Menu, select the Debugger option. The Debugger prompt, a right angle bracket (>), then appears on the screen. You respond to this prompt by entering the Debugger commands described in Section 4.6.

The second way to invoke the Debugger is from the DOS Command Processor (COMMAND.COM) option on the Main System Menu. The next screen displays the DOS prompt, A:, where A represents the default drive. Respond to the prompt using the following format:

```
DEBUG [<filespec> [<argument list>]]
```

If you specify a file, the Debugger loads that file into memory. You can specify an <argument list> if <filespec> is present. The <argument list> contains any file name parameters and switches to be passed to the program you are debugging. The Debugger loads the specified file into memory as if it had been invoked from the DOS Command Processor. The Debugger prompt then appears, and you can work on the program by responding to the prompt with the Debugger commands.

The third way to invoke the Debugger is from the Other option on the Main System Menu. When you select Other, a prompt for a file specification appears on the screen. Respond to this prompt as you would to the DOS Command Processor prompt.

With any of these methods, you can invoke the Debugger without specifying a file to be debugged. Select Debug Programs from the Program Development Menu or enter DEBUG in response to the DOS prompt or the Other option's file specification prompt. The Debugger prompt then appears. You can then use the N and L commands described in Section 4.6 to specify a file to be debugged and load it into memory.

4.3 INITIALIZATION

If the program you are debugging is not an .EXE file, the 8086 flags and registers have the following initial values when the Debugger program begins:

- The segment registers (CS, DS, ES, and SS) contain the first segment after the Debugger program.
- The Instruction Pointer (IP) contains 0100H.
- The Stack Pointer (SP) contains the end of the stack segment or the low address of the transient portion of COMMAND.COM (refer to Section A.3.3), whichever is lower. The segment size at offset 6 in the Program Segment Header (PSH) of the program you are debugging is reduced by X'100' to allow for a stack of that size.
- The other registers (AX, BX, CX, DX, BP, SI, and DI) contain zero. If you invoke the Debugger with a <filespec>, however, the BX and CX registers contain the length of the file in bytes.

- The flags are cleared. (Refer to the Register command, Section 4.6.14, for their cleared values.)
- The default disk transfer address is offset 80H in the code segment.

When the Debugger loads an .EXE file, it performs the necessary relocating and sets the segment registers, Instruction Pointer, and Stack Pointer to the values defined in the file. The DS and ES registers, however, contain the address of the PSH at the lowest free segment. The CX register contains zero.

CAUTION:

When you invoke the Debugger, it sets up a PSH at offset 0 in its program work area. If you do not supply a file specification when you invoke the Debugger, it sets up a default PSH. It is acceptable to overwrite the default PSH. However, once you have given the Debugger a file specification, you must not modify the PSH below offset 5CH. Modifications in this area could cause the system to crash when it exits from the Debugger. It is recommended that you do not make any modifications before 100H.

If you specify the appropriate switch when the Linker creates the file (refer to Section 2.7), the Debugger loads the program at the high end of memory.

The Debugger assumes a .HEX file contains an ASCII representation of hexadecimal characters. The Debugger converts these files to binary while loading them.

4.4 USING THE DEBUGGER

Each Debugger command consists of a single letter, followed by one or more parameters.

If a syntax error occurs, the Debugger reprints the command line and indicates the location of the error with an up arrow (↑) and the word error. For example:

```

dcs:100 cs:110
      ↑ error

```

You can enter all commands in any combination of uppercase and lowercase letters.

The Debugger commands are summarized in Table 4-1 and described in detail, with examples, in Section 4.6. The Debugger command parameters are described in Section 4.5.

Table 4-1. Debugger Commands

Function	DEBUG Command
Assemble	A<address>
Compare	C<range> <address>
Dump	D[<address>] D[<range>]
Enter	E<address> [<list>]
Fill	F<range> <list>
Go	G[=<address>] [<address>...]
Hex	H<address> <address>
Input	I<value>
Load	L[<address> [<drive> <record> <value>]]
Move	M<range> <address>
Name	N<filespec>[/S...] [<filespec>[/S...]...]
Output	O<value> <byte>
Quit	Q
Register	R[<register-name>]
Search	S<range> <list>
Trace	T[=<address>] [<value>]
Unassemble	U[<address>] U[<range>]
Write	W[<address> [<drive> <record> <value>]]
Change I/O device	X<device>

4.5 DEBUGGER COMMAND PARAMETERS

As Table 4-1 shows, all the Debugger commands accept parameters, with the exception of the Q command. You can separate parameters with delimiters (spaces or commas), but a delimiter is required only between two consecutive hexadecimal values. Thus, the following commands are equivalent:

```

dcs:100 110
d cs:100 110
d,cs:100,110

```

The following is a list of Debugger parameters and their definitions:

<u>Parameter</u>	<u>Definition</u>								
<address>	Either an alphabetic segment register designation or a 1- to 4-digit segment address, followed by a colon and an offset value. For example: <pre> CS:0100 04BA:0100 </pre> The colon is required between a segment designation (whether numeric or alphabetic) and an offset. You can omit the segment designation or segment address, in which case the default segment is used. DS is the default segment for all commands, except A, G, L, T, U, and W. For the latter commands, the default segment is CS. All numeric values are hexadecimal. Note that <address>es must reference existing memory locations. Otherwise, undefined results occur.								
<byte>	A 1- or 2-digit hexadecimal value to be placed in or read from an address or register.								
<device>	A reserved name for a device.								
<drive>	A 1- or 2-digit hexadecimal value indicating which drive to load a file from or write it to. The valid values are 0 through 3F. The values currently assigned to drives are as follows: <table> <tr> <th><Value></th><th><Drive></th></tr> <tr> <td>0</td><td>A</td></tr> <tr> <td>1</td><td>B</td></tr> <tr> <td>2</td><td>C</td></tr> </table>	<Value>	<Drive>	0	A	1	B	2	C
<Value>	<Drive>								
0	A								
1	B								
2	C								
<filespec>	A 1- to 8-character file name, optionally preceded by a drive designation, optionally followed by a period and a 1- to 3-character extension.								

<u>Parameter</u>	<u>Definition</u>
<list>	A series of <byte> values or of <u>strings</u>. <list> must be the last parameter on the command line. The following example has three items in the <list>, two byte entries and one string entry: <pre>fcs:100 L 10 42 'abc'</pre>
<range>	Either two <address>es (<address> <address>) or one <address>, an L, and a <value> (<address> L <value>), where <value> is the number of lines or bytes the command operates on. If you specify two <address>es, the second must be higher than the first and must be specified by an offset only. For example: <pre>CS:100 110 CS:100 L 10</pre> The following <range> is illegal because the second address includes a register: <pre>CS:100 CS:110 ↑ error</pre> The limit for <range> is 10000H. To specify a <value> of 10000H within four digits, enter 0000 (or 0).
<sector>	A 1- to 4-digit hexadecimal value used to indicate the physical sector on the disk at which loading begins.
<register-name>	Either a 2-character register name or the character F for flags.
<string>	Any number of characters enclosed in quotes. The ASCII values of the characters in the string become a <list> of byte values. Quotes can be either single (') or double ("). Within <string>s, the opposite set of quote marks can appear freely as literals. If the delimiter quote marks must appear within a <string>, you must repeat each quote mark, whether single or double. For example, the following strings are legal: <pre>'This "string" is okay.'</pre> <pre>'This ''string'' is okay.'</pre> However, this string is illegal: <pre>'This 'string' is not okay.'</pre>

These strings are legal:

```
"This 'string' is okay."  
"This ""string"" is okay."
```

However, this string is illegal:

```
"This "string" is not okay."
```

The repeated quotations in the following strings are unnecessary and should not be there, unless you want them to be part of the string:

```
"This is a ''string''."  
'This is a ""string"".'
```

<value> A 1- to 4-digit hexadecimal value used to specify a port number, the number of times a command repeats its functions, or the number of bytes to be operated on.

4.6 DEBUGGER COMMANDS

This section explains the function and syntax of the Debugger commands. It also provides examples for each command.

4.6.1 A Command (Assemble)

Function

This command converts MACRO-86 instructions to binary form and places them in memory at an address you specify.

Format

A <address>

<address> either an alphabetic segment register designation or a 1- to 4-digit segment address, followed by a colon and an offset value. The default segment is CS.

Comments

1. When you enter this command, the Debugger prompt appears on the next line, followed by the address you specified. Respond to the prompt by entering a statement to be assembled and then press RETURN. The Debugger converts the statement to machine code, places the values in memory, and displays the address of the next available memory location.
2. To terminate the command, press RETURN without entering a response to the prompt.
3. The Debugger responds to invalid statements by displaying the following message:

↑ Error

and redisplaying the current assemble address.

4. In general, the syntax of the Assembly language statements used in the A command is standard 8086/8088 Assembly language. The following are minor exceptions to this rule:
 - a. The Debugger assumes that all numeric values you enter are hexadecimal.
 - b. Several prefixes (lock, repeat, and segment override) can appear in one statement, but all prefixes must precede the opcode of the statement. Alternately, you can enter a prefix on a line by itself.
 - c. The Debugger distinguishes between byte and word string instructions as follows:

<u>Byte</u>	<u>Word</u>
LODSB	LODSW
STOSB	STOSW
SCASB	SCASW
MOVSb	MOVSW
CMPSB	CMPSW

- d. The mnemonic for the far return is RETF. The assembler automatically assembles short, near, or far jumps and calls, depending on byte displacement to the destination address. You can override the normal interpretation of the byte displacement with the NEAR or FAR prefix. For example:

```
0100:500 JMP 502      ; a 2 byte short jump
0100:0502 JMP NEAR 505 ; a 3 byte near jump
0100:0505 JMP FAR 50A  ; a 5 byte far jump
```

You can abbreviate the NEAR prefix to NE, but you cannot abbreviate the FAR prefix.

- e. The segment override mnemonics are CS:, DS:, ES:, and SS:.
- f. Operands that could refer to either a byte or word are ambiguous. You must precede them with the prefix WORD PTR or BYTE PTR. You can abbreviate these prefixes by BY and WO. For example:

```
INC     BYTE PTR [BP]
NOT     WO [1234]
```

- g. Operands that address memory directly must be enclosed in square brackets to distinguish them from immediate values. For example:

```
ADD     AX,5      ;add 5 to register AX
ADD     AX,[5]    ;add the contents of memory location 5 to AX
```

- h. Two new pseudo-instructions are included. The DB opcode assembles byte values directly into memory. The DW opcode assembles word values directly into memory. For example:

```
DB      1,2,3,4,"THIS IS AN EXAMPLE"
DB      'THIS IS A QUOTE: "'
DB      "THIS IS A QUOTE: '"
DW      1000,2000,3000,"I CAN DO IT TOO!"
```

- i. All forms of the register indirect commands are supported. For example:

```
ADD     BX,34[BP+2].[SI-1]
POP     [BP+DI]
PUSH    [SI]
```


j. All opcode synonyms are supported. For example:

```
LOOPZ 100
LOOPE 100
JA     200
JNBE   200
```

Example

```
>a 3fa:100
>03fa:0100 MOV BX,AX      ; BB CB
>03fa:0102 CALL [BX].25   ; FF 57 25
```

4.6.2 C Command (Compare)

Function

This command compares the area of memory whose starting point and length is specified by <range> to an area of the same size beginning at <address>. It also displays any differences in the areas.

Format

C<range> <address>

<range>	either two <addresses> or one <address>, an L, and a <value> (<address> L <value>). A second <address> must be higher than the first and must be an offset only.
<address>	either an alphabetic segment register designation or a 1- to 4-digit segment address, followed by a colon and an offset value. The default segment is DS.

Comments

1. If the two areas of memory are identical, no display appears. The C command terminates, and the Debugger prompt returns to the screen.
2. If the two areas are different, they are displayed in the following format:

<address1> <byte1> <byte2> <address2> ...

Examples

The following commands have the same result:

C100,200 300

C100L100 300

Each command compares the block of memory from 100H to 200H with the block of memory from 300H to 400H.

4.6.3 D Command (Dump)Function

This command displays the contents of a series of bytes in memory.

Formats

1. D[<address>]

<address>

either an alphabetic segment register designation or a 1- to 4-digit segment address, followed by a colon and an offset value. The default segment is DS.

2. D<range>

<range>

either two <addresses> or one <address>, an L, and a <value> (<address> L <value>). A second <address> must be higher than the first and must be an offset only.

Comments

1. If you specify a single address only, the Debugger displays the contents of 128 bytes beginning at that address. If you specify a range of addresses, the Debugger displays the contents of the range. If you enter the D command without parameters, the Debugger displays the contents of 128 bytes beginning at the current location in the DS segment. For the first D command, the current location is at offset X'100'. For subsequent commands, the current location is that following the last address displayed.

2. The display has two parts: a part that shows each byte as a hexadecimal value and a part that shows the bytes as ASCII characters. Bytes that do not contain printable characters are denoted by a period (.) in the ASCII portion of the display.

3. Each display line shows 16 bytes with a hyphen between the eighth and ninth bytes.

4. Each displayed line begins on a 16-byte boundary, unless the starting address specified in the command is not at a boundary. In that case, the first line can contain fewer than 16 bytes, and the second line begins at a boundary.

Examples

Assume that you enter the following command:

```
dcS:100 110
```

The following display (here split into two lines to fit it on the page) appears:

```
04BA:0100 42 45 52 54 41 20 54 00-20 42 4F 52 4C 41 4E
      44 BERTA T. BORLAND
04BA:0100 2E
```

Assume that you enter the following command:

```
D
```

The Debugger displays 128 bytes, with each line of the display formatted as described in the Comments above. Each line of the display begins with an address incremented by 16 from the address on the previous line. Subsequent D commands entered without parameters display the bytes immediately following those last displayed.

Assume that you enter the following command:

```
DCS:100 L 20
```

The display is formatted as described above, but all the bytes for 20H bytes are displayed.

Assume that you enter the following command:

```
DCS:100 115
```

The display is formatted as described above, but all the bytes in the range of lines from 100H to 115H in the CS segment are displayed.

4.6.4 E Command (Enter)

Function

This command displays bytes one at a time and allows you to modify their contents.

Format

E<address> [<list>]

- | | |
|-----------|---|
| <address> | either an alphabetic segment register designation or a 1- to 4-digit segment address, followed by a colon and an offset value. The default segment is DS. |
| <list> | a series of <byte> values or of characters enclosed in quotes. |

Comments

1. If you enter the optional <list> of hexadecimal values, the replacement of byte values occurs automatically. If an error occurs, byte values remain unchanged.
2. If you enter the <address> without the optional <list>, the Debugger displays the address and its contents, then repeats the address on the next line and waits for your input. At this point, the E command waits for you to perform one of the following actions:
 - a. Replace a byte value with one that you type in. You simply type the value after the current value, at the cursor position. If the value you type is not a legal hexadecimal value or if you type more than two digits, a bell sounds and the illegal or extra character is not moved to memory.
 - b. Press the space bar to display the next byte. To change the value, simply enter the new value as described in (2a). If you space beyond an 8-byte boundary, the Debugger starts a new display line with the address displayed at the beginning of the line.
 - c. Type a hyphen (-) to return to the preceding byte. If you decide to change a byte before the current byte, typing the hyphen returns the current position to the previous byte. When you type the hyphen, a new line is started with the address of the byte, and its value is displayed.

- d. Press RETURN to terminate the E command. You can press RETURN at any byte position.

CAUTION:

You should not use the E command to change video memory, since video memory is word oriented.

Example

Assume you enter the following command:

ECS:100

The Debugger displays the following:

04BA:0100 EB._

To change this value to 41, enter 41 as follows:

04BA:0100 EB.41_

To step through the subsequent bytes, press the space bar. The following line appears:

04BA:0100 EB.41 10. 00. BC._

To change BC to 42, enter 42 as follows:

04BA:0100 EB.41 10. 00. BC.42_

To change that 10 to 6F, enter the hyphen as many times as needed to return to byte 0101 (value 10), then replace 10 with 6F, as follows:

04BA:0100 EB.41 10. 00. BC.42-
 04BA:0102 00.-
 04BA:0101 10.6F_

Press RETURN to end the E command and return to the Debugger command level.

4.6.5 F Command (Fill)Function

This command fills the addresses in the <range> with the values in the <list>.

Format

F<range> <list>

<range>	either two <addresses> or one <address>, an L, and a <value> (<address> L <value>). A second <address> must be higher than the first and must be an offset only.
<list>	a series of <byte> values or of characters enclosed in quotes.

Comments

1. The F command is similar to multiple E commands in that it allows you to change more than one address at a time. The F command does not abort on an error as the E command does. If any of the memory in the <range> is not valid (bad or nonexistent), the remaining portions of the <range> may receive incorrect data.

2. If the <range> contains more bytes than the number of values in the <list>, the command uses the <list> repeatedly until all bytes in the <range> are filled. If the <list> contains more values than the number of bytes in the <range>, the command ignores the extra values in the <list>.

CAUTION:

You should not use the F command to change video memory, because video memory is word oriented.

Example

Assume you enter the following command:

```
F04BA:100 L 100 42 45 52 54 41
```

The Debugger fills memory locations 04BA:100 through 04BA:200 with the bytes you specified. The five values are repeated until all 100H bytes are filled.

4.6.6 G Command (Go)

Function

This command executes the program currently in memory. Optionally, it stops the execution at a specified breakpoint and displays registers, flags, and the decoded instruction.

Format

G[=<address>] [<address>...]

<address>	either an alphabetic segment register designation or a 1- to 4-digit segment address, followed by a colon and an offset value. The default segment is CS.
-----------	---

Comments

1. If you enter the G command with no parameters, the program executes as if the program had run outside the Debugger.
2. If you specify =<address>, execution begins at that address. If you omit the segment designation from =<address>, the command sets the instruction pointer only. If you include the segment designation in =<address>, the command sets both the CS segment and the instruction pointer. The equal sign (=) is required, so that the Debugger can distinguish the start <address> from the breakpoint <address>es.
3. With the breakpoint <address> list set, execution stops at whatever breakpoint <address> is encountered first, no matter which branch the program takes. When program execution reaches a breakpoint, the registers, the flags, and the decoded instruction are displayed for the next instruction. (The result is the same as if you entered the R command for the breakpoint address.)
4. You can set up to ten breakpoints. Because breakpoints are instruction-oriented, you can set them only at the address of the first byte of an instruction. If you set more than 10 breakpoints, the Debugger returns the BP (Too Many Breakpoints) error message.
5. After an executing program encounters a breakpoint, if you enter the G command again, the program continues executing just as if you had entered the file name at the DOS command level. The only difference is that program execution resumes at the instruction after the breakpoint rather than at the usual start address.

6. The user Stack Pointer must be valid, and the stack must have six bytes available for this command. The G command uses a IRET instruction (Return from Interrupt) to cause a jump to the program under test. The command sets the user Stack Pointer and pushes the user flags, Code Segment register, and Instruction Pointer onto the user stack.

CAUTION:

If the user stack is not valid or is too small, the operating system may crash.

7. The G command places a 1-byte interrupt opcode (03H) at the breakpoint address(es) you specify. The Debugger reserves an area of memory to save the original instructions from those addresses. When the executing program encounters an instruction with the breakpoint code, the Debugger restores all breakpoint addresses to the original instructions. If execution does not stop at one of the breakpoints, the interrupt codes are not replaced with the original instructions.

Example

Assume you enter the following command:

GCS:7550

The program currently in memory executes to address 7550H in the CS segment. The Debugger displays registers and flags. Then, the Debugger prompt appears.

4.6.7 H Command (Hex Arithmetic)

Function

This command performs hexadecimal arithmetic on the two parameters.

Format

H<address> <address>

<address>

either an alphabetic segment register designation or a 1- to 4-digit segment address, followed by a colon and an offset value. The default segment is DS.

Comments

The Debugger adds the two parameter values, then subtracts the second parameter from the first. The results appear on one line, first the sum and then the difference.

Example

Assume you enter the following command:

H10A 19F

The Debugger performs the calculations and then returns the results:

02A9 0095

4.6.8 I Command (Input)

Function

This command inputs and displays one byte from the port specified by <value>.

Format

I<value>

<value> a 1- to 4-digit hexadecimal number.

Comments

A 16-bit port address is allowed.

Example

Assume you enter the following command:

I2F8

Assume also that the byte at the port contains 42H. The Debugger inputs the byte and displays the following value:

42

4.6.9 L Command (Load)

Function

This command loads a file into memory.

Format

L[<address> [<drive> <sector> <value>]]

<address>	either an alphabetic segment register designation or a 1- to 4-digit segment address, followed by a colon and an offset value. The default segment is CS.
<drive>	a 1-digit hexadecimal value.
<sector>	a 1- to 4-digit hexadecimal value used to indicate the physical sector on the disk at which loading begins.
<value>	a 1- to 4-digit hexadecimal number.

Comments

1. You must have named the file either in the Debugger invocation or with the most recently issued N command. Both the invocation and the N commands format a file name properly in the normal format of a File Control Block at CS:5C. (Refer to Sections A.3.5, Program Initialization, and A.4.1, File Control Block.)
2. If you specify a file or part of a file that is too large to fit in available memory, the L command aborts, and the Debugger prompt returns to the screen.
3. If you enter the L command without any parameters, the Debugger loads the file into memory beginning at address CS:100 and sets BX:CX to the number of bytes loaded. If you enter the L command with an <address> parameter, loading begins at the memory address you specify, with BX:CX set to the number of bytes loaded. If you enter L with all parameters, the Debugger loads absolute disk sectors, not a file. The Debugger takes <value> sectors from the <drive> you specify. The drive designation is numeric (0=A:, 1=B:, 2=C:). The Debugger begins loading with the first <sector> you specify and continues until it loads the number of sectors you specify in <value>.

4. If the file has an .EXE extension, then it relocates to the load address specified in the header of the .EXE file. For .EXE files, the Debugger always ignores the <address> parameter of the L command. Note that the header itself is removed from the .EXE file before it is loaded into memory. Thus, the size of a .EXE file on disk differs from its size in memory.

5. If the file you named with the N command or specified on invocation is a .HEX file, then entering the L command with no parameters causes loading of the file beginning at the address specified in the .HEX file. If the L command includes the <address> parameter, the Debugger adds the address specified in the L command to the address found in the .HEX file to determine the start address for loading the file.

Examples

Assume you enter the following commands:

```
A:DEBUG  
>NFILE.COM
```

Now, to load FILE.COM, enter:

```
L
```

The Debugger loads the file and displays the Debugger prompt.

Assume you want to load portions of a disk. To do this, enter:

```
L04ba:100 2 0F 6D
```

The Debugger then loads 109 (6D hex) sectors, starting with absolute sector number 15, into memory beginning at address 04BA:0100. When the Debugger has loaded the sectors, it returns with its prompt.

4.6.10 M Command (Move)Function

This command moves the block of memory specified by <range> to the location beginning at the specified <address>.

Format

M<range> <address>

<range> either two <addresses> or one <address>, an L, and a <value> (<address> L <value>). A second <address> must be higher than the first and must be an offset only.

<address> either an alphabetic segment register designation or a 1- to 4-digit segment address, followed by a colon and an offset value. The default segment is DS.

Comments

The M command performs overlapping moves (where the source and destination areas have some common addresses) without loss of data. It moves the data that could be overwritten first. The sequence for moves from higher addresses to lower addresses is to move the data beginning at the block's lowest address and work towards the highest. The sequence for moves from lower addresses to the higher addresses is to move the data beginning at the block's highest address and work towards the lowest.

CAUTION:

You should not use the M command to change video memory, because video memory is word oriented.

Example

Assume you enter the following:

MCS:100 110 CS:500

The Debugger first moves address CS:110 to address CS:510. Then, it moves CS:10F to CS:50F and so on, until it moves CS:100 to CS:500. To review the results of the move, enter the D command using the <address> entered for the M command.

4.6.11 N Command (Name)Function

This command performs two distinct functions. First, you can use N to assign a file specification for a later L or W command. Second, you can use N to pass file specifications as parameters to the file you are debugging.

Format

N<filespec>[/S...] [<filespec>[/S...]]...

<filespec> a 1- to 8-character file name, optionally preceded by a drive designation, optionally followed by a period and a 1- to 3-character extension.

</S> any DOS switch set for the specified file.

Comments

1. If you invoke the Debugger without naming any file to be debugged, you must enter the N command before the Debugger can load a file.
2. The N command can affect four distinct regions of memory:

DS:5C FCB for file 1
 DS:6C FCB for file 2
 DS:80 Count of characters
 DS:81 All characters entered

Refer to Section A.3.5 for a description of these areas.

3. The Debugger sets up a File Control Block (FCB) for the first file specification in the N command at DS:5C. (Refer to Section A.4.1.) If you give a second file specification, the Debugger sets up an FCB for it beginning at DS:6C. Location DS:80 contains the number of characters you type in the N command (exclusive of the first character, N). The characters in the N command (again, exclusive of the letter N) begin at DS:81. Note that these characters can contain switches and delimiters that are legal in any command typed at the DOS command level.

4. The functions of assigning a file specification for L and W commands and of passing file specification parameters overlap. Consider the following set of the Debugger commands:


```
>NFILE1.EXE
>L
>G
```

Because of the 2-pronged effect of the N command, the following happens:

- a. N assigns the extended file name FILE1.EXE to any later Load or Write commands.
- b. N also assigns the extended file name FILE1.EXE as the first file specification parameter used by any program that is later debugged. (See paragraph D, below.)
- c. L loads FILE1.EXE into memory.
- d. G executes FILE1.EXE. FILE1.EXE also acts as the single file specification parameter (that is, FILE1.EXE executes as if you had typed FILE1.EXE at the DOS command level.)

A more useful set of commands is as follows:

```
>NFILE1.EXE
>L
>NFILE2.DAT FILE3.DAT
>G
```

In this example, N sets FILE1.EXE as the file specification for the subsequent L command. (You can also use the DEBUG command for this purpose.) The L command loads FILE1.EXE into memory. Then, the N command specifies the parameters used by FILE1.EXE. Finally, when the G command executes, FILE1.EXE runs as if you had typed FILE1 FILE2.DAT FILE3.DAT at the DOS command level.

If you enter a W command at this point, the file you are debugging (FILE1.EXE) is saved with the name FILE2.DAT. To avoid such undesired results, always execute an N command before an L or W.

Example

A typical use of the Name command is:

```
DEBUG PROG.COM
NPARAM1 PARAM2/C
G
```

In this example, the G command executes the file in memory as if you entered the following DOS command line:

```
PROG PARAM1 PARAM2/C
```

Testing and debugging, therefore, reflect a normal runtime environment for PROG.COM.

The Debugger loads PROG.COM at DS:100. This file specification is placed in the FCB at DS:5C. Then N formats FCBs for PARAM1 and PARAM2/C at DS:5C and DS:6C, respectively, overwriting the FCB for PROG.COM.

4.6.12 O Command (Output)

Function

This command sends the <byte> specified to the output port specified by <value>.

Format

O<value> <byte>

<value> a 1- to 4-digit hexadecimal number.

<byte> a 2-digit hexadecimal value to be placed in the output port.

Comments

1. A 16-bit port address is allowed.
2. The O command operates on byte-oriented ports only, not on word-oriented ports such as those for the video card.

Example

Assume you enter the following:

O2F8 4F

The Debugger outputs the byte value 4F to output port 2F8.

4.6.13 Q Command (Quit)Function

This command terminates the Debugger program.

Format

Q

Comments

The Q command takes no parameters and exits the Debugger without saving the file currently being operated on. Then, Q returns you to the program from which you invoked the Debugger.

Example

To end the Debugger program, enter the following and then press RETURN:

Q

The Debugger terminates and returns control to the level from which you entered the Debugger.

4.6.14 R Command (Register)Function

This command has three functions. First, R can display the contents of a CPU register and allow you to change its contents. Second, R can display the eight flag settings and allow you to change them. Third, R can display the contents of all the registers, the flag settings, and the next instruction to be executed, in decoded format.

Format

R[<register-name>]

<register-name> either a 2-character register name or the character F for flags.

Comments

1. If you do not enter a <register-name>, the R command dumps the register save area and displays the contents of all registers and flags.

At start up, the segment registers contain the bottom of free memory, the Instruction Pointer contains 0100H, the Stack Pointer contains FFEH, all flags clear, and the remaining registers contain zero.

2. If you enter a register name, the Debugger displays the 16-bit value of that register in hexadecimal, and then a colon appears as a prompt. You then enter a <value> to change the register or press RETURN if you do not want a change.

3. The only valid <register-names> are as follows. (IP and PC both refer to the instruction pointer.)

AX	BP	SS
BX	SI	CS
CX	DI	IP
DX	DS	PC
SP	ES	F

Any other entry for <register-name> results in a BR (Bad Register) error message.

4. If you enter F as the <register-name>, the Debugger displays eight 2-character alphabetic codes. To alter the flag, enter the opposite 2-letter code. The flags are either set or clear.

Table 4-2 lists the flags with their codes for set and clear.

Table 4-2. Flags and Flag Codes

Flag Name	SET	Clear
Overflow	OV	NV
Direction	DN (Decrement)	UP (Increment)
Interrupt	EI (Enabled)	DI (Disabled)
Sign	NG (Negative)	PL (Plus)
Zero	ZR	NZ
Auxiliary Carry	AC	NA
Parity	PE (Even)	PO (Odd)
Carry	CY	NC

5. When you enter the command RF, the Debugger displays the flags in a row at the beginning of a line, in the order shown in Table 4-2. At the end of the list of flags, the Debugger displays a hyphen (-), then the Debugger prompt (>). You can enter new flag values as alphabetic pairs and in any order. You are not required to leave spaces between the flag entries. To exit the R command, press RETURN. Flags for which new values were not entered remain unchanged.

6. If you enter more than one value for a flag, the Debugger returns a DF (Double Flag) error message. If you enter a flag code other than those shown in Table 4-2, the Debugger returns a BF (Bad Flag) error message. In both cases, the flags up to the error in the list are changed. However, flags at and after the error are not changed.

Examples

Assume you enter the following:

R

The Debugger displays all registers, flags, and the decoded instruction for the current location. If the location is CS:11A, then the Debugger might display:

```
AX=0E00 BX=00FF CX=0007 DX=01FF SP=039D BP=0000
SI=005C DI=0000 DS=04BA ES=04BA SS=04BA CS=04BA
IP=011A NV UP DI NG NZ AC PE NC
04BA:011A CD21 INT 21
```


Assume you enter the following:

RF

The Debugger displays the flags:

NV UP DI NG NZ AC PE NC -

Now, enter any valid flag designation, in any order, with or without spaces. For example:

NV UP DI NG NZ AC PE NC - PLEICY

The Debugger responds only with the Debugger prompt. To see the changes, enter either the R or RF command:

RF

NV UP EI PL NZ AC PE CY -

Enter different flag values or press RETURN if you want the flags to remain this way.

4.6.15 S Command (Search)Function

This command searches the <range> specified for the specified <list>.

Format

S<range> <list>

<range>	either two <addresses> or one <address>, an L, and a <value> (<address> L <value>). A second <address> must be higher than the first and must be an offset only.
<list>	a series of <byte> values or of characters enclosed in quotes.

Comments

1. The list can contain one or more values, each separated by a space or a comma. If the list contains more than one value, the Debugger displays the starting address of each matching byte string in the range. If the list contains only one value, the Debugger displays the addresses of all the bytes in <range> containing that value.
2. If the Debugger prompt returns with no other display, the search found no match for the list in the range.

Example

Assume you enter the following:

```
SCS:100 110 41
```

The Debugger might return the response:

```
04BA:0104
04BA:010D
> _
```

4.6.16 T Command (Trace)

Function

This command executes one or more instructions, displaying the contents of all registers and flags after execution, and displaying the next instruction in decoded format.

Format

T[=] [<value>]

<address> either an alphabetic segment register designation or a 1- to 4-digit segment address, followed by a colon and an offset value. The default segment is CS.

<value> a 1- to 4-digit hexadecimal number.

Comments

1. If you enter the optional =<address>, tracing occurs at the address specified. If the address includes the segment designation, then both the CS register and the instruction pointer are specified. If the address omits the segment designation, only the instruction pointer is specified.
2. The optional <value> causes the Debugger to execute and trace the number of instructions specified by <value>.
3. The 8086 processor does not allow single-stepping for instructions that alter segment registers. Thus, a single invocation of the T command may cause two instructions, for example MOV SS, AX and MOV SP, DX, to execute. The T command then displays the contents of the registers and flags after the execution of the second instruction, not the first. And it does not display the second instruction at all, but the instruction following it.

Examples

Assume you enter the following:

T

The command causes one (or two, as in the last comment) instructions to execute. Then, it returns a display of the registers, flags, and the decoded instruction for the address now referenced by the Instruction Pointer. Assume that the address of the next instruction is 04BA:011A; then the Debugger might return the display:

AX=0E00 BX=00FF CX=0007 DX=01FF SP=039D BP=0000
SI=005C DI=0000 DS=04BA ES=04BA SS=04BA CS=04BA
IP=011A NV UP DI NG NZ AC PE NC
04BA:011A CD21 INT 21

Now, enter the following:

T=011A 10

The Debugger executes ten instructions beginning at 011A in the current segment and then displays all registers and flags for each instruction as it is executed. The display scrolls away until the last instruction has executed. Then the display stops, and you can see the register and flag values for the last few instructions performed. Remember that CONTROL + S suspends the display at any point, so that you can study the registers and flags for any instruction. CONTROL + Q resumes the display.

4.6.17 U Command (Unassemble)

Function

This command disassembles bytes and displays the source statements that correspond to them, along with addresses and hexadecimal byte values.

Formats

1. U[<address>]

<address> either an alphabetic segment register designation or a 1- to 4-digit segment address, followed by a colon and an offset value. The default segment is CS.

2. U<range>

<range> either two <addresses> or one <address>, an L, and a <value> (<address> L <value>). A second <address> must be higher than the first and must be an offset only.

Comments

1. The display of disassembled code looks like a listing for an assembled source file.

2. If you enter the U command without parameters, the Debugger disassembles 20H bytes, starting from CS:IP, to show the corresponding instructions. If you enter the U command with the <range> parameter, the Debugger disassembles all bytes in the range up to 20 bytes. If there are fewer than 20H bytes in the <range>, the Debugger displays only that many bytes.

3. If you enter the U command with the <address> parameter only, the Debugger disassembles the default number of bytes, beginning at the address specified. If you enter the U command with the <address> L <value> parameters, the Debugger disassembles all bytes beginning at the address specified for the number of lines specified by <value>. Entering the U command with the <address> L <value> parameters overrides the default limit (20H bytes).

4. If you have used the E command to alter the bytes in some addresses, the disassembler alters the corresponding source instructions. You can enter the U command to view the changed locations and use the disassembled code to edit the source file.

5. For the U command, the syntax of Assembly language statements differs slightly from the standard 8086/8088 Assembly language. Comment 4 in Section 4.6.1 lists the differences.

Examples

Assume you enter the following:

U04BA:100

The Debugger disassembles 20 bytes beginning at address 04BA:0100, as follows:

04BA:0100	BA7406	MOV	DX, 0674
04BA:0103	OE	PUSH	CS
04BA:0104	1F	POP	DS
04BA:0105	E8F802	CALL	0420
04BA:0108	EB12	JMP	013C
04BA:010A	CD21	INT	21
04BA:010C	72D5	JB	0103
04BA:010E	50	PUSH	AX
04BA:010F	B44D	MOV	AH,4D
04BA:0111	CD21	INT	21
04BA:0113	2E	CS:	
04BA:0114	A22F07	MOV	[072F],AL
04BA:0117	80FC01	CMP	AH,01
04BA:011A	7403	JZ	013F
04BA:011C	E98000	JMP	01BF
04BA:011F	58	POP	AX

Assume you enter the following:

u04ba:0100 0108

The display shows:

04BA:0100	BA7406	MOV	DX, 0674
04BA:0103	OE	PUSH	CS
04BA:0104	1F	POP	DS
04BA:0105	E8F802	CALL	0420
04BA:0108	EB12	JMP	013C

However, if you enter the following:

u04ba:100 120

the display appears exactly the same as above with the UCS:100 command. If, however, you enter the command:

UCS:100 L 20

all of the bytes in the thirty-two lines of instructions beginning at address CS:100 through address CS:120 are disassembled and displayed in both display formats.

4.6.18 W Command (Write)Function

This command writes the file being debugged to a disk file.

Format

W[<address> [<drive> <sector> <value>]]

<address>	either an alphabetic segment register designation or a 1- to 4-digit segment address, followed by a colon and an offset value. The default segment is CS.
<drive>	a 1-digit hexadecimal value.
<sector>	a 1- to 4-digit hexadecimal value used to indicate the physical sector on the disk at which writing begins.
<value>	a 1- to 4-digit hexadecimal number.

Comments

1. You must have named the file with the Debugger invocation or with the N command. Both methods format a file name properly in the file control block at DS:5C. (Refer to Section A.4.1, File Control Block.) If a subsequent N command has changed the original file name, you issue another N command to change the file name back, before writing the original file.
2. The Debugger writes the debugged file over the original file, or to a new file if the file named in the file control block at CS:5C did not previously exist. However, the Debugger does not rewrite .EXE files. If you load and modify a file before writing it to disk, the Debugger sets the date and length correctly to save the modified file.
3. If you enter no parameters, BX:CX defines the number of bytes to be written, and the starting address is CS:100. If G, R, or T command has changed these values, or if new values are needed, you must explicitly reset them before entering the W command without parameters. (You can use the R command to reset them.)
4. If you enter the W command with parameters, the write begins from the memory address you specify. W writes the file to the drive specified. The drive designation is numeric here -- 0 for Drive A, 1 for Drive B, 2 for Drive C, and so on. The Debugger writes into the file beginning at the specified <sector>, and continues until it writes the number of sectors specified in <value>.

CAUTION:

Be EXTREMELY careful when using the <sector> and <value> parameters because a mistake can write the file to incorrect locations on the disk, destroying the data at those locations.

Examples

Assume you enter the following:

W

The Debugger writes the file to disk then displays the Debugger prompt (>) to let you know that the write is complete.

Assume you enter the following:

WCS:100 1 37 2B

The Debugger writes the contents of memory, beginning with the address CS:100 to the disk in Drive B:. The data written starts at disk physical sector number 37H and consists of 2BH records. When the write is complete, the Debugger displays its prompt.

4.6.19 X Command (Change I/O Device)

Function

This command changes the device to which Debugger I/O is sent. Debugger I/O normally goes to the monitor.

Format

X <device>

<device> a reserved file name for a device

Comments

The Debugger does not use any system calls to do its diagnostic I/O. Instead, it calls the BIOS directly. Since it uses the BIOS for diagnostic I/O, the Debugger can debug MSDOS itself. Using the BIOS also allows Debugger I/O to be moved to a different device easily--a feature that can be useful for debugging graphics software.

Example

Assume you enter the following:

>X AUX

The Debugger sends all its I/O to the AUX device.

APPENDICES

Appendix A	PROCESSOR, OPERATING SYSTEM, AND FILE MANAGEEMENT INFORMATION	A-1
Appendix B	MESSAGES	B-1
Appendix C	DOS INTERRUPTS AND FUNCTION CALLS	C-1
Appendix D	BIOS INFORMATION	D-1
Appendix E	DEVICE DRIVERS	E-1
Appendix F	LOGICAL KEYBOARD PROTOCOL	F-1
Appendix G	SCREEN MANIPULATION	G-1
Appendix H	PRINTER CHARACTER SETS, CODES, AND ESCAPE SEQUENCES	H-1
Appendix I	IBM TO WANG CODE CONVERSION	I-1
Appendix J	CONTENTS OF THE WANG PC SYSTEM SOFTWARE DISKETTE	J-1
Appendix K	ABBREVIATIONS GLOSSARY	K-1

APPENDIX A

PROCESSOR, OPERATING SYSTEM, AND FILE MANAGEEMENT INFORMATION

A.1 INTRODUCTION

This appendix provides a technical overview of the Intel 8086 processor and the Microsoft Disk Operating System (DOS) as implemented on the Wang PC. The information presented here is of value to those using the Wang PC Assembler, compiled languages, Debugger, or Linker. The 8086 processor is the central processor of the Wang PC.

A.2 ARCHITECTURE OVERVIEW

This section is an overview of the 8086 processor, describing registers, memory segmentation, status and control flags, storage organization, and address generation.

A.2.1 Registers and Flags

The 8086 uses twelve 16-bit registers. Eight of them are general registers, and four are segment registers. In addition, there are a 16-bit instruction pointer (IP), six 1-bit status flags, and three control flags.

General Registers

The general registers fall into two groups: a group of four data registers and a group of four pointer and index registers. Figure A-1 gives the name, function, and group for the general registers. As Figure A-1 indicates, you can address the upper and lower halves of the data registers separately. Thus, each data register can function interchangeably as a 16-bit register, or as two 8-bit registers. The other registers always function as 16-bit units.

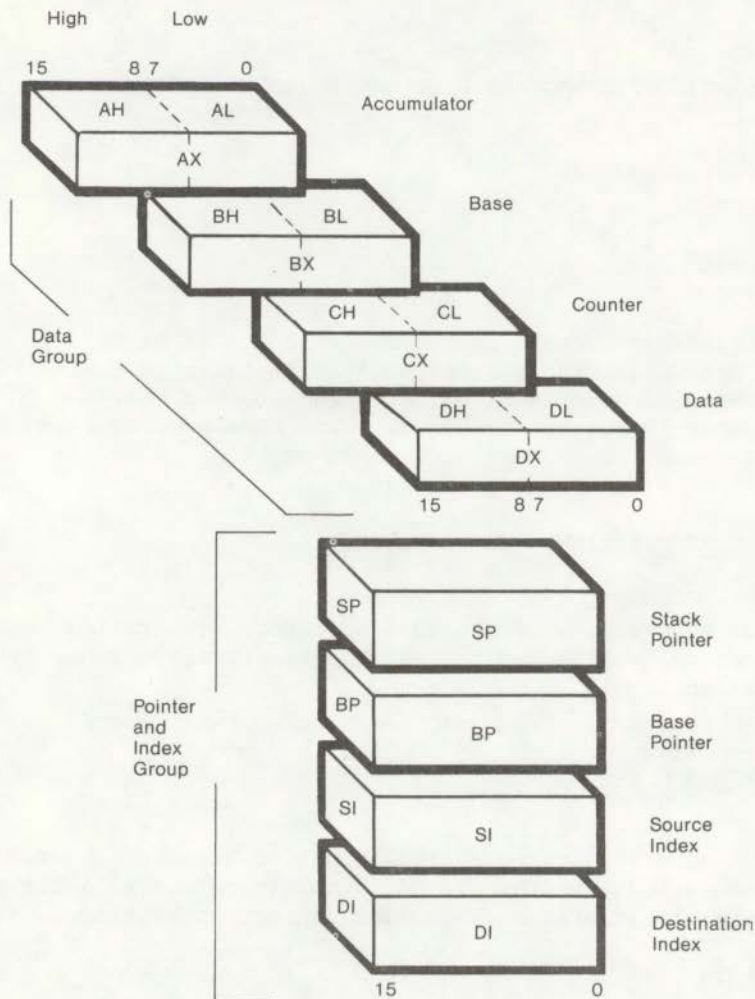


Figure A-1. General Registers

Segmentation and Segment Registers

The 8086 has a physical address space of one megabyte. Programs view the megabyte of memory space as a collection of logical "segments". A segment is a unit of memory up to 64KB long. Each segment is made up of contiguous memory locations and is an independent, separately addressable unit. The system assigns every segment a base address, which is its starting location in physical memory. All segments begin on paragraph boundaries. As Figure A-2 shows, segments may be adjacent, disjoint, partially overlapped, or fully overlapped. More than one logical segment can contain the same physical memory location.

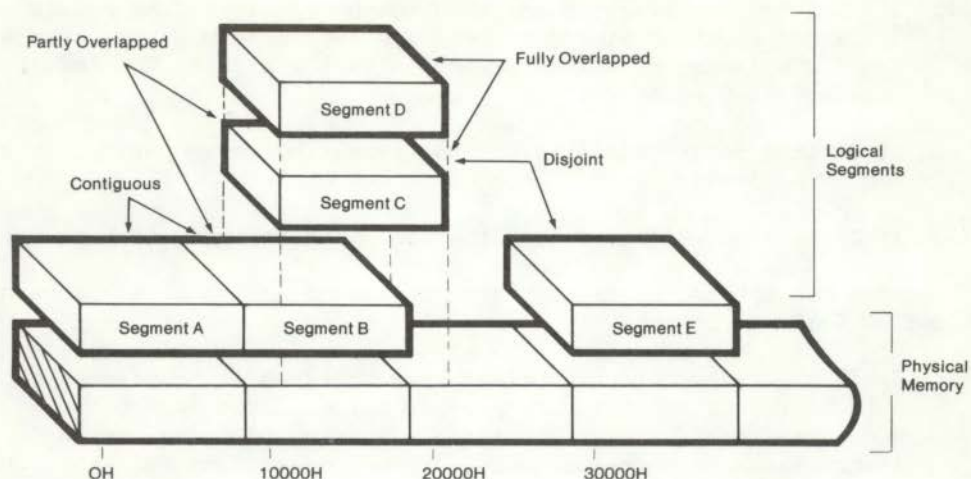


Figure A-2. The Relations of Logical Segments to Physical Memory

The system uses the four segment registers to hold the base addresses of currently addressable segments. The CS register points to the current code segment; instructions are contained in this segment. The SS register points to the current stack segment; stack operations use locations in this segment. The DS register points to the current data segment; it generally contains program variables. The ES register points to a current extra segment, which is also typically used for data storage.

Flags

The status flags reflect properties of the results of arithmetic or logical operations. Different machine instructions affect the flags differently. In general, however, the status flags reflect the following conditions:

1. If AF (the auxiliary carry flag) is on, there has been a carry out of the low four bits into the high four bits or a borrow from the high four bits into the low four bits of an 8-bit quantity (low-order byte of a 16-bit quantity). Decimal arithmetic instructions use this flag.
2. If CF (the carry flag) is on, there has been a carry out of, or a borrow into, the high-order bit of the result (8- or 16-bit). This flag is used for add and subtract instructions. Rotate instructions can also isolate a bit in memory or a register by placing it in the carry flag.
3. If OF (the overflow flag) is on, an arithmetic overflow has occurred; that is, a significant digit was lost because the size of the result exceeded the capacity of its destination location. An Interrupt On Overflow instruction is available that generates an interrupt in this situation.

4. If SF (the sign flag) is on, the high-order bit of the result is a 1. Since negative binary numbers are represented in the standard two's complement notation, SF indicates the sign of the result (0 = positive, 1 = negative).
5. If PF (the parity flag) is on, the result has even parity, an even number of 1-bits.
6. If ZF (the zero flag) is on, the result of the operation is 0.

Programs can set and clear the three control flags to alter processor operations as follows:

1. Setting DF (the direction flag) causes string instructions to auto-decrement, that is, to process strings from high to low addresses, or from right to left. Clearing DF causes string instructions to auto-increment, or to process from left to right.
2. Setting IF (the interrupt-enable flag) allows the CPU to recognize external (maskable) interrupt requests. Clearing IF disables these interrupts. IF has no affect on either nonmaskable external or internally generated interrupts.
3. Setting TF (the trap flag) puts the processor into single-step mode for debugging. In this mode, the CPU automatically generates an internal interrupt after each instruction, allowing the debugger to inspect a program as it executes, instruction by instruction.

Figure A-3 shows the usage of the control and status flags.

Control Flags	TF	Trap
	DF	Direction
	IF	Interrupt-Enable
Status Flags	OF	Overflow
	SF	Sign
	ZF	Zero
	AF	Auxiliary Carry
	PF	Parity
	CF	Carry

Figure A-3. Flags

A.2.2 Memory

This section discusses memory organization and addressing.

Storage Organization

From a storage point of view, memory spaces are identical arrays of 8-bit bytes. The system always stores word data with the most significant byte in the higher memory location. For example, in Figure A-4 the value of the word stored at address 724H is 5502H.

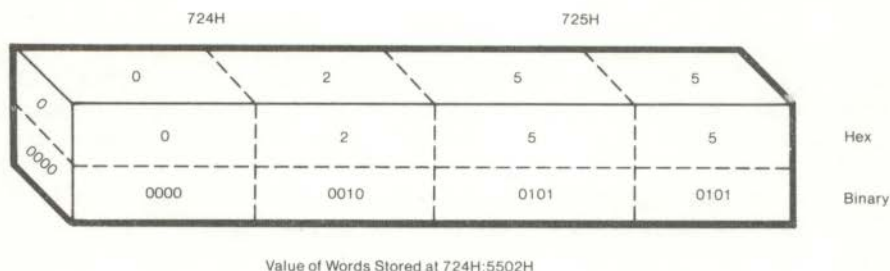
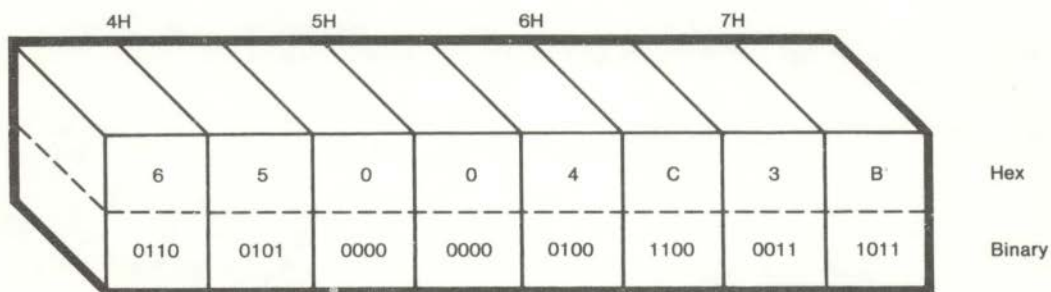


Figure A-4. Storage of Word Variables

Doubleword data items, used to hold addresses outside the segments currently pointed to by the segment registers, contain an offset value in the lower addressed word and a segment base address in the higher addressed word. Again, the higher addressed byte of each word contains the most significant byte of the word. For example, in Figure A-5 the value of the pointer stored at address 0004H is segment base address 3B4CH and offset 0065H.



Value of Pointer Stored at 4H:
Segment Base Address: 3B4CH
Offset: 0065H

Figure A-5. Storage of an Address in a Doubleword

Physical Address Generation

Every memory location can be thought of as having two kinds of addresses, physical and logical. A physical address is the 20-bit value that uniquely identifies each byte of the megabyte address space. The same physical address can have many different logical addresses.

A logical address consists of a segment base value and an offset value. For any memory location, the segment base value identifies the first byte of a segment containing the location, and the offset value is the number of bytes from the beginning of the segment to the location. The lowest addressed byte in a segment has an offset of 0.

As stored in memory or registers, base and offset values are unsigned 16-bit quantities. In order to represent physical addresses up to 1MB, however, 20 bits are required. Since segments start on 16-byte boundaries, each segment base address is a 20-bit number, with zero in the four low-order bits. In order to reference a segment base address by means of a 16-bit word, the value of the word represents the uppermost 16 bits of the 20-bit base address. That is, the segment base address is equal to the value in a segment base register shifted to the left by four binary positions. For example, if ES contains the value 6789H, then ES points to the segment starting at physical address 67890H.

A physical address is derived from a base:offset pair by adding the 16-bit offset to the contents of a segment register shifted to the left by four positions. Refer to Figure A-6. The resulting 20-bit quantity uniquely identifies a location in physical memory.

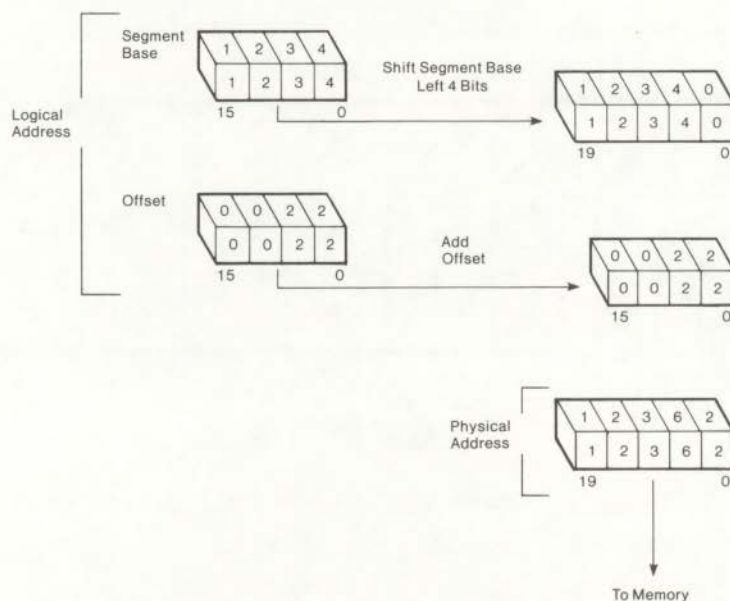


Figure A-6. Logical to Physical Address Generation

Logical addresses allow you to develop programs without prior knowledge of where the system will locate the code in memory. This facilitates dynamic management of memory resources. In order to be dynamically relocatable, a program must not load or alter its segment registers and must not transfer directly to a location outside the current code segment. In other words, all offsets in the program must be relative to fixed values contained in the segment registers.

A.3 DOS TECHNICAL INFORMATION

This section provides a technical overview of the Wang PC DOS, including the DOS Command Processor, and its use of system resources.

A.3.1 The Contents of a DOS Diskette

A system disk must contain the following items in order to load and use the DOS Command Processor:

- A start-up sector. This is the first sector on all disks formatted with the FORMAT command. The FORMAT command puts this sector on all disks in order to begin execution of the operating system.
- File Allocation Tables (FATs). Refer to Section A.4.6.
- Disk Directory. Refer to Section A.4.5.
- BIOS.SYS, the basic I/O system. This is the hardware dependent code that handles all hardware interrupts and performs all I/O for the operating system.
- MSDOS.SYS, the operating system nucleus. This is a high-level user-program interface. It contains data and file handling routines and many functions readily accessed by user programs.
- CONFIG.SYS, a system configuration file. Refer to Section A.3.2.
- COMMAND.COM, the command processor. Refer to Section A.3.3.

Figure A-7 shows the relationship between BIOS.SYS, MSDOS.SYS, COMMAND.COM and system resources.

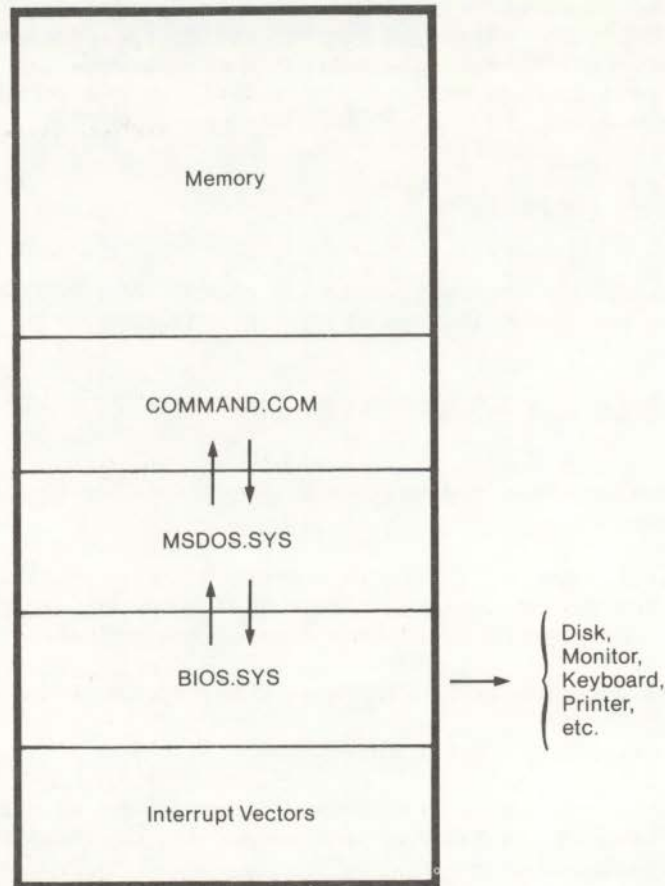


Figure A-7. DOS and Its Resources

A.3.2 DOS Initialization

When you start or restart the system, the computer reads the start-up sector, loads the start-up code into upper memory, and gives it control. The start-up code reads the disk directory to ensure BIOS.SYS and MSDOS.SYS are on the disk. If they are not on the disk, it displays an error message. If the start-up program finds these files, it loads them and gives BIOS.SYS control. MSDOS.SYS occupies the first available memory after BIOS.SYS. BIOS.SYS and MSDOS.SYS can appear anywhere on the disk.

The beginning of BIOS.SYS has a long call to its initialization code, BIOSINIT. BIOSINIT is located at the high end of BIOS.SYS. A variety of BIOS initializations are done. Then BIOS.SYS makes a long jump to the SYSINIT routine.

The SYSINIT routine initializes the DOS and reads CONFIG.SYS. SYSINIT creates a Program Segment Header (PSH) at the lowest possible segment for the initial program to which DOS gives control. Then SYSINIT gives control to that program.

The following is the list of commands for the file CONFIG.SYS. The commands can appear in any order. The list gives Wang-specified values wherever they differ from the defaults.

BUFFERS = <number>	This specifies the number of sector buffers in the system list. If CONFIG.SYS contains several BUFFERS commands, the value in the last command specifies the number of buffers.
FILES = <number>	This is the number of open files that the XENIX system calls can access. The default is 5. A number less than or equal to 5 is ignored. As shipped by Wang Laboratories, the value of FILES is set to 10.
DEVICE = <filespec>	This causes the installation of the device driver in <filespec> into the system list.
BREAK = <ON or OFF>	If ON is specified (the default is OFF), a check for CONTROL + C at the console input takes place every time control returns to DOS. As shipped by Wang Laboratories, ON is specified.
SWITCHAR = <char>	This causes function call 37H to return <char> as the current switch designator. The default is "/". As shipped by Wang Laboratories, the switch designator is "-".
AVAILDEV = <TRUE OR FALSE>	TRUE, the default, means that you can reference a device by the reserved name <dev> or by /dev/<dev>. False means that only /dev/<dev> refers to the device and <dev> by itself refers to a file of that name in the current directory.

SHELL = <filespec>

This begins execution of the shell (top-level command processor) from the last entry in <filespec>. As shipped by Wang Laboratories, the default is MENUDRV.R.COM. If you specify COMMAND.COM, the menus are not loaded and the DOS Command Processor becomes the shell.

CONFIG.SYS is an ASCII text file with each line ending with a carriage return and linefeed. A typical set of CONFIG.SYS commands would be as follows:

```
BUFFERS = 10
FILES = 10
DRIVER = network
BREAK = ON
SHELL = a:command.com a: -p
```

-p is a switch indicating that COMMAND.COM is to be permanent; otherwise, COMMAND.COM could be overwritten later.

WARNING:

If you modify CONFIG.SYS, be very careful. If you make a mistake, you may not be able to start your system. Always keep backups of the original CONFIG.SYS.

A.3.3 The DOS Command Processor

The command processor (COMMAND.COM) consists of three sections occupying different portions of memory, a resident section, an initialization section, and a transient section. (Refer to Section A.3.4 for a description of memory usage.)

The resident section is located in memory immediately after MSDOS.SYS and its data area. This section has routines to process interrupts 22H (terminate), 23H (SHIFT + CANCEL), 24H (fatal error abort) and 27H (terminate but stay resident), as well as code to reload the transient section. This section of COMMAND.COM also does all standard disk error handling, including displaying error messages and interpreting the response to the Abort, Retry, or Ignore prompt.

After the resident section, there follows an initialization section that has control during system start-up. This section determines the address of the segment where programs are loaded. The first program COMMAND.COM overlays the initialization section, which is no longer needed.

Finally, there is a transient section that is loaded in memory at the high end. The transient section is the command processor itself, with processors for the internal commands and batch files and a routine to load and run external commands. (External commands are routines with the file name extensions .COM or .EXE). This section of COMMAND.COM displays the DOS prompt, reads a command from the terminal or batch file, and executes the command. It creates a PSH for external commands immediately after the resident section of COMMAND.COM, loads the external command program into this segment, sets the terminate and SHIFT + CANCEL interrupt addresses to point to the resident section of COMMAND.COM, then passes control to the program it has loaded.

The transient section is reloaded at the termination of any program that overlaid it. If the overlaid part of COMMAND.COM is not available on disk because the disk on which it resides has been removed or if the resident section of COMMAND.COM finds an incorrect transient section, a message appears requesting you to insert a COMMAND.COM disk in the default drive and strike any key when ready. Space permitting, COMMAND.COM loads .EXE files specified for high memory immediately below the transient section of COMMAND.COM to prevent the loading process from overlaying COMMAND.COM itself.

A.3.4 DOS Memory Usage

Figure A-8 shows a typical map of memory while MSDOS.SYS is running. All addresses are physical addresses in the form NNNNNH. H indicates that the value is hexadecimal.

FFFFFH	Diagnostic and start-up PROM.
FC000H FBFFFH	Video memory (usually deselected).
E0000H DFFFFH	Maximum limit of available memory (with a memory expansion card). The transient portion of COMMAND.COM, containing the command processor, occupies the highest available part of memory. All user programs are loaded after the resident portion of COMMAND.COM.
XXXXZ0H XXXXYFH	The resident section of COMMAND.COM. This section contains interrupt handlers for interrupts 22H (terminate), 23H (CONTROL + C), 24H (fatal error), and 27H (terminate but stay resident), as well as code for loading programs and reloading the transient section.
XXXXX0H XXXXWFH	MSDOS.SYS.
XXXV0H XXXUFH	BIOS.SYS.
00400H 003FFH	Interrupt Vectors. These vectors contain segment:offset pairs indicating the address of an interrupt service routine. These vectors are defined in Table A-1. Initially, all interrupt vectors invoke an error message routine indicating that an undefined interrupt has been used.
00000H	

Figure A-8. DOS Memory Usage

Table A-1 lists the definitions of the interrupt vectors in low memory.

Table A-1. Interrupt Vectors

Address	Usage
COH - FFH	Reserved for user programs
80H - BFH	Reserved for the BIOS
40H - 7FH	Not defined
20H - 3FH	Reserved for the operating system
14H - 1FH	Reserved for the Intel 8087 emulator
05H - 13H	Reserved by Intel
04H	Dedicated - Overflow interrupt
03H	Dedicated - 1-byte INT instruction
02H	Dedicated - Nonmaskable interrupt
01H	Dedicated - Single-step interrupt
00H	Dedicated - Divide error interrupt

A.3.5 Program Initialization

When you issue an external command, COMMAND.COM locates the program at the lowest available area, immediately above COMMAND.COM's resident section. COMMAND.COM creates the PSH.

The PSH stores an INT 20 instruction at offset 0. The PSH also stores the addresses for the terminate and SHIFT + CANCEL interrupts. When finished, the program returns control to COMMAND.COM by a jump to offset 0, by issuing an INT 20, by issuing an INT 21 with the AH register set to 0, or by using function call 4CH.

Each of these methods results in control being passed to the resident section of COMMAND.COM by an INT 20 instruction. COMMAND.COM restores the terminate and SHIFT + CANCEL exit addresses from the PSH. It then passes control to the terminate address specified in the program that is finishing. If this is a program run by a batch file, control returns to the batch file. Otherwise, control returns to the transient section of COMMAND.COM, which displays the DOS prompt and awaits another command from the keyboard. When ending via INT 20 or 21, the program must set the CS register to the segment of the PSH.

Figure A-9 shows the format of the PSH. The memory size is the number of paragraphs. The word at offset 6 has the number of bytes available in the segment. Offsets are in hex.

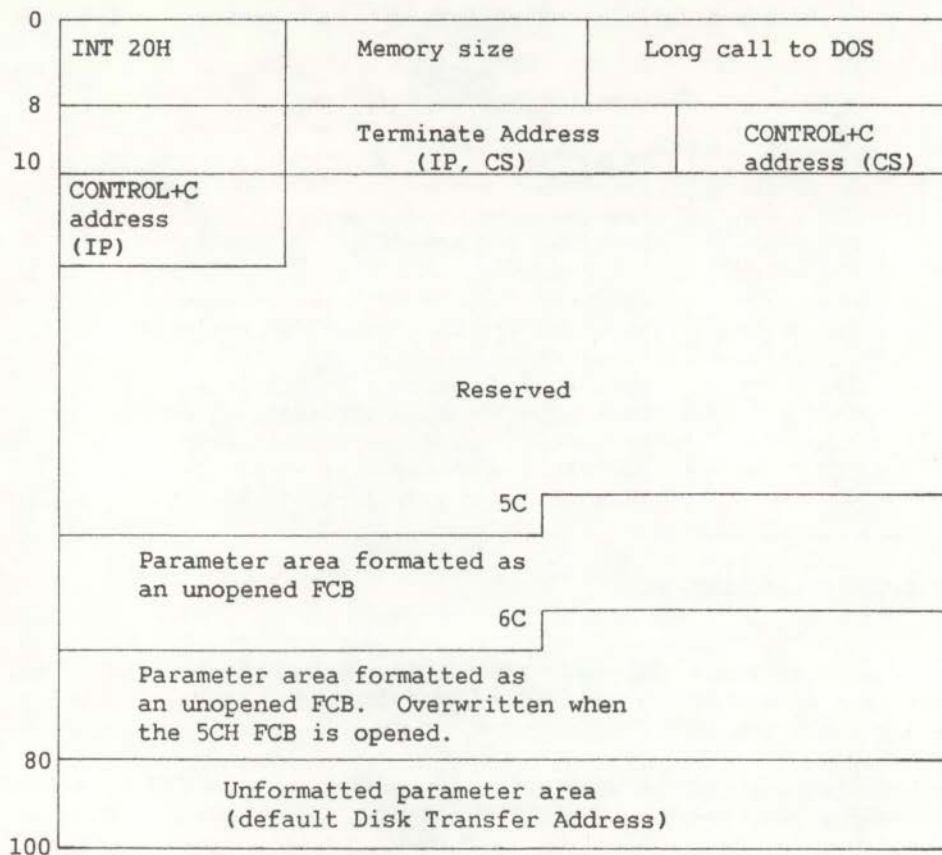


Figure A-9. The Program Segment Header

The following conditions hold when control passes to any program:

- If the drive designation of the first or second parameter was invalid, the AL or AH register contains FFH, respectively. Otherwise, they contain 00H.
- The word at offset 6 has the number of bytes available in the segment.
- The word at offset 2 has the total number of paragraphs in memory.

- The contents of the File Control Blocks (FCBs) at 5CH and 6CH are defined from the first two parameters of the command that invoked the program. (Refer to Section A.4.1.)
- The default Disk Transfer Address (DTA) is the 128 bytes beginning at 80H in the PSH. (Refer to Section A.4.2.) The byte at 80H contains the number of characters, including leading and embedded separators, following the command key word. Beginning at 81H is an unformatted parameter area containing the characters themselves.

A .COM program is a direct memory image, including space for uninitialized data. The entire program must be loaded into one segment of memory. Therefore, .COM programs must be no greater than 64KB. The following conditions hold when control passes to a .COM program:

- The four segment registers have the segment of the PSH.
- The Instruction Pointer (IP) contains 100H.
- The Stack Pointer (SP) contains the end of the program's segment, or of the transient section of COMMAND.COM, whichever is lower. To allow for a stack of 100H, the value at offset 6 is reduced by that amount.
- A word of zeros is pushed on the stack.

.EXE programs are output from the Linker with relocatable addresses. .EXE programs may be greater than 64KB. The Linker creates a 28-byte header and a relocation table of up to 200H bytes for each .EXE program. COMMAND.COM uses the information in the header to set up the registers locating the .EXE program. The following is the format of of an .EXE file header, defined as an Assembler structure.


```

EXE file      STRUC
exe_signature  DW ?      ; must contain "ZM" (5A4DH)
exe_len_mod 512 DW ?      ; number of bytes contained in
                        ; the last page (low 9 bits of
                        ; length)
exe_pages      DW ?      ; number of 512-byte pages
                        ; in the file including the
                        ; header
exe_rle_count  DW ?      ; number of entries in the
                        ; relocation table
exe_par_dir    DW ?      ; number of 16-byte paragraphs
                        ; in the header
exe_min_BSS    DW ?      ; minimum number of paragraphs
                        ; required above the end of the
                        ; loaded program (for version
exe_max_BSS    DW ?      ; 2.0 and above of the DOS)
                        ; maximum number of paragraphs
                        ; required above the end of the
                        ; program. 0FFFFH indicates the
                        ; program is located as low as
                        ; possible
exe_SS         DW ?      ; initial value to be loaded into
                        ; SS before execution.
                        ; Relocation must adjust this
                        ; value.
exe_SP         DW ?      ; initial value to be loaded
                        ; into SP before execution
exe_chksum     DW ?      ; negative of the sum of all the
                        ; words in the run file
                        ; (presently ignored)
exe_IP         DW ?      ; initial offset to be loaded
                        ; into IP before execution
exe_CS         DW ?      ; initial value to be loaded into
                        ; CS before execution.
                        ; Relocation must adjust this
                        ; value.
exe_rle_table  DW ?      ; relative byte offset of the
                        ; relocation table from the
                        ; beginning of the run file.
                        ; This should be 01CH.
exe_iov        DW ?      ; number of the overlay. The
                        ; resident part of the program
                        ; will have iov = 0.

EXE file      ENDS

```

The relocation table follows the fixed portion of the run file header. The entries in the relocation table are 4-byte pointers to words in the run file to which a relocation factor is to be added. The relocation factor is the physical address of the first byte of the resident portion divided by 16. The relocation factor must first be added to the first word of each table entry before the table entry points to the actual location of the word.

The resident portion begins at the first 512 byte boundary following the end of the relocation table. If there is no relocation, the resident portion begins immediately after the header. The layout of the entire run file, with relocation table, is as follows:

28-Byte Header	
Relocation table	512-byte boundary
Padding (less than 200H bytes)	
Memory image	512-byte boundary

The following conditions hold when control passes to an .EXE program:

- DS and ES registers have the segment of the PSH.
- CS, IP, SS, and SP registers contain values set by COMMAND.COM using information from the file header.

A.3.6 Disk Error Handling

Whenever a disk read or write error occurs or the FAT is bad, DOS transfers control to the error handler by an INT 24H (fatal error) instruction. COMMAND.COM contains the default error handler. To use your own error handler set the interrupt 24H vector to the address of your code using function call 25H. DOS gives you error information through the registers and supports Abort, Retry, or Ignore through return codes.

When it gives control to a program, DOS does not save the original contents of the fatal error address. If you change this interrupt address, your program must first save its original contents and then restore them before terminating.

A.4 FILE AND DISK MANAGEMENT

DOS supplies ways (through interrupt 21H) to create, rename, read, write, and delete files. File handling uses two areas of memory and four areas of disk space. The memory areas are the File Control Block (FCB), which can be preceded by an Extended File Control Block, and the DTA. The disk space areas are: the reserved start-up sector, the Disk Directory, the FAT, and the file section. Sections A.4.1 and A.4.2 describe memory management for files. The following subsections describe disk management.

A.4.1 File Control Block Definition

This subsection describes the DOS FCB and Extended FCB. Any reference in the description of DOS function calls (Appendix C) to an FCB, whether opened or unopened, can use either a normal FCB or an extended FCB. However, only programs written for pre-2.0 versions of MS-DOS need to use the FCBs.

The DOS File Control Block

The DOS File Control Block is formatted as follows:

byte 0	Drive indicator. 0 = the default drive, 1 = Drive A, 2 = Drive B, etc. The maximum number of drives is 64.
bytes 1-8	File name. If the file is less than 8 characters, the name must be left justified with trailing blanks.
bytes 9-11	Extension to file name. If less than 3 characters, the extension must be left justified with trailing spaces (20H). It can also be all spaces.
bytes 12-13	Current block. This word (low byte first) specifies the current block of 128 records, relative to the start of the file, in which sequential disk reads and writes occur. Zero indicates the first block of the file; one indicates the second, etc. Combined with the current record field (byte 32), these bytes identify a particular logical record.
bytes 14-15	Size of the record you want to work with. You fill this word immediately after an Open of the file, if you do not want the default logical record size (128 bytes). The Open and Create functions set this field to 128. An attempt to read or write with the field set to zero also changes it to 128.
bytes 16-19	File size. This is the current size, in bytes, of the file. User programs can read this field but must not write it.
bytes 20-21	Date of creation or last write. All creates and writes set this field to the current date. Opens set it to the date recorded in the file's directory. To change the date recorded in the file's directory, user programs can modify this field after writing to, but before closing, a file.

The format of this 16-bit field is as follows: bits 0-4, day of month; bits 5-8, month of year; bits 9-15, year minus 1980. All zeros mean no date.

bytes 22-23	Time of creation or last write. This field is updated in the same way as the Date field, bytes 20-21. The format is as follows: bits 0-4, seconds/2; bits 5-10, minutes; bits 11-15, hours.
byte 24	Flags. Bit 7 is set to 0 for a file, 1 for a device. Bit 6 is set to 0 if there are changes that have not been written. Bits 0 to 5 contain the device ID.
bytes 25-26	Disk address of the first allocation unit in the file.
bytes 27-28	Disk address of the last allocation unit in the file.
bytes 29-31	Disk addresses of the allocation unit last accessed and the directory of the file. These are 12-bit addresses occupying 1.5 bytes each.
byte 32	Next record. Identifies the record within the current block of 128 records that a sequential read or write function will access. Refer to Current Block, bytes 12-13.
bytes 32-35	Random record. You must set this field only if you are accessing a file with a random read or write function. If the record size is greater than or equal to 64 bytes, the first 3 bytes only are used as a 24-bit number representing the position of a record in the file. If the record size is less than 64 bytes, all 4 bytes are used for a 30-bit number with the same purpose. This field is thus large enough to address any byte in a file of up to 1 megabyte.

Figure A-10 shows the layout of the FCB and the extended FCB. Offsets are in decimal.

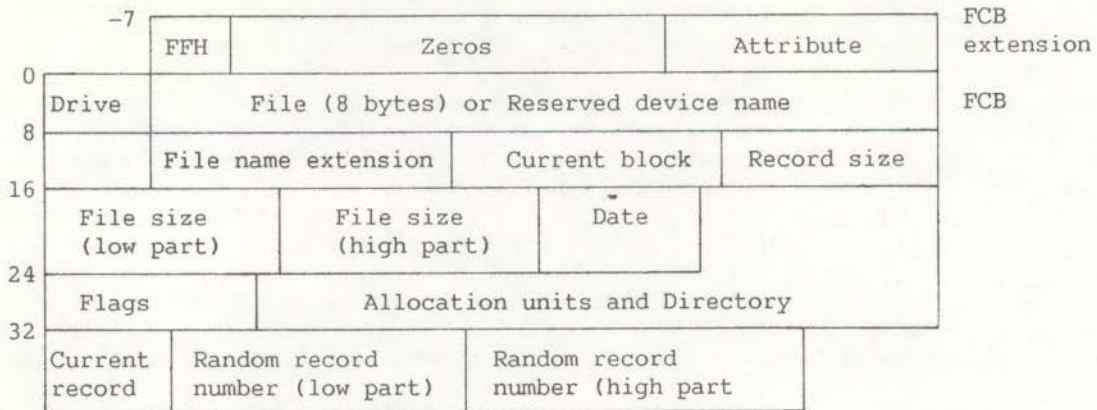


Figure A-10. File Control Block

The Extended File Control Block

The extended FCB is a special format used to search directories for files excluded from normal searches (system files and hidden files). Consisting of the 7 bytes in front of a normal FCB, the format of the extended FCB is as follows:

- FCB - 7 Flag. FF hex placed here signals an extended FCB.
- FCB - 6 to Zero field.
- FCB - 2
- FCB - 1 Attribute byte. If bit 1 = 1, directory searches include hidden files. If bit 2 = 1, directory searches include system files.

A.4.2 The Disk Transfer Address

The DTA is the memory area that contains the data for disk reads and writes. The DTA can be at any location. Your program sets the DTA by using function call 1AH.

A program can have only one DTA in effect at one time. The program must set the DTA before doing any disk I/O. Once defined, a DTA remains in effect for all disk operations until the program declares a new DTA by issuing another 1A function call. When it transfers control to a program, COMMAND.COM builds a default DTA for 128 bytes at 80H into the PSH. (Refer to Section A.3.5.)

A.4.3 Diskette Layouts

The Wang PC BIOS supports eight diskette formats, four formats with 512-byte sectors and four formats with 256-byte sectors. This section lists the specifications, together with the contents of the startup sector, for each format. Where the specifications differ for 512- and 256-byte sectors, the value for 512-byte sectors appears first, and the value for 256-byte sectors follows immediately in parenthesis.

Single-Sided, Double-Density (8 sectors per track)

512 (256) bytes per sector
 320 (640) sectors per diskette
 163840 bytes per diskette
 FAT ID Byte = 0FEH (0FAH)
 2 FATS
 1 (2) sector per FAT
 1 (2) sector per allocation unit
 64 disk directory entries
 32 bytes per disk directory entry
 Sector 0 (0-1) = Boot Sector
 Sector 1 (2-3) = File Allocation Table (Copy 1)
 Sector 2 (4-5) = File Allocation Table (Copy 2)
 Sectors 3 through 6 (6 through 13) = Disk Directory
 Sectors 7 through 319 (14 through 639) = Data Area, unreserved,
 160256 bytes

Single-Sided, Double-Density (9 sectors per track)

512 (256) bytes per sector
 360 (720) sectors per diskette
 184320 bytes per diskette
 FAT ID Byte = 0FCH (0F8H)
 2 FATS
 2 (4) sectors per FAT
 1 (2) sector per allocation unit
 64 disk directory entries
 32 bytes per disk directory entry
 Sector 0 (0-1) = Boot Sector
 Sectors 1 through 2 (2 through 5) = File Allocation Table (Copy 1)
 Sectors 3 through 4 (6 through 9) = File Allocation Table (Copy 2)
 Sectors 5 through 8 (10 through 17) = Disk Directory
 Sectors 9 through 359 (18 through 359) = Data Area, unreserved,
 179712 bytes

Double-Sided, Double-Density (8 sectors per track)

512 (256) bytes per sector
 640 (1280) sectors per diskette
 327680 bytes per diskette
 FAT ID byte = 0FFH (0FBH)
 2 FATS
 1 (2) sector per FAT
 2 (4) sectors per allocation unit
 112 disk directory entries
 32 bytes per disk directory Entry
 Sector 0 (0-1) = Boot Sector
 Sector 1 (2-3) = File Allocation Table (Copy 1)
 Sector 2 (4-5) = File Allocation Table (Copy 2)
 Sectors 3 through 9 (6 through 19) = Disk Directory
 Sectors 10 through 639 (14 through 1179) = Data Area, unreserved,
 322560 bytes

Double-Sided, Double-Density (9 sectors per track)

512 (256) bytes per sector
 720 (1140) sectors per diskette
 368640 bytes per diskette
 FAT ID byte = 0FDH (0F9H)
 2 FATS
 2 (4) sectors per FAT
 2 (4) sectors per allocation unit
 112 disk directory entries
 32 bytes per disk directory Entry
 Sector 0 (0-1) = Boot Sector
 Sectors 1 through 2 (2 through 5) = File Allocation Table (Copy 1)
 Sectors 3 through 4 (6 through 9) = File Allocation Table (Copy 2)
 Sectors 5 thru 11 (10 through 23) = Disk Directory
 Sectors 12 through 719 (24 through 1139) = Data Area, unreserved,
 362496 bytes

The Startup Sector

Figure A-11 shows the BIOS Parameter Block variables that the startup sector contains for each diskette format with 512-byte sectors. Figure A-12 gives the same information for formats with 256-byte sectors.

TYPE	DESCRIPTION	SSDD/8	SSDD/9	DSDD/8	DSDD/9
WORD	BYTES PER SECTOR	512 (200H)	512 (200H)	512 (200H)	512 (200H)
BYTE	SECTORS PER ALLOCATION UNIT	1	1	2	2
WORD	RESERVED SECTORS (NUMBER REQUIRED BY STARTUP)	1	1	1	1
BYTE	NUMBER OF FATS	2	2	2	2
WORD	NUMBER OF ROOT DIR ENTRIES	64 (40H)	64 (40H)	112 (70H)	112 (70H)
WORD	NUMBER OF SECTORS/LOGICAL IMAGE (i.e. NUMBER OF SECTORS/FLOPPY)	320 (140H) BYTES= 163840	360 (168H) BYTES= 184320	640 (280H) BYTES= 327680	720 (2D0H) BYTES= 368640
BYTE	MEDIA DESCRIPTOR (i.e. FATID)	0FEH	0FCH	0FFH	0FDH
WORD	NUMBER OF SECTORS PER FAT	1	2	1	2
WORD	SECTORS PER TRACK	8	9	8	9
WORD	NUMBER OF HEADS	1	1	2	2
WORD	NUMBER OF HIDDEN SECTORS	0	0	0	0
WORD	NUMBER OF SECTORS RESERVED FOR STARTUP + FATS + DIRECTORY	7 (07H)	9 (09H)	10 (0AH)	12 (0CH)

Figure A-11. Startup Sector BIOS Parameter Block Variables (512-byte Sectors)

TYPE	DESCRIPTION	SSDD/16	SSDD/18	DSDD/16	DSDD/18
WORD	BYTES PER SECTOR	256 (100H)	256 (100H)	256 (100H)	256 (100H)
BYTE	SECTORS PER ALLOCATION UNIT	2	2	4	4
WORD	RESERVED SECTORS (NUMBER REQUIRED BY STARTUP)	2	2	2	2
BYTE	NUMBER OF FATS	2	2	2	2
WORD	NUMBER OF ROOT DIR ENTRIES	64 (40H)	64 (40H)	112 (70H)	112 (70H)
WORD	NUMBER OF SECTORS/LOGICAL IMAGE (i.e. NUMBER OF SECTORS/FLOPPY)	640 (280H) BYTES= 163840	720 (2D0H) BYTES= 184320	1280 (500H) BYTES= 327680	1440 (5A0H) BYTES= 368640
BYTE	MEDIA DESCRIPTOR (i.e. FATID)	0FAH	0F8H	0FBH	0F9H
WORD	NUMBER OF SECTORS PER FAT	2	4	2	4
WORD	SECTORS PER TRACK	16	18	16	18
WORD	NUMBER OF HEADS	1	1	2	2
WORD	NUMBER OF HIDDEN SECTORS	0	0	0	0
WORD	NUMBER OF SECTORS RESERVED FOR STARTUP + FATS + DIRECTORY	14 (0EH)	18 (12H)	20 (14H)	24 (18H)

Figure A-12. Startup Sector BIOS Parameter Block Variables (256-byte Sectors)

A.4.4 Disk Space Allocation

DOS disk space is divided into four parts:

- **Reserved Sector.** This region contains the DOS start-up code.
- **File Allocation Table.** The FAT contains location information for the data making up each file on a given disk.
- **Disk Directory.** The directory contains information about each file on a given disk. This information includes the file's complete file name, its size, its attributes, the time and date of its last modification, and the location on disk of its first sector.
- **The File Section.** The great majority of disk space is reserved for the contents of files.

On system disks, formatted with the -S option of the FORMAT command, the first four files in the disk directory are BIOS.SYS, MSDOS.SYS, CONFIG.SYS and COMMAND.COM.

An individual file does not necessarily reside in contiguous sectors on disk, and may be scattered to decrease waste of disk space. Through the chaining of file sectors, using the FAT and the fields specified in the FCB, you can access a file either sequentially or randomly. You can access records sequentially by using the sequential read or write function calls together with the current block and record fields of the FCB. You can access records randomly by using the random read or write function calls and the random record fields. Also, the random block functions allow reading or writing large amounts of data with one function call. With any of these methods, DOS itself calculates the correct locations on the disk.

A.4.5 The Disk Directory

The FORMAT command creates a directory on each diskette as described in Section A.4.3. Directory entries, whether in the disk directory or in subordinate directories, are formatted as indicated in Table A-2 (offset values are in decimal).

Table A-2. Directory Entries

Location	Bytes	Description
0	11	File name and extension
11	1	Attributes. Table A-3 defines the entries in this field
12	10	Zero-filled field (for expansion)
22	2	Time of creation or last write Bits 0-4 = sec*2 5-10 = min 11-15 = hrs
24	2	Date of creation or last write Bits 0-4 = day 5-10 = month 9-15 = year
26	2	First allocation unit
28	4	File size, in bytes (30 bits max)

If the first byte of the file name field is E5, the directory entry is free. 00 indicates the end of the allocated directory.

Table A-3 defines the entries in the attribute byte of a directory.

Table A-3. File Attribute Codes

Byte	Attribute
1H	read only
2H	hidden file
4H	system file
8H	volume ID
10H	directory

A file can have any combination of the read only, hidden file, and system file attributes. The volume ID or directory attribute must be the only attribute of the file.

A.4.6 The File Allocation Table

DOS uses the FAT to allocate disk space to files, one allocation unit at a time. A copy of the FAT for the last disk used in each drive is retained in internal memory, space permitting. DOS writes the FAT, when the status of the disk's file section changes.

The FAT uses a 12-bit (1.5 byte) entry for each allocation unit on the disk. FAT entry number zero is used as an end-of-file trap in the DOS and is passed to the BIOS to help determine disk format. Entry 1 is reserved for future use. The first available allocation unit is assigned entry number two, and even though it is the first, is called allocation unit two.

Each FAT entry contains three hexadecimal characters. If the allocation unit is unused and available, the entry contains 000. Entries greater than FF7H are end-of-file marks. FF7H indicates that the allocation unit has a bad sector. All other hexadecimal characters are the relative number of the next allocation unit in the file. The file's directory entry contains the relative allocation unit number of the first sector in the file.

To obtain a record's absolute track and sector, use the file's first allocation unit number from the directory entry. To find the next allocation unit, multiply the FAT entry from the directory by 1.5 (for the length of each FAT entry). The whole part of the product is an offset pointing to the FAT entry mapping the next allocation unit. Now, move the word at that entry into a register by means of a MOV instruction. If the last file allocation unit number was even, keep the low-order 12 bits of the register; if it was odd, keep the high-order 12 bits. The resultant 12 bits contain the file allocation unit number of the next allocation unit in the file. If the 12 bits are all 1s (FFFH), the file has no more sectors.

APPENDIX B MESSAGES

B.1 INTRODUCTION

This appendix lists the error messages returned by the Linker, the Library Manager, and the Debugger. The list describes the cause of the error and, where appropriate, the way to correct it.

B.2 LINKER MESSAGES

All errors cause the Linker session to abort. Therefore, after you find and correct the cause, you must run the Linker again.

During the Linker session, you can receive any of the following error messages:

ATTEMPT TO ACCESS DATA OUTSIDE OF SEGMENT BOUNDS, POSSIBLY BAD OBJECT
MODULE

Probably a bad object file.

BAD NUMERIC PARAMETER

Numeric value is not in digits. Use characters from 0 through 9
only.

CANNOT OPEN TEMPORARY FILE

The Linker is unable to create the file VM.TMP because the disk is
full. Insert a new disk. Do not change the disk that receives the
list.MAP file.

ERROR: DUP RECORD TOO COMPLEX

A DUP construct in an Assembler language module is too complex. Simplify the DUP construct in the Assembler language program.

ERROR: FIXUP OFFSET EXCEEDS FIELD WIDTH

An Assembler language instruction uses a short address (an offset only) for something the Linker has positioned more than 64KB away from the base. Edit Assembler language source and reassemble.

INPUT FILE READ ERROR

Probably a bad object file.

INVALID OBJECT MODULE

Object module(s) incorrectly formed or incomplete (as when assembly was stopped in the middle).

PROGRAM SIZE OR NUMBER OF SEGMENTS EXCEEDS CAPACITY OF LINKER

The total size cannot exceed 384KB, and the number of segments cannot exceed 255.

REQUESTED STACK SIZE EXCEEDS 64KB

Specify a size less than or equal to 64KB with the /STACK switch.

SEGMENT SIZE EXCEEDS 64K

The addressing system limit is 64KB.

SYMBOL DEFINED MORE THAN ONCE

The Linker found two or more modules that define a single symbol name.

SYMBOL TABLE CAPACITY EXCEEDED

Many, very long names entered, the total list exceeding approximately 25KB.

TOO MANY EXTERNAL SYMBOLS IN ONE MODULE

The limit is 256 external symbols for each module.

TOO MANY GROUPS

The limit is 10 groups.

TOO MANY LIBRARIES SPECIFIED

The limit is 8 libraries.

TOO MANY PUBLIC SYMBOLS

The limit is 1024 public symbols.

TOO MANY SEGMENTS OR CLASSES

The limit is 256 segments and classes taken together.

UNRESOLVED EXTERNALS: <list>

The external symbols listed have no defining module among the modules or libraries files specified.

VM READ ERROR

File VM.TMP cannot be read. A disk problem; not caused by the Linker.

WARNING: NO STACK SEGMENT

A bad object module or an attempt to link a module the Linker cannot handle because it is nonrelocatable, for example, a module already linked.

WRITE ERROR IN TMP FILE

No more disk space remaining to expand the VM.TMP file.

WRITE ERROR ON RUN FILE

Usually, not enough disk space for the run file.

B.3 LIBRARY MANAGER MESSAGES

During the Library Manager session, you can receive any of the following error messages:

<symbol> is a multiply defined PUBLIC. Proceed?

A new module defines the same public symbol as a module already in the library. The message asks you to confirm the removal of the old definition of the symbol. A No response leaves the library in an undetermined state. Remove the PUBLIC declaration from one of the object modules and recompile or reassemble.

Allocate error on VM.TMP

No more space on the disk when the file is being opened.

Cannot create extract file

No place in the disk directory for the file to be created by copying a module from the library file (the Asterisk command).

Cannot create list file

No place in the disk directory for the list file.

Cannot nest response file

A response file cannot contain a response file (<@filespec>) as a parameter.

Cannot open VM.TMP

No place for VM.TMP in the disk directory.

Cannot write library file

No more space on the disk.

Close error on extract file

Not enough space on the disk for the file created by copying a module from the library file (the Asterisk command).

Error: An internal error has occurred

Contact the Wang Professional Computer Assistance Center. Refer to The Wang Professional Computer Introductory Guide for the toll free number.

Fatal Error: Cannot open input file

The object file name is improperly formed, or it cannot be found in the disk directory.

Fatal Error: Module is not in the library

You are trying to delete a module that is not in the library.

Input file read error

A bad object module or faulty disk.

Invalid object module/library

A bad object module and/or library.

Library Disk is full

No more space on the disk.

Listing file write error

No more space on the disk.

No library file specified

No response to Library File prompt.

Read error on VM.TMP

Disk not ready for read. The disk drive door is open or the disk is not properly placed in the drive.

Symbol table capacity exceeded

Too many characters (about 30KB) in public symbols.

Too many object modules

More than the maximum of 500 object modules.

Too many public symbols

More than the maximum of 1024 public symbols.

Write error on library/extract file.

Not enough space on the disk for a library file or a file extracted from a library.

Write error on VM.TMP

No more space on the disk.

B.4 DEBUGGER MESSAGES

During the DEBUG session, you can receive any of the following error codes. Each error terminates the DEBUG command with which it is associated, but does not terminate DEBUG itself.

ERROR CODE	DEFINITION
BF	Bad Flag On an attempt to alter a flag, the characters you entered were not one of the acceptable pairs of flag values. Refer to the R command, Section 4.6, for the list of acceptable flag entries.
BP	Too many Breakpoints More than 10 breakpoints specified as parameters to the G command. Reenter the G command with 10 or fewer breakpoints.
BR	Bad Register An R command entered with an invalid register name. Refer to the R command, Section 4.6, for the list of valid register names.
DF	Double Flag Two values entered for one flag. You can specify a flag value only once for each RF command.

APPENDIX C

DOS INTERRUPTS AND FUNCTION CALLS

C.1 INTERRUPTS

DOS reserves interrupt types 20H to 3FH for its use. This means absolute memory locations 80H to FFH are reserved by DOS.

To write your own interrupt processing routine, use function call 25H (set interrupt vector). Return from the interrupt routine via the IRET instruction, which restores all registers and flags. On return from the interrupt, be sure to reset the interrupt vector to the original value so that it is available for other applications.

In the following, the defined interrupts are first listed alphabetically. Then they are listed numerically with explanations. All values are in hex.

- | | |
|----|---|
| 25 | Absolute disk read. |
| 26 | Absolute disk write. |
| 24 | Fatal error abort vector. |
| 21 | Function request. |
| 20 | Program terminate. |
| 23 | SHIFT + CANCEL exit address. |
| 27 | Terminate but stay resident. |
| 20 | Program terminate. This is one way to exit a program. This vector transfers control to the logic in DOS that restores the SHIFT + CANCEL exit addresses to the values they had on entry to the program. All file buffers are flushed to disk. Be sure that all files that have changed in length are closed (see function call 10H) prior to issuing this interrupt. If the changed file is not closed prior to this interrupt, the directory does not contain its correct length. When this interrupt executes, CS <u>must</u> point to the 100H parameter area. |
| 21 | Function request. See Section C.2, Function Calls. |

- 22 Terminate address. The address represented by this interrupt (88-8BH) is the address to which control transfers when a child program terminates. If a program wishes to execute a child program and have control return to the parent when the child terminates, the parent must set the terminate address prior to creation of the segment it will load the child program into. Otherwise, once the second program executes, its termination would also terminate the parent. This interrupt does not specify the termination address of the program that calls it, only the termination address of tasks invoked by that program.
- 23 SHIFT + CANCEL exit address. If you press SHIFT + CANCEL during keyboard input or video output, UC appears on the console and an interrupt type 23H executes. If function 9H or 0AH (buffered output and input) is executing, then I/O continues from the start of the line. When the interrupt occurs, all registers contain the value they had when the original call to DOS occurred. If you do not want normal SHIFT + CANCEL processing, which terminates the program, the program can point to a special routine. If this SHIFT + CANCEL routine preserves all registers, it may end with a return-from-interrupt instruction (IRET) to continue program execution. There are no restrictions on what the SHIFT + CANCEL handler is allowed to do, including DOS function calls, so long as the registers do not change if IRET is used.
- If the program creates a new segment and then loads in a child program that itself changes the SHIFT + CANCEL address, the termination of the child program and return to the parent restores the SHIFT + CANCEL address to the value it had before execution of the second program.
- 24 Fatal error abort vector. When a fatal error occurs within DOS, control transfers with an INT 24H. On entry to the error handler, BP:SI points to the device header of the device causing the error. (Refer to Appendix E for the format of the device header.)

AH has its bit 7=0 if the error is a hard disk error (probably the most common occurrence). Bit 7=1 for character device errors or for a bad image of the FAT. If the error is a hard disk error, bits 0-2 include the following:

bit 0		0 if read, 1 if write
<u>bit 2</u>	<u>1</u>	<u>AFFECTED DISK AREA</u>
0	0	reserved area
0	1	file allocation table
1	0	directory
1	1	data area

AL, CX, DX, and DS:BX are set for a retry of the transfer with INT 25H or INT 26H (below). The hardware sets DI to a 16-bit error code.

The error codes are:

0	write protect
1	unknown device
2	drive not ready
3	unknown command
4	data error
5	bad drive request structure length
6	seek error
7	unknown media
8	sector not found
9	printer out of paper
A	write fault
B	read fault
C	general disk failure

The registers are set for a BIOS disk call and the error code is in the lower half of the DI register with the upper half undefined. The only function calls the error handler can make are calls 1 to 12. Making any other calls destroys the DOS stack.

The user stack looks as follows starting from the top:

```

SP
CS
FLAGS
AX
BX
CX
DX
SI
DI
BP
DS
ES
IP
CS
FLAGS

```

If an IRET is executed, DOS responds according to AL as follows:

(AL)=0	ignore the error
=1	retry the operation (IF YOU USE THIS OPTION, STACK DS, ES, BX, CX, AND DX MUST NOT BE MODIFIED.)
=2	abort the program

25 Absolute disk read. This transfers control directly to the DOS BIOS. For this entry point, records and sectors are the same size. The request is as follows:

(AL)	Drive number (0 = A, 1 = B, and so on.)
(CX)	Number of sectors to read
(DX)	Beginning logical record number
(DS:BX)	Transfer address

The routine transfers the number of records specified between the given drive and the transfer address. Logical record numbers are obtained by numbering each sector sequentially starting from zero and continuing across track boundaries. For example, on diskettes formatted with nine sectors per track, logical record number 0 is track 0, sector 1, whereas logical record number 12H is track 2, sector 0.

NOTE:

Upon return, the original flags are still on the stack (the INT instruction puts them there). This is necessary because the flags contain return information. Be sure to pop the stack to prevent uncontrolled growth.

This call destroys all registers but the segment registers. If the transfer is successful, the carry flag (CF) is zero. If the transfer is not successful CF=1 and (AL) indicates the error as follows:

<u>Return</u>	<u>Description</u>
0	attempt to write on write protected disk
2	disk not ready
4	data error
6	seek error
8	sector not found
A	general disk failure
C	write fault

- 26 Absolute disk write. This vector is the counterpart to interrupt 25 above. Except for the fact that this is a write, the description above applies.

- 27 Terminate but stay resident. Programs that are to remain resident when COMMAND regains control use this vector. COMMAND loads such a program as an executing .COM file. After the program is initialized, it must set DX to its last address plus one in the segment it is executing in, then execute an interrupt 27H. COMMAND then treats the program as an extension of DOS and does not overlay the program when other programs are executed.

C.2 FUNCTION CALLS

You call a function by placing a function number in the AH register, supplying additional information in other registers as necessary for the specific function, then executing an interrupt type 21H. When DOS takes control, it switches to an internal stack. User registers, except AX, are preserved, except as indicated in the specific calls. The user stack must be sufficient to accommodate the interrupt system. We recommended that the user stack be 80H in addition to your program's needs.

Versions of MS-DOS prior to version 2.0 implement function calls 0 to 2E. These calls are compatible with versions 2.0 and above, except in the following cases.

Programs that use functions 27 or 28 to get a pointer to the FAT will no longer work. Because the FAT is now cached with other disk resources, there is no fixed location in memory for the FAT. Functions 27 and 28 still exist, but you can only use them to get the FAT ID byte. On return from 27 or 28, ES:BX points to a FAT ID byte for the drive you are using.

NOTE:

Doing anything with 27 and 28 other than reading this one byte will probably cause the system to crash.

Versions 2.0 and above of MS-DOS implement the remaining function calls. Some of the 2.0 calls duplicate the functions of pre-2.0 calls but add additional functionality. The additional functionality allows for future upgrades of the operating system. Therefore, it is advisable to use 2.0 calls whenever you have a choice.

I/O control with versions 2.0 and above makes use of a 16-bit file handle assigned to the file or device by the system when the file or device is opened. Subsequent I/O functions on the file or device must refer to the file handle. Function calls 3C, 3D, and 45 return the file handle in AX. The following is a list of the default file handles:

- 0 = standard input device
- 1 = standard output device
- 2 = standard error message device
- 3 = standard auxiliary device
- 4 = standard printer

Previous to version 2.0, writes to the screen would use function call 6 (Direct console I/O). Under 2.0 and above, writes to the screen should use function call 40 (Write a file) with the default output file handle (1).

Another important difference between pre-2.0 and 2.0 I/O function calls is that 2.0 file I/O makes use of directory paths where pre-2.0 calls use file names and extensions only. Refer to The Wang Professional Computer Introductory Guide for a discussion of directory paths.

Several of the descriptions that follow refer to ASCIZ character strings. An ASCIZ string is a series of ASCII characters terminated by 0H as the end-of-string indicator.

The function calls from 39 on are compatible with the XENIX operating system. XENIX is a multitasking operating system, and use of these calls allows you to upgrade to that system.

For 2.0 calls, the AX register receives error codes and return values. The error code definitions are as follows:

- 0 = no error occurred
- 1 = error invalid function
- 2 = error file not found
- 3 = error path not found
- 4 = error too many open files
- 5 = error access denied
- 6 = error invalid handle
- 7 = error memory block bad
- 8 = error not enough memory
- 9 = error invalid block
- A = error bad environment (refer to function call 4B)
- B = error bad format
- C = error invalid access
- D = error invalid data
- F = error invalid drive
- 10 = error current directory
- 11 = error not same device
- 12 = error no more files

NOTE:

When the value in AX could be either an error code or return value, AX has an error code, if the Carry Flag is on. Otherwise, it has the return value.

The following is an alphabetical list of the function calls. Where pre-2.0 and 2.0 calls have duplicate functions, the list indicates which category the call belongs to.

48	Allocate memory.
3	Auxiliary input.
4	Auxiliary output.
A	Buffered keyboard input.
3B	Change the current directory.
37	Change incompatible configuration parameters.
43	Change write protection.
C	Character input with buffer flush.
B	Check keyboard status.
3E	Close a file. (2.0)
10	Close file. (1.25)
8	Console input without echo.
3C	Create a file. (2.0)
16	Create file. (1.25)
26	Create a new program segment.
47	Current Directory.
19	Current disk.
41	Delete a directory entry.
13	Delete file.
7	Direct console input.
6	Direct console I/O.
D	Disk reset.
45	Duplicate a file handle.
4B	Execute a program.
4C	Exit from a process.
23	File size.
57	File times.
4E	Find first.
4F	Find next.
46	Force a duplicate of a handle.
49	Free allocated memory.
2F	Get current DTA.
2A	Get date.
36	Get disk free space.
30	Get DOS version number.
34	Get DOS critical section flag.
35	Get interrupt vector.
2C	Get time.
54	Get Verify Flag.
38	International information.
44	I/O control for devices.
31	Keep Process.
1	Keyboard input.
39	Make Directory.
4A	Modify allocated block.
42	Move file read/write pointer.
3D	Open a file. (2.0)
F	Open file. (1.25)
29	Parse file name.
9	Print string.
5	Printer output.
27	Random block read.
28	Random block write.
21	Random read. (1.25)

22	Random write. (1.25)
3F	Read a file or device. (2.0)
3A	Remove Directory.
56	Rename. (2.0)
17	Rename file. (1.25)
4D	Retrieve the return code of a child process.
2	Screen output.
11	Search for the first entry.
12	Search for the next entry.
E	Select disk.
14	Sequential read. (1.25)
15	Sequential write. (1.25)
2B	Set date.
1A	Set disk transfer address.
33	Set or Get CONTROL + C Trapping.
24	Set random record field.
2D	Set time.
25	Set interrupt vector.
2E	Set/Reset Verify Flag.
40	Write a file. (2.0)

Sections C.2.1 and C.2.2 list the function calls numerically and give an explanation of each. Section C.2.1 lists the pre-2.0 calls, and section C.2.2 lists the 2.0 calls. All values are in hex.

C.2.1 Pre-2.0 Function Calls

0	Program terminate. The terminate and SHIFT + CANCEL exit addresses contain the values they had on entry to the terminating program. The routine flushes all file buffers but does not set directories correctly for unclosed files that have changed in length. Control transfers to the terminate address.
1	Keyboard input. The routine waits for you to type a character at the keyboard, then displays the character on the screen and returns it in AL. The character is checked to see if it is a CONTROL + C (SHIFT + CANCEL). If the character is a CONTROL + C, the routine executes interrupt 23.
2	Screen output. The routine sends the character in DL to the screen. If a SHIFT + CANCEL is detected after the output, the routine executes interrupt 23.
3	Auxiliary input. The routine waits for a character from the auxiliary input device, then returns that character in AL.

- 4 Auxiliary output. The routine sends the character in DL to the auxiliary device.
- 5 Printer output. The routine sends the character in DL to the printer.
- 6 Direct console I/O. If DL contains FF, AL returns with the keyboard input character, if one is ready; otherwise, it returns with 00. If DL is not FF, then the routine assumes DL has a valid character that it sends to the screen.
- 7 Direct console input. The routine waits for you to type a character at the keyboard, then returns the character in AL. It is similar to function 6, but no checks for SHIFT + CANCEL are made on the character.
- 8 Console input without echo. This function is similar to function 1, except that the key does not appear on the screen.
- 9 Print string. On entry, DS:DX must point to a character string in memory terminated by a \$ (24). Each character in the string appears on the screen in the same form as with function 2.
- A Buffered keyboard input. On entry, DS:DX point to an input buffer. The first byte specifies the number of characters the buffer can hold; the number must not be zero. The routine reads characters from the keyboard and places them in the buffer beginning at the third byte. The computer continues to read the keyboard and fill the buffer until you press RETURN. If the buffer fills to one less than the maximum, additional keyboard input is ignored until you press RETURN. The routine sets the second byte of the buffer to the number of characters received, excluding the carriage return (0D), which is always the last character.
- B Check keyboard status. If a character is available from the keyboard, AL is FF; otherwise, AL is 00.
- C Character input with buffer flush. The routine first empties the keyboard type-ahead buffer. Then, if AL is 1, 6, 7, 8, or 0A, the corresponding DOS input function executes. If AL is not one of these values, no further operation occurs, and AL returns 00.
- D Disk reset. The routine flushes all file buffers. Files that have changed in size will not be properly recorded in the directory until they are closed. You do not need to call this function before a disk change if all files that you have written to are already closed.
- E Select disk. The drive you specified in DL (0 = A, 1 = B, and so on) becomes the default drive. AL returns the number of drives.

- F Open file. On entry, DS:DX point to an unopened file control block (FCB). The routine searches the directories for the named file and sets AL to FF if it does not find the file. If the routine does find the file, it sets AL to 00 and fills the FCB as follows:

If the drive code is 0 (default disk), the routine changes it to the actual disk used (A=1, B=2, and so on). This allows changing the default disk without interfering with subsequent operations on this file. The routine sets the high byte of the current block field. FCB bytes 0E-0F (the size of the record to be worked with) contain the system default of 80. The routine sets the size of the file, the time, and the date in the FCB, from information obtained from the directory.

You must set the record size (FCB bytes 0E-0F) to the size you wish, if the default 80 is not appropriate. Also, you must set the random record field and/or current block and record fields.

- 10 Close file. This function is necessary after file writes to ensure all directory information is current. On entry, DS:DX point to an opened FCB. The routine searches the directories and, if it finds the file, compares its position with that kept in the FCB. If the routine does not find the file, it assumes you have changed the disk; and AL returns FF. Otherwise, it updates the directory to reflect the file status as recorded in the FCB, and AL returns 00.

- 11 Search for the first entry. On entry, DS:DX point to an unopened FCB. The routine searches the directory for the first matching name (the name could have ?s indicating any letter matched). If it does not find a match, AL returns FF. Otherwise, it sets locations at the disk transfer address as follows:

1. If the FCB you specified for searching is an extended FCB, the routine sets the first byte to FF, then 5 bytes to zeros, then one byte to the attribute byte from the search FCB, then one byte to the drive number used (A=1, B=2, and so on.), then 32 bytes to the the directory entry. Thus, the disk transfer address contains a valid, unopened, extended FCB with the same search attributes as the search FCB.
2. If the FCB you specified for searching is a normal FCB, the routine sets the first byte to the drive number used (A=1, B=2, and so on.) and the next 32 bytes contain the matching directory entry. Thus, the disk transfer address contains a valid, unopened, normal FCB.

Refer to Subsection A.4.5 for the format of directory entries.

- 12 Search for the next entry. After function 11 finds a match, a call to function 12 finds the next match to an ambiguous request (?s in the search file name). Both inputs and outputs are the same as for function 11. The reserved area of the FCB contains information necessary for continuing the search, so you must not modify the reserved area.

- 13 Delete file. On entry, DS:DX point to an unopened FCB. This function deletes all matching directory entries. If no directory entries match, AL returns FF; otherwise, AL returns 00.
- 14 Sequential read. On entry, DS:DX point to an opened FCB. The routine loads the record addressed by the current block (FCB bytes C-D) and the current record (FCB byte 1F) at the disk transfer address, then increments the record address. Encountering an end-of-file causes AL to return either 01 or 03. A return of 01 indicates there was no data in the record, 03 indicates a partial record read filled out with zeros. A return of 02 means not enough room was available in the disk transfer segment to read one record, so the transfer aborted. AL returns 00 if the transfer completes successfully.
- 15 Sequential write. On entry, DS:DX point to an opened FCB. The routine writes the record addressed by the current block and current record fields from the disk transfer address (or in the case of records less than sector sizes, buffers the records for an eventual write when a sector's worth of data has accumulated). The record address is then incremented. If the disk is full, AL returns with a 01. A return of 02 means there was not enough room in the disk transfer segment to write one record, so the transfer aborted. AL returns 00 if the transfer completed successfully.
- 16 Create file. On entry, DS:DX point to an unopened FCB. The routine searches the directory for an empty entry. If it finds none, AL returns FF. Otherwise, the routine initializes the entry to a zero-length file and opens the file (see function F). AL returns 00.
- 17 Rename file. On entry, DS:DX point to a modified FCB that has a drive code and file name in the usual position, and a second file name starting 6 bytes after the first (DS:DX+11) in what is normally a reserved area. The routine changes every character in the first file name to the character in the corresponding position in the second (with the restriction that two files cannot have the exact same name and extension). If ?s appear in the second name, the corresponding positions in the original name will be unchanged. AL returns FF if no match was found, otherwise 00.
- 19 Current disk. AL returns with the code of the default drive (0 = A, 1 = B, and so on.)
- 1A Set disk transfer address. This call sets the disk transfer address to DS:DX. DOS will not allow disk transfers to wrap around within the segment, nor to overflow into the next segment.

- 21 Random read. On entry, DS:DX point to an opened FCB. The routine sets the current block and current record to agree with the random record field, then it loads the record addressed by these fields at the current disk transfer address. An end-of-file causes AL to return either 01 or 03. If AL returns 01, no more data is available. If it returns 03, a partial record is available, filled out with zeros. A return of 02 means there was not enough room in the disk transfer segment to read one record, so the transfer aborted. AL returns 00 if the transfer completed successfully.
- 22 Random write. On entry, DS:DX point to an opened FCB. The routine sets the current block and current record to agree with the random record field, then writes the record addressed by these fields (or buffered, in the case of records not the same as sector sizes) from the disk transfer address. If the disk is full, AL returns 01. A return of 02 means there was not enough room in the disk transfer segment to write one record, so the transfer aborted. AL returns 00 if the transfer completed successfully.
- 23 File size. On entry, DS:DX point to an unopened FCB. The routine searches the directory for the first matching entry, and if it finds none, AL returns FF. Otherwise, it sets the random record field with the size of the file (in terms of the record size field rounded up), and AL returns 00.
- 24 Set random record field. On entry, DS:DX point to an opened FCB. This function sets the random record field to the same file address as the current block and record fields.
- 25 Set interrupt vector. This function sets the interrupt type specified in AL to the 4-byte address DS:DX.
- 26 Create a new program segment. On entry, DX has a segment number at which to set up a new program segment. The routine copies the entire 100 byte area at location zero in the current program segment into location zero in the new program segment. Then, the routine updates the memory size information at location 6 and saves the current termination and SHIFT + CANCEL exit addresses in the new program segment, starting at 0A.
- 27 Random block read. On entry, DS:DX point to an opened FCB, and CX contains a record count that must not be zero. The routine reads into the disk transfer address the specified number of records (in terms of the record size field) from the file address specified in the random record field. If end-of-file appears before all records are read, AL returns either 01 or 03. A return of 01 indicates end-of-file, and the last record read is a complete record. 03 indicates the last record read is a partial record. If wraparound above address FFFF in the disk transfer segment would occur, the routine reads as many records as possible, and AL returns 02. If AL returns 00, the attempt to read all records was successful. In any

case, CX returns with the actual number of records read, and the random record field and the current block/record fields receive the address of the next record.

- 28 Random block write. This function is similar to function 27, except that this function is a write and uses a write-protect indication. If there is insufficient space on the disk, AL returns 01, and no write takes place. If CX is zero upon entry, no write takes place, but the routine sets the file to the length specified by the Random Record field (whether longer or shorter than the current file size), allocating or releasing allocation units, as needed.
- 29 Parse file name. On entry, DS:SI points to a command line to parse; ES:DI points to a portion of memory to receive an unopened FCB. The scan ignores leading TABs and spaces. If bit 0 of AL is equal to 1 on entry, then the scan ignores at most one leading file name separator along with any trailing TABs and spaces. The four file name separators are:

; , = +

If bit 0 of AL is equal to 0, then all parsing stops on encountering a separator. The routine scans the command line for the file name of the form d:filename.ext. If it finds one, it sets up a corresponding unopened FCB at ES:DI. The value of AL bits 1, 2, and 3 on entry determine what to do if the drive, file name, or extension, respectively, are missing. In each case, if the bit is a zero and the file is not present on the command line, the FCB receives a fixed value (0, meaning the default drive for the drive field; all blanks for the file name and extension fields). If the bit is a 1, and the field is not present on the command line, then that field in the destination FCB at ES:DI does not change. If an asterisk (*) appears in the file name or extension, then the routine sets all remaining characters in the name or extension to ?.

The following characters are illegal within DOS file specifications:

" / [] + = ; ,

Also, control characters and spaces may not appear as elements of file specifications. If the scan encounters any of these characters or finds the period (.) or colon (:) at an invalid position, parsing stops at that point.

If either ? or * appears in the file name or extension, AL returns 01; otherwise, it returns 00. DS:SI returns pointing to the first character after the file name.

- 2A Get date. Returns the date in CX:DX. CX has the year, DH has the month (1=Jan, 2=Feb, and so on.), and DL has the day. If the time-of-day clock rolls over to the next day, the date changes accordingly, taking into account the number of days in each month and leap years.

- 2B Set date. On entry CX:DX must have a valid date in the same format as returned by function 2A. If the date is valid and the set operation is successful, AL returns 00. If the date is not valid, then AL returns FF.

- 2C Get time. Returns with the time of day in CX:DX. Time is represented as four 8-bit binary quantities, as follows: CH has the hours (0-23), CL has minutes (0-59), DH has seconds (0-59), DL has 1/100 seconds (0-99). This format easily converts to a printable form, yet you can also calculate with it (e.g., subtracting two times).

- 2D Set time. On entry, CX:DX has the time in the same format as returned by function 2C. If any component of the time is not valid, the set operation aborts, and AL returns FF. If the time is valid, AL returns 00.

C.2.2 2.0 Function Calls

- 2E Set/Reset Verify Flag. On entry, DL must be 0 and AL has the verify flag: 0 = no verify, 1 = verify after write. The I/O system checks this flag on each write and interprets its meaning as write only or write with verify.

- 2F Get current DTA. The Disk Transfer Address is returned in ES:BX.

- 30 Get DOS version number. On return, AL-AH has the two part version designation. Thus, for MS-DOS 1.28, AL would be 1 and AH would be 28. For versions previous to 1.28, AL would be 0. BH has a code indicating the manufacturer. For Wang Laboratories, the code is 1.

- 31 Keep Process. This routine allows you to terminate a process while retaining it in memory. On entry, you pass an exit code in AL. DX has the size, in paragraphs, of the block allocated for retaining the process. The size may be less than or equal to the size of the initial allocation block of the process. The routine does not free other allocation blocks belonging to the process. On exit, AX has the exit code you supplied. You may retrieve the exit code by using the Wait function call (4D).

- 33 Set or get CONTROL + C trapping. This routine allows you to specify Control + C trapping for all function calls (1 through 12 include it automatically). It also allows you determine whether CONTROL + C trapping has been set for all calls. On entry, DL has 0, for CONTROL + C trapping off, or 1, for CONTROL + C trapping on. AL has a subfunction code: 0 = get CONTROL + C trapping, 1 = set CONTROL + C trapping. If AL has 0 on entry, on return, DL has 0 for CONTROL + C trapping off or 1 for CONTROL + C trapping on.
- 34 Get DOS critical section flag. In an interrupt service routine, this routine indicates whether or not the DOS was interrupted in a critical section. On return, ES:BX is the address of a byte memory cell inside the DOS. If the byte is zero, then the DOS is not in a critical section, and the interrupt routine can call it. If the byte is non-zero, you should consider the DOS to be in an uninterruptable state, and for reliability, you should issue no DOS calls.
- 35 Get interrupt vector. On entry, AL contains the number of an interrupt. This routine returns the address of the interrupt routine in ES:BX.
- 36 Get disk free space. On entry, DL contains the disk drive: 0 = the default, 1 = A, 2 = B, 3 = C. On return, BX has the number of free allocation units on the disk. DX has the total number of allocation units. CX has the number of bytes per sector. And AX has the number of sectors per allocation unit. If you give an invalid drive number, AX is set to -1.
- 37 Change incompatible configuration parameters. The routine passes a subfunction code in AL as follows:
- 0 DL, on return, contains the DOS switch character. The default is '-'.
 - 1 The switch character is set to a character passed in DL.
 - 2 DL is set to the device availability byte. If this byte is 0, then devices must be accessed in file I/O calls by coding /dev/<device>. If this byte is non-zero, then the devices are available at every level of the directory tree (e.g., CON is the console device not the file CON).
 - 3 The device availability byte is set to the value passed in DL.

If the subfunction code is out of range, AL contains FF on return.

- 38 International information. On entry, AL contains a geographic code: 0 = USA; 1 = Europe; 3 = Japan. In the block of memory addressed by DS:DX, this routine returns the following country dependent information:

First word: date/time format
 Third byte: ASCIZ currency symbol
 Fourth byte: ASCIZ thousands separator
 Fifth byte: ASCIZ decimal separator

The values in the date/time format word have the following meanings:

0 - USA standard h:m:s m/d/y
 1 - European standard h:m:s d/m/y
 2 - Japanese standard y/m/d h:m:s

- 39 Make Directory. This routine creates a new entry in a directory. On entry, DS:DX points to the ASCIZ name of a directory path. The routine creates a directory entry for the last name on the path. On return, AX is 3 if the path was not found, 5 if access was denied.
- 3A Remove Directory. On entry, DS:DX points to the ASCIZ name of a directory path whose last entry is a subdirectory. This routine removes the final entry in the path from its parent directory. You cannot remove the current default directory or any subdirectory that contains files. On return, AX is 3 if the path was not found, 5 if access was denied, and 10 if the final entry is the current default directory.
- 3B Change the current directory. On entry, DS:DX point to the ASCIZ name of a directory path. That path becomes the current path. If any member of the specified pathname does not exist, the current directory does not change. On return, AX is 3 if the path was not found, 5 if access was denied.
- 3C Create a file. On entry, DS:DX points to the name of a file. If the file does not exist, this routine creates the file in the proper directory and gives the file the read/write access protection code, 0 passed in CX. If the file exists, the routine truncates it to zero length to prepare it for writing. On normal return, AX has the handle of the file. Otherwise, AX has 3 if the path was not found, 4 if there were too many open files, or 5 if access was denied.
- 3D Open a file. On entry, DS:DX points to the ASCIZ name of the file to be opened. AL contains an access protection code. The code values and their meanings are as follows:
- 0 file is opened for reading.
 1 file is opened for writing.
 2 file is opened for both reading and writing.

This routine sets the read/write pointer to the first byte of the file and sets the record size of the file to 1 byte. On normal return, AX contains a file handle. Subsequent I/O to the file must use the file handle. The DOS, on initialization, has the maximum number of files defined in CONFIG.SYS. If the file was not found, AX has 2; if too many files were open, AX has 4; if access was denied, AX has 5; if the access was invalid, AX has C.

- 3E Close a file. On entry, BX has a file handle. This routine closes the associated file and flushes the internal buffers. On return, AX is 6 if the handle was invalid.
- 3F Read a file or device. On entry, DX has the address of a buffer; CX has the number of bytes for I/O, and BX has a file handle. This routine transfers the specified number of bytes from a file or device into the buffer. It is not guaranteed that all the bytes will be read. For example, one line of text is the most read from the keyboard at one time. AX returns the number of bytes read. If AX is zero, the program tried to read from the end of file. If AX is 5, access was denied. If AX is 6 the file handle was invalid. The computer uses normalized pointers for all I/O; no segment wraparound occurs.
- 40 Write a file. On entry, DS:DX has the address of a buffer; CX has the number of bytes for I/O, and BX has a file handle. This routine transfers the specified number of bytes from the buffer to a file or device. AX returns the number of bytes written. An error has occurred if AX differs from CX. If AX has 5, access was denied. If AX has 6, the file handle was invalid. The computer uses normalized pointers for all I/O; no segment wraparound occurs.
- 41 Delete a directory entry. On entry, DS:DX contains a pointer to a file name. This routine deletes the file from its directory. If the file is currently open on another handle, then no removal will take place. On return, AX is 2 if the file was not found, 5 if access was denied.
- 42 Move file read/write pointer. On entry, CX contains the most significant 16 bits, and DX the least significant, of a 32-bit, integer offset. BX contains a file handle. AL contains a subfunction code specifying how to move the read/write pointer. The meanings of the subfunction codes are as follows:
- 0 The pointer moves offset bytes from the beginning of the file.
 - 1 The pointer moves to the current location plus the offset.
 - 2 The pointer moves to the end of file plus the offset.

Subfunctions 6 and 7 return the status, ready or not ready, of a file open for input or output. The file can be open to a device or to a disk file. The status is defined as follows:

Input

AL = FF	Ready. AL is always FF until the end of file is reached.
= 0	Not ready. After the end of file is reached, AL is always 0 unless the function call 42 has moved the current pointer.

Output

AL = FF	Ready. AL is always FF, even if the disk is full.
---------	---

NOTE:

This routine returns the I/O status as it is at the time it calls the system. On future versions, when control returns to you from the system, the status returned may not correctly reflect the true current state of the device or file.

On exit, AX returns 1 if the subfunction was invalid, 6 if the file handle was invalid, 0D if the data was invalid, and 5 if access was denied.

- 45 Duplicate a file handle. On entry, BX has a file handle. On normal return, AX has a new handle that refers to the same file at the same position. Otherwise, AX has 4 if there were too many open files or 6 if the file handle was invalid.
- 46 Force a duplicate of a handle. On entry, BX has a file handle, and CX has a new handle. The new handle now refers to the same file as the first. If there was already a file on the new handle, the old file is closed. If AX returns 6, the file handle was invalid.
- 47 Current Directory. This routine returns the current directory for a drive. On entry, DS:SI has a pointer to a 64-byte area that contains the current directory. DL contains a drive code: 0 = the default; 1 = A, 2 = B, 3 = C, and so on. On return, AX contains 0F if the drive was invalid.
- 48 Allocate memory. On entry, BX contains the size of the requested block in paragraphs. On normal return, AX:0 contains the address of a free block of memory that has the requested size. Otherwise, AX has 7 if the allocated area is bad or 8 if there was not enough memory. If there is no free block of the specified size, BX returns the size of the largest available block.

- 49 Free allocated memory. On entry, ES has the address of a block of memory allocated by function call 48. That block is returned to the system pool. On return AX has 7 if the area freed is bad, 9 if the block is invalid.
- 4A Modify allocated block. On entry, ES has the paragraph address of a block of memory allocated by function call 48, and BX has a new size specification, in paragraphs, for the block. The size of the block is increased or decreased, accordingly. If the specified size is too large, BX will contain the maximum possible size. On return, AX has 7 if the allocated area is bad, 8 if there was not enough memory, or 9 if the address specified in ES was invalid.
- 4B Execute a program. This routine allows a program to load another program into memory and begin execution of it. On entry, DS:DX points to the ASCIZ name of the file to be loaded. ES:BX points to a parameter block for the load. AL has a subfunction code. The meanings of the codes are as follows:

- 0 Load and execute the program. The routine establishes a header for the program and sets the terminate and CONTROL + C addresses to the instruction that follows this call.

NOTE:

When control returns to the calling program, by means of CONTROL + C or program termination, all registers are altered, including the stack pointer. This is because control is returned from the program you are executing, not from the system. To regain your stack, store the SS:SP value at a location addressable from your CS.

- 3 Load the program; do not create the program header, and do not begin execution. The area you specify receives the CS:IP and SS:SP of the program.

The parameter block has the following formats for the different subfunction codes:

AL = 0 (load and execute)

WORD segment address of environment
DWORD pointer to command line at 80H
DWORD pointer to default FCB to be passed at 5CH
DWORD pointer to default FCB to be passed at 6CH

AL = 3 (load overlay)

WORD segment address where file will be loaded
--

WORD relocation factor to be applied to the image

This routine duplicates all open files of a process in the child process. For example, the parent could write a series of records to a file; open the file as standard input; open a listing file as standard output, and execute a sort program that takes its input from the file open as standard input and writes it to the listing file. Use function calls 0 to C or 44 to control I/O devices.

The child process also receives an "environment" from the parent. An environment is a block of text strings that convey various configuration parameters. The block must begin on a paragraph boundary, be less than 32K bytes, and end with 00H.

Environment text strings usually have the following form:

parameter=value

If the environment address field contains zero, the child process receives the parent's environment unchanged.

On return, AX has 1 if the subfunction was invalid, 2 if the file was not found, 5 if access was denied, 8 if there was not enough memory, 9 if the parameter block was invalid, 0A if the environment was bad, or 0B if the format of the parameter block was bad.

- 4C Exit from a process. This routine terminates the current process and transfers control to the invoking process. In addition, you can supply a return code in AL. The routine closes all files open at the time.

- 4D Retrieve the return code of a child process. AX receives the return code of a child process. The low byte of this code is that sent by function call 31 or 4C. The high byte is 0 for normal termination (through function call 4C), 1 for CONTROL + C termination, 2 for a fatal error, or 3 for termination through function call 31 (terminate but stay resident). The exit code is returned only once.

- 4E Find First. This routine searches for the first file that matches a file name containing one or more all-purpose characters and has a subset of the attributes specified for the search file. On entry, DX points to a path name whose last component has all-purpose characters. CX has the attribute specification (refer to function call 43). When it finds a match, the routine writes a datablock at the current disk transfer address (refer to Section A.4). The data is formatted as follows:

<u>Byte</u>	<u>Meaning</u>
0-20	reserved. These bytes contain the search attribute, the drive number (0 = the default, 1 = A, 2 = B, and so on.), and the file name and extension, along with other information.
21	attribute found.
22-23	time of creation or last revision.
24-25	date of creation or last revision.
26-27	file size (low order bytes).
28-29	file size (high order bytes).
30-42	file name and extension, without punctuation, left-justified and filled with blanks, if necessary. The field terminates with 00H.

On return, AX has 2 if the file was not found, 12 if there are no more files in the directory.

To obtain subsequent matches to the search file use function call 4F.

- 4F Find next. This routine uses the data block written by the Find First function call (4E) to find the next entry in a directory with a matching file name and a subset of the specified attributes. On entry, the current disk transfer address (refer to Section A.4) must point to a data block returned by Find First. On return, AX has 12 if there are no more files in the directory.
- 54 Get Verify Flag. This routine returns the current write verify status in AL: 0 = verify off (the default); 1 = verify on.
- 56 Rename. This routine renames a file, either on the same path or on another path. The paths must be on the same device. On entry, DX points to the source path name, and DI points to the new path name. On exit, AX has 3 if one of the paths was not found, 5 if access was denied, and 11 if the paths were not on the same device.
- 57 File times. This routine returns or sets the time and date of the last write to a file. On entry, BX has a file handle. AL has a subfunction code: 0 = return the time and date; 1 = set the time and date. If the subfunction code is 1 on entry, CX must have the time and DX the date. If the subfunction code is 0, CX and DX return the time and date, respectively. On return, AX has 1 if the subfunction is invalid, 6 if the file handle is invalid.

C.3 CP/M CALLING CONVENTIONS

An additional mechanism is provided for programs that conform to CP/M calling conventions. The CL register receives the function number; other registers receive the values that are normal for the function specification, and an intrasegment call to location 5 in the current code segment takes place. This method is only available to functions that do not pass a parameter in AL and whose numbers are equal to or less than 36. This mechanism always destroys register AX; otherwise, this mechanism is the same as normal function requests.

APPENDIX D BIOS INFORMATION

D.1 INTRODUCTION

This appendix contains information for programmers about the Wang PC BIOS. Some functions described in this appendix may not be implemented in the current version of the BIOS. Return code 02H informs you that a requested function is not yet implemented.

A return code of 0 always indicates successful completion. Because future versions of the BIOS may have more error codes than the current version, you should write your program to check for a nonzero result.

D.2 THE SOFTWARE/BIOS INTERFACE

The software BIOS interface entry point is interrupt 88H. The BIOS interface receives a function number in the AL register and other parameters in other registers, if required. The functions available from the BIOS are as follows:

<u>Value</u>	<u>Function</u>
000H	Set ES:BX to point to the system name. For the Wang PC, the string is "Wang Professional Computer" terminated by a null (0) character.
001H	Set ES:BX to point to the system configuration table.
002H	Set event trap. BX contains the event index, CX contains the count of events to occur before issuing trap, and DS:DX contains the address of the routine to call when the trap occurs. Upon return, BX contains the trap queue identification (QID) number.
003H	Clear event trap. BX contains the QID of the event trap to clear.
004H	Reserved.
005H	Reserved.

<u>Value</u>	<u>Function</u>
006H	Send character to the screen. BL contains the character to send to the screen. This function provides programs with a faster route to the screen than the DOS provides. In the future, BH will contain an index indicating which screen to send output to, with zero being the default. It is good practice to make sure that BH is zero, thereby ensuring future compatibility.
007H	Send one or more bytes to the keyboard. If CX is 1, BL contains the byte to send to the keyboard. Otherwise, DS:DX points to a character buffer, and CX has the count. This function is useful for programs that use the sound capabilities of the keyboard. Programmers who wish to send sound chip sequences to the keyboard should send the entire 2- or 3-byte keyboard sequence in one call. This prevents a keystroke from causing an interrupt that would ordinarily send a click, beep, or CAPS LOCK LED to the keyboard. Also, the keyboard output buffer is of a fixed size. If it becomes full, any subsequent characters are ignored. If you send multiple bytes in one call, the buffer does not receive any of them unless it can hold them all. AL returns 05H if the buffer cannot hold the entire block of bytes.
008H	Reserved.
009H	Set up DMA channel. The DMA channel specified in BL is set up for the mode of transfer specified in BH. 000H indicates verify transfer; 001H indicates write transfer, and 002H indicates read transfer. If BH is 0FFH, the channel is disabled. Otherwise, CX is the number of bytes transferred (CX = 0 means 64KB) and DS:DX is the buffer address. A return code of 004H means the transfer crosses a physical 64KB boundary.
00AH	Allocate DMA channel. On exit, BL contains the channel number of the DMA channel allocated, if one was available. If none was available, AL returns with 08H as an error code. On entry, DS:DX points to the routine to be called (via a long call) when end-of-process is signaled for that channel.
00BH	Free DMA channel. On entry, BL contains the DMA channel previously allocated.
00CH	Set or clear the CONTROL + S/CONTROL + Q enable byte. BL contains the value for the enable byte. If BL is not zero, the BIOS handles CONTROL + S and CONTROL + Q for both the console and the serial port. Otherwise, these keys are buffered to the DOS as normal keys, and the DOS handles them. The default is for the DOS handle them.

NOTE:

Function 0CH affects serial I/O and I/O to the console.

Upon return, AL contains the completion code for the function. The following values are defined for this code:

<u>Value</u>	<u>Description</u>
000H	Function completed normally
001H	Function index (AL) is invalid
002H	Function specified has not been implemented
003H	Invalid parameter(s) specified
004H	Operation impossible
005H	Event queue or keyboard buffer full
006H	Invalid QID specified
007H	Invalid screen index specified
008H	Resource allocation denied
009H	Internal error of unspecifiable nature

D.3 BIOS TRAPPABLE EVENTS

In BIOS functions 002H and 003H (set and clear traps), you specify which event to trap on by the index in BX. The trap routine, located at DS:DX, is called (via a far call) after the event occurs the number of times specified by the count in CX. If the count in CX is zero, the BIOS assumes that the trap handler will go to the hardware to process the interrupt. Otherwise, the BIOS handles the interrupt as it normally does, and then calls the specified trap routine.

NOTE:

Because the trap routine call occurs within an interrupt, the trap routine cannot call the DOS to perform any functions. Also, the trap routine should not process for any length of time, and it cannot enable interrupts.

When the trap routine call occurs, AX contains the event index. Therefore, you can have a single trap routine for several conditions. Also, the routine may assume it has up to 64 bytes of stack space and that all registers are preserved. Before the trap routine call occurs, the count is reset so that the trap can occur again. Only the Clear Trap function (002H) can clear the trap. In other words, these traps are not one time traps, but repeating traps. You must use the QID number, which returns when you set the trap, to clear it.

You can trap the following events (some of which may not yet be implemented):

<u>Index</u>	<u>Event</u>
0000H	10 ms timer (see note 1)
0001H	Serial port input character ready
0002H	Serial port output character ready
0003H	Serial port data set change state (see note 2)
0004H	Parallel port input character ready
0005H	Parallel port output character ready
0006H	Keyboard input character ready
0007H	Keyboard output character ready
0008H	8087 interrupt (see note 2)
0009H	Control-C (or Shift-Cancel) trap (see note 2)
000AH	Cancel key pressed (see note 2)
000BH	Floppy disk request complete
000CH	Winchester disk request complete
000DH	Parity error
000EH	I/O error (from option slot)
000FH	Invalid interrupt attempted
0010H	Channel 0 hardware interrupt (see note 2)
0011H	Channel 1 hardware interrupt (see note 2)
0012H	Channel 2 hardware interrupt (see note 2)
0013H	Channel 3 hardware interrupt (see note 2)
0014H	Channel 4 hardware interrupt (see note 2)
0015H	Channel 5 hardware interrupt (see note 2)
0016H	Channel 6 hardware interrupt (see note 2)
0017H	Channel 7 hardware interrupt (see note 2)
0018H	DMA EOP on DMA channel 1 (see note 2)
0019H	DMA EOP on DMA channel 2 (see note 2)
001AH	DMA EOP on DMA channel 3 (see note 2)

- Notes:
- 1) You cannot request the 10 ms timer event trap with a count of zero. A bad parameter error is returned if you try.
 - 2) Events with note 2 on them will trap at each occurrence regardless of the count specified in CX. Therefore, the count specified in CX is ignored.

D.4 SYSTEM CONFIGURATION TABLE

The system configuration table is maintained by the BIOS and contains information concerning the current system configuration. You may obtain its location using the Software BIOS Interface (function 01H).

The format of the system configuration table is as follows:

<u>Offset</u>	<u>Name</u>	<u>Size</u>	<u>Description</u>
0000H	VERSION	Word	BIOS version (the high byte has the number to the left of the decimal point; the low byte has the number to the right)
0002H	MEMSIZE	Word	Size of memory in paragraphs
0004H	Reserved	Dword	Reserved for future use
0008H	SCRNCNT	Word	Count of screens in the system
000AH	SCRNPTR	Word	Screen information block pointer (repeated 0 or more times)
mmmmH	DISKCNT	Word	Count of disk drives in the system (including the two floppies)
mmmnH	FLOPPY0	Word	Floppy drive 0 control block pointer
mmmoH	FLOPPY1	Word	Floppy drive 1 control block pointer
mmmpH	WINSTAT	Word	Winchester control block pointer (repeated 0 or more times)

There is one screen information block pointer (offset 000AH) for each video card.

D.4.1 Screen Information Block

The screen information block for each video card has the following format:

<u>Offset</u>	<u>Name</u>	<u>Size</u>	<u>Description</u>
0000H	STATE	Byte	Active/Type/Slot information
0001H	SCNOFF	Byte	Scan register offset
0002H	BUFSEG	Word	Segment containing frame buffer image
0004H	COLORS	Byte	Current foreground and background colors (high nibble: foreground; low nibble:background)
0005H	ROW	Byte	Current row of cursor (0 to 24)
0006H	COL	Byte	Current column of cursor (0 to 79)
0007H	ATTR	Byte	Current attributes
0008H	AUXMOD	Byte	Auxiliary mode byte (determines card interrupt status)
0009H	AUXMD2	Byte	Second auxiliary mode byte

The STATE byte has the following format:

<u>Bit(s)</u>	<u>Name</u>	<u>Description</u>
3-0	SLOTID	Contains the slot number of the selected card.
6-4	TYPE	Contains the type of video card as follows: 0 indicates 320x225 resolution (low res) 1 indicates 640x225 resolution (low res) 2 indicates 800x300 resolution (medium res w/graphics) 3 indicates medium res character card only 4 indicates 800x600 resolution (high res)
7	ACTIVE	If set, indicates the default screen.

The ATTR byte has the following format:

<u>Bit</u>	<u>Description</u>
0	Blink
1	Reverse video
2	Blank
3	Bold
4	Overscore
5	Underscore
6	Subscript
7	Superscript

D.4.2 Disk Drive Control Blocks

The disk drive control block, defined as an Assembler structure, has the following format:

FCB	Struc	
SxTYPE	Db ?	;Disk type (see "Disk Type", below)
SxSTATE	Db ?	;Drive state (see "Status Byte", below)
SxBPS	Db ?	;Bytes per sector
SxSPT	Db ?	;Sectors per track
SxHPC	Db ?	;Heads per cylinder
SxCPD	Dw ?	;Cylinders per disk
SxSPD	Dw ?	;Sectors per disk
FCB	Ends	

Disk Type

The disk type byte (SxTYPE) indicates which type of disk drive the control block contains information about. The currently defined types are 00H for floppies, 01H for Winchesters, 02H for RAM disks, and FFH for other.

Status Byte

The status byte (SxSTATE) is encoded as follows:

<u>Bit(s)</u>	<u>Name</u>	<u>Meaning if Bit Set On</u>
7	EXIST	The drive exists.
6	WP	The disk is write protected.
5	ACTIVE	The drive is active.
4	SELECT	The drive is selected.
3	MEDIA	The disk may have been changed.
2	DISK	A disk is in the drive.
1	MOTOR	The motor is on.
0	INIT	The drive has been initialized.

NOTE:

Although some bits may not pertain to some types of disk, they are usually set according to their corresponding meanings.

APPENDIX E DEVICE DRIVERS

E.1 INTRODUCTION

This appendix provides information on the structure and functions of device drivers. The information is intended for those who want to write their own drivers or access an existing driver. Only experienced programmers should attempt to write or access a driver.

E.2 DEVICE INTERFACE

The MS-DOS device interface for Version 2 is not designed just to interface with the Version 2 DOS, but is a more general interface designed for future versions of the DOS as well.

The new interface does not use a jump table at a fixed location as previous versions did; instead, it links devices together in a chain. Chaining allows you to write and install new drivers for optional hardware.

Each driver in the chain defines two entry points: one for an enqueue routine and one for a service routine. Version 2 does not use the two entry points with full functionality; it simply calls the enqueue routine, then immediately calls the service routine.

The dual entry point scheme is meant to facilitate future multi-tasking versions of MS-DOS. In multi-tasking environments, I/O must be concurrent with CPU functions. To accomplish this, the enqueue routine will be called to queue a request and return quickly. In queuing requests, it is no longer sufficient to pass I/O information in registers, because many requests can be pending at any one time. Therefore, the enqueue routine will use a data block to pass request information. The device driver has a pointer to the data block, and the call to the enqueue routine will place the data block pointer in the queue.

It is the responsibility of the service routine to perform the actual I/O by processing requests from the queue set up by the enqueue routine. The call to the service routine passes no parameters. The service routine performs the operation based on the queued data block. When it has finished processing a request, the service routine sets up any returns and flags the request as done.

In addition to the chain of requests for a particular driver, the DOS will link the data block into a global chain of all pending I/O requests. The device driver will then link the block into its own local chain of requests for this particular device. The DOS will periodically scan the global chain looking for requests flagged as done, and wake up the requests waiting for processing. The device service routine will take requests for processing from the local chain.

MS-DOS Version 2 does not implement most of these features, as future versions will. There is no global or local queue. Only one request is pending at any one time, and the DOS waits for this current request to be completed. For Version 2, it is sufficient for the enqueue routine to store the address of the data block at a fixed location, and for the service routine to process this block by performing the request and returning. However, for compatibility with future versions of MS-DOS, the driver should link and unlink the block to the local chain. When the service routine returns control to the DOS, the DOS assumes that the request is complete.

Device drivers can call DOS functions only in the INIT code (refer to Section E.4). To set hardware interrupts, drivers must use BIOS software interface routines (refer to Appendix D). You do not have to save registers when you call hardware interrupts from the BIOS; the BIOS routines save them.

E.3 FORMAT OF A DEVICE DRIVER

A device driver is a .COM file with all of the code necessary to implement the device. In addition, the file has a special header that identifies it as a device driver, defines the enqueue and service entry points, and specifies various attributes.

There are two basic types of devices: character devices and block devices. Character devices do character I/O in a serial manner. The BIOS installs four character devices with the names CON, AUX, PRN, and CLOCK. These drivers are always present. You can perform I/O by opening FCB's for them. Additionally, you can assign user-defined names to these devices and to other devices for which you can also open FCBs.

Block devices are the disk drives on the system. They perform random I/O in units called blocks, usually equal to the physical sector size. These devices do not have names as character devices do. Therefore you cannot open them directly. Instead, block devices are mapped by the drive letters (A,B,C, etc.).

Character device drivers cannot define multiple units because character devices have only one name. A single block device driver, however, can be responsible for one or more disk drives. For instance, block device driver ALPHA can be responsible for drives A,B,C and D. That is, ALPHA can have four units defined and, therefore, takes up four drive letters. The position of the driver in the chain of drivers determines which units correspond to which drive letters. For instance, if driver ALPHA is the first block driver in the

device chain and it defines 4 units, then these four units will be A,B,C and D. If BETA is the second block driver and defines three units, then BETA's three units will be E, F and G, and so on.

The theoretical limit for block device units is 63 (2^6-1). After 26, however, the drive designators cease to be alphabetic.

The device driver header begins at CS:00. The following is the format of the device driver header:

DWORD Pointer to next driver header (-1 if last device)
WORD Attributes Bit 15 = 1, if char device; 0, if blk If bit 15 is 1 Bit 0 = 1 if Current sti device Bit 1 = 1 if Current sto output Bit 2 = 1 if Current NUL device Bit 3 = 1 if Current CLOCK dev Bit 14 is the IOCTL bit (see below) Bit 13 is the NONIBM FORMAT bit
WORD Pointer to Device enqueue entry point
WORD Pointer to Device service entry point
8-BYTE character device name field. Character devices set a device name. 1-byte number of units for block devices (set by SYSINIT).

The pointer to the next device must be set to -1 by the driver because this is the last device at the time of installation. The DOS changes this field if there are other devices to point to. The device entry points are words. They must be offsets from the same segment number used to point to this table.

The most important function of the attribute field is to inform the system whether this device is a block or character device (bit 15). Most of the other bits give selected character devices certain special treatment. (These bits mean nothing for a block device.) For example, assume you want a new device to be the standard input (STI) and standard output (STO) device. In addition to installing the driver, you must inform SYSINIT and the DOS that the new device is to override the current STI and STO (the "CON" device). To do this, you set Bits 0 and 1 to the value 1.

NOTE:

Although there is a NUL device attribute, you cannot reassign the NUL device. This attribute exists so that the DOS knows if the NUL device is in use.

The NONIBM Format bit applies only to block devices and effects the operation of the get BIOS Parameter Block (BPB) device call (refer to Section E.7.4).

The IOCTL bit has meaning on both character and block devices. This bit tells the DOS whether this device can receive control strings from the IOCTL system call (refer to Section C.2.2).

IOCTL functions allow data to be sent and received by the device, itself, for its own use (for example, to set the baud rate, stop bits, or form length), instead of passing data over the device channel as a normal read or write does. The interpretation of the information passed depends on the device, but the data is not usually treated as a normal I/O.

If a driver cannot process control strings, you should initially set this bit to 0. Zero tells the DOS to return an error code on an attempt to send or receive control strings (via the IOCTL system call) to this device. For a device which can process control strings, you should initialize this bit to 1. For devices of this type, the DOS makes calls to the IOCTL input and output device functions to send and receive IOCTL strings.

In order for SYSINIT to be able to install a device driver, you must create a .COM file with a device driver header. You should set the link field to -1. SYSINIT may change it after the driver is installed. You must set the attribute field and entry points correctly. If the device is a character device, you must fill in the name field. This name can be any legal 8-character file name. However, SYSINIT always installs devices at the start of the device list. Therefore, if you want to install a new CON device, all you have to do is name it CON. The new device is ahead of the old one in the list. As a result, the old device disappears because the search for devices stops on the first match. Be sure to set the STI and STO bits on for a new CON device. If the device is a block device, SYSINIT fills in the name field with the correct unit count.

NOTE:

Because SYSINIT may install the driver anywhere, you must be very careful about FAR memory references. You should not expect that your driver will go in the same place every time.

E.4 INSTALLATION OF DEVICE DRIVERS

Unlike past versions MS-DOS Version 2 and future versions allow new device drivers to be installed dynamically at startup time. This is accomplished by the SYSINIT module that reads and processes the CONFIG.SYS file. CONFIG.SYS must have the line `DEVICE = <filename>`, where <filename> is the device driver file.

One of the routines defined for each device is INIT. This routine is called by SYSINIT upon device installation. It is never called again. INIT returns the doubleword break address field in the INIT data block (refer to Section E.7.7). This is a pointer to the first free byte of memory after the device driver. You must set this pointer. SYSINIT uses it to install the next device in the chain or to install the menu driver or command processor if there are no other drivers to install. You can use this pointer to save space by discarding the initialization code. The free byte address should normally be the first byte of the INIT routine.

SYSINIT installs block devices in the same way it installs character devices. The INIT routine for block devices will set the INIT data block (refer to Section E.7.7) to return the first free byte pointer and additional information. It returns the number of device units, thereby determining the device letters. If the maximum device letter is F at the time of the install call and the INIT routine returns 4 as the number of units, then the new units receive the designations G, H, I and J. The position of the driver in the device list and the number of units in the device driver (stored in the first byte of the device name field) determine the mapping of units to letters.

The INIT routine for block devices must also set and return a pointer to an array of N word pointers to BPBs, where N is the number of units defined. A BPB is similar to the INIT table used in versions of MS-DOS previous to Version 2. The BPB, however, contains more information. The DOS uses these blocks to build a device parameter block (DPB) for each of the units. There is one BPB pointer for each unit. If all units are the same, all of the pointers can point to the same BPB, thereby saving space.

NOTE:

The BPB pointer array must be protected (below the first free byte returned by the INIT routine) because the DPB starts at the byte indicated by the first free byte.

The format of a BPB is as follows:

WORD Sector size in Bytes
BYTE Sectors/Allocation unit
WORD Number of reserved sectors before the first FAT
BYTE Number of FATS
WORD Maximum number of disk directory entries
WORD Total number of sectors
BYTE Media descriptor
WORD Number of sectors occupied by each FAT

The sector size defined must be less than or equal to the current maximum sector size. If not, the installation will fail.

One piece of DPB information set from the BPB is the media descriptor byte. This byte means nothing to the DOS, but is passed to devices so that they know what form of a DPB the DOS is currently using for a particular drive. Block devices may take several approaches; they may be dumb or smart. A driver for a dumb device would define a unit (and therefore a DPB) for each possible media drive combination. Unit 0 = drive 0 single sided, unit 1 = drive 0 double sided, and so on. For this approach, media descriptor bytes would mean nothing. A driver for a smart device would allow multiple media per unit. In this case, the BPB table returned by INIT must define space large enough to accommodate the largest possible media supported. Drivers for smart devices use the media byte to pass information about what media is currently in a unit.

NOTE:

If the DPB is made to get the right sizes for a smart device, it should give an invalid media byte back to the DOS.

When the default BIOS drivers are installed, the preset device list is scanned. If block drivers are found, they are installed as above, except that the break address does not change because the drivers are already resident in the BIOS. Drive letters are assigned in list order. Thus, the drive which is to have logical A must be the first unit of the first block device in the list. The order of character devices is also important. There must be at least 4 character devices defined at startup. These must be the first four devices of either type. The first character device defined becomes the STI, STO, and standard error output device. The second becomes the standard auxiliary input and output device. The third becomes the standard list output device. The fourth becomes the date/time (CLOCK) device. Thus the BIOS device list must appear as follows:

->CON->AUX->PRN->CLOCK->any other block or character devices

E.5 DRIVER FUNCTIONS

A device driver defines the following functions:

<u>Command</u> <u>Code</u>	<u>Function</u>
0	INIT
1	MEDIA CHECK (Block only, NOP or error return for character)
2	BUILD BPB " " " " "
3	IOCTL INPUT (Only called IF DEVICE HAS IOCTL)
4	INPUT (read)
5	NON-DESTRUCTIVE INPUT NO WAIT (Char devs only)
6	INPUT STATUS " " "
7	INPUT FLUSH " " "
8	OUTPUT (write)
9	OUTPUT (write) with verify
10	OUTPUT STATUS " " "
11	OUTPUT FLUSH " " "
12	IOCTL OUTPUT (Only called if device has IOCTL)

The DOS FATREAD logic explains the BUILD BPB and MEDIA CHECK functions. The FATREAD logic is as follows:

1. Turn drive letter into DPB pointer by looking for DPB with correct driver-unit number.
2. Call media check routine for Drive-Unit. DOS passes in its current Media descriptor byte (from DPB). The call returns are as follows:

<u>Code</u>	<u>Meaning</u>
-------------	----------------

01H	Media Not Changed
FFH	Media Changed
00H	Not Sure
Other	Error

Media Not Changed	Current DPB and media byte are correct; the media check routine is complete.
Media Changed	Current DPB and media are wrong; invalidate any buffers for this unit, and go to 3.
Not Sure	If there are dirty buffers for this unit, assume the DPB and the media byte are all right and done. If there are no dirty buffers, assume the media changed, invalidate any buffers for this unit, and go to 3.
Error	If an error occurs, the driver should set the error code in the status word of the static request header accordingly.

NOTE:

If a DPB for a smart device is built at initialization and an invalid Media byte is set, the driver should return media changed when it encounters this invalid media byte.

3. Call build BPB with media byte and buffer.

What the driver must now do is determine the media that is currently in the unit and return a pointer to a BPB table. This table will be used at init to build a correct DPB for the unit (the difference being that no space is created because it is assumed enough space was reserved). If the determined media descriptor byte in the table turns out to be the same as the one passed in, then the DOS will not build a new table, instead it uses the old one. Therefore, in this case, the driver need not fill in the other entries.

The build BPB call also gets a pointer to a one sector buffer. The NONIBM FORMAT bit in the attribute field determines what this buffer contains. If the NONIBM FORMAT bit is set, then the pointer points to one sector of scratch space which you can use for anything. If the bit is zero (device is IBM format compatible), the buffer contains the first sector of the FAT. The FAT ID byte is the first byte of this buffer. The location of the FAT must be the same for all possible media because the first FAT sector must be read BEFORE the actual BPB is returned.

On return, if the error flag in the status word is off, the DPB is set up and the function complete. If the error flag is on, the BIOS should have set unknown media.

E.6 THE STATIC REQUEST HEADER

The data block for the device driver interrupt has two parts: a static request header that has the same format for all requests, and a part that has information specific to the request. The second part has a variable size and format. The driver defines one entry point, and the DOS passes a pointer to the data block in ES:BX.

NOTE:

It is the driver's responsibility to preserve machine state.

The structure of the static request header is as follows:

BYTE length of record length in bytes of this Driver Request Structure
BYTE unit code The subunit that the operation is for (minor device). (No meaning on character devices.)
BYTE command code
WORD Status
8 bytes reserved here for two DWORD links. One will be a link for the DOS queue; the other for the device queue.

The structure of the status word is as follows:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
E R R	RESERVED						B U I	D O N	ERROR CODE (bit 15 on)						

The status word is zero on entry. The driver service routine sets it on return. Bit 8 is the done bit, indicating that the operation is complete. For the time being, the Driver code must always set it on. In the future, the service routine will set it to tell the DOS the operation is complete.

Bit 15 is the error bit. If it is set, then the low 8 bits indicate the error as follows:

- 0 Write Protect violation
- 1 Unknown Unit
- 2 Drive not ready
- 3 Unknown command
- 4 CRC error
- 5 Bad Driver Request Structure length
- 6 Seek error
- 7 Unknown media
- 8 Sector not found
- 9 Printer out of paper
- A Write Fault
- B Read Fault
- C General Failure

Bit 9 is the busy bit that is set only by status calls as follows:

For output on character devices: A return of 1 means a write request would wait for completion of a current request. A return of 0 means, there is no current request and a write request would start immediately.

For input on character devices with a buffer: A return of 1 means a read request would go to the physical device. A return of 0 means, there are characters in the device buffer and a read would return quickly. It also indicates that the user has entered something.

The DOS assumes all character devices have an input type-ahead buffer. Devices which don't have them should always return 0 for the busy bit. If not, the DOS can hang while waiting for input to a nonexistent buffer.

E.7 DATA BLOCK FORMATS

This section gives the data block formats for each driver function.

E.7.1 Read or Write - ES:BX (Including IOCTL)

13-BYTE Static Request Header
BYTE Media descriptor from DPB (no real meaning on Char Devs)
DWORD transfer address
WORD byte/sector count
WORD starting sector number (no real meaning on Char Devs)

In addition to setting the status word, the driver must set the byte/sector count to the actual number of bytes or sectors transferred.

NOTE:

No error check is performed on an IOCTL I/O call. However, the driver must correctly set the return sector (byte) count to the actual number of bytes transferred.

E.7.2 Nondestructive Read No Wait - ES:BX

13-BYTE Static Request Header
BYTE read from device

If the character device returns Busy bit = 0 (characters in buffer), the next character to be read is returned. This character is NOT removed from the input buffer (hence the term Nondestructive Read). In other words, this call allows the DOS to look ahead one input character.

E.7.3 Media Check - ES:BX

13-BYTE Static Request Header
BYTE Media Descriptor from DPB
BYTE returned

In addition to setting the status word, the driver must set the return byte as follows:

Return Byte	-1	Media has been changed
	0	Unknown whether the media has been changed
	1	Media has not been changed

E.7.4 Build BPB - ES:BX

13-BYTE Static Request Header
BYTE Media Descriptor from DPB
DWORD Transfer Address (points to one sector's worth of scratch space or first sector of FAT depending on the value of the NONIBM FORMAT bit).
DWORD Pointer to BPB

If the NONIBM Format Bit of the device is set, the DWORD Transfer Address points to a 1-sector buffer that you can use for any purpose. If the NONIBM FORMAT bit is 0, this buffer contains the first sector of the FAT. In this case, the driver must not alter this buffer. (This mode is useful if all that is desired is to read the FAT ID byte.)

If an IBM compatible format is in effect (the NONIBM Format Bit = 0), the first sector of the first FAT must be located at the same sector on all media. This is because the FAT sector will be read BEFORE the media is actually determined.

In addition to setting the status word, the driver must set the pointer to the BPB on return.

The information relating to the BPB for a particular piece of media is kept in the startup sector for the media. The format of the startup sector is as follows:

	3 BYTE near JUMP to startup code
	8 BYTES OEM name and version
B	WORD bytes per sector
P	BYTE sectors per allocation unit
B	WORD reserved sectors for startup
	BYTE number of FATs
	WORD number of root dir entries
	WORD number of sectors in logical image
B	BYTE media descriptor
P	WORD number of FAT sectors
B	
	WORD sectors per track
	WORD number of heads
	WORD number of hidden files
	WORD number reserved sectors for Startup, FATS and Directory

Refer to Section A.4.3 for the values in the startup sector determined by the various diskette formats.

The three words at the end are optional. The DOS does not need them because they are not part of the BPB. They are intended to help the BIOS understand the media. Sectors per track may be redundant (could be figured out from total size of the disk). Number of heads is useful for supporting different multi-head drives which have the same storage capacity but a different number of surfaces. Number of hidden sectors is useful for supporting drive partitioning schemes.

Currently, the flag bits of the media descriptor byte are defined as follows:

<u>Bit</u>	<u>Meaning</u>
0	On = double-sided Off = single-sided
1	On = 16 sectors per track Off = 18 sectors per track
2	On = 512 byte sectors Off = 256 byte sectors

E.7.5 Status Calls - ES:BX

13-BYTE Static Request Header

All the driver must do is set status word. Refer to Section E.6.

E.7.6 Flush Calls - ES:BX

13-BYTE Static Request Header

This call tells the driver to delete all pending requests that it has knowledge of. Its primary use is to clear the input queue on character devices.

E.7.7 Init - ES:BX

13-BYTE Static Request Header
BYTE number of units (not used by Character devices)
DWORD Break Address
DWORD Pointer to BPB array (not set by Character devices)

The driver sets the number of units, the break address, and the BPB pointer.

E.8 MODIFYING CONFIG.SYS TO INSTALL A DRIVER

The Wang PC system diskette contains a file called CONFIG.SYS. CONFIG.SYS has a list of installation-specific parameters used at system start-up. To add a device to the system configuration, you must add the following line to CONFIG.SYS using the Editor:

```
DEVICE = <drivername>
```

where <drivername> is the device driver file name. You can use DRIVER instead of DEVICE for the key word. Refer to Section A.3 for more information on CONFIG.SYS.

After you have edited CONFIG.SYS, copy it and the device driver to the system disk and restart the system. The DOS installs the driver as soon as the copyright notice appears. If the DOS does not find the driver file, a message that the driver file is bad or missing appears. If it finds the file, the DOS loads it in low memory and immediately calls the INIT routine.

The INIT routine should print the driver's name and version number on the screen. This is so that users are aware that a special device has been installed.

CAUTION:

If you modify CONFIG.SYS, be very careful. If you make a mistake, you may not be able to start your system. Always keep backups of the original CONFIG.SYS.

APPENDIX F LOGICAL KEYBOARD PROTOCOL

F.1 INTRODUCTION

This appendix explains the logical keystrokes available to all application programs that run on the Wang PC. Logical keystrokes are independent of the different physical keyboards available with the Wang PC. The file BIOS.SYS contains tables for translating the codes received from a specific physical keyboard into the logical keystrokes described in this appendix. Therefore, the specifications presented in the following tables are valid for all languages for which the Wang PC supplies keyboards.

There are three categories of logical keystrokes: simple characters, special characters, and dead keys. 16-bit codes define the keystrokes. For simple characters, the high order byte is 00H. For special characters, the high order byte is 1FH. For dead keys, the high order byte is 1EH. The system stores the high order byte first. However, the system does not send the 00H byte to the keyboard buffer.

F.2 SIMPLE CHARACTERS

Simple characters include the printable, WISCII I character set from 20H to FDH and control characters between 01H and 1BH.

F.2.1 The WISCII I Character Set

The WISCII I character set, Table F-1, is the standard (nonscientific) WISCII character set from 20H to FDH.

Table F-1. The WISCII I Character Set

Decimal	Hex	Character	Explanation
032	20		Space
033	21	!	Exclamation point
034	22	"	Double quote
035	23	#	Number sign
036	24	\$	Dollar sign
037	25	%	Percent sign
038	26	&	Ampersand
039	27	'	Single quote
040	28	(	Left parenthesis
041	29	)	Right parenthesis
042	2A	*	Asterisk
043	2B	+	Plus sign
044	2C	,	Comma
045	2D	-	Minus sign
046	2E	.	Period
047	2F	/	Slash
048	30	0	0
049	31	1	1
050	32	2	2
051	33	3	3
052	34	4	4
053	35	5	5
054	36	6	6
055	37	7	7
056	38	8	8
057	39	9	9
058	3A	:	Colon
059	3B	;	Semicolon
060	3C	<	Less than sign
061	3D	=	Equal sign
062	3E	>	Greater than sign
063	3F	?	Question mark
064	40	@	At sign
065	41	A	A
066	42	B	B
067	43	C	C
068	44	D	D
069	45	E	E
070	46	F	F
071	47	G	G
072	48	H	H

Table F-1. The WISCII I Character Set (continued)

Decimal	Hex	Character	Explanation
073	49	I	I
074	4A	J	J
075	4B	K	K
076	4C	L	L
077	4D	M	M
078	4E	N	N
079	4F	O	O
080	50	P	P
081	51	Q	Q
082	52	R	R
083	53	S	S
084	54	T	T
085	55	U	U
086	56	V	V
087	57	W	W
088	58	X	X
089	59	Y	Y
090	5A	Z	Z
091	5B	[	Left bracket
092	5C	\	Backslash
093	5D	]	Right bracket
094	5E	↑	Up arrow
095	5F	¯	Underscore
096	60	`	Open quote
097	61	a	a
098	62	b	b
099	63	c	c
100	64	d	d
101	65	e	e
102	66	f	f
103	67	g	g
104	68	h	h
105	69	i	i
106	6A	j	j
107	6B	k	k
108	6C	l	l
109	6D	m	m
110	6E	n	n
111	6F	o	o
112	70	p	p
113	71	q	q
114	72	r	r

Table F-1. The WISCII I Character Set (continued)

Decimal	Hex	Character	Explanation
115	73	s	s
116	74	t	t
117	75	u	u
118	76	v	v
119	77	w	w
120	78	x	x
121	79	y	y
122	7A	z	z
123	7B	{	Left brace
124	7C		Vertical bar
125	7D	}	Right brace
126	7E	~	Approximate symbol
127	7F	¢	Cent sign
128	80	°	Degree sign
129	81	◆	Center graphic
130	82	►	Tab graphic
131	83	◄	Return graphic
132	84	→	Indent graphic
133	85	┌	Decimal tabulation graphic
134	86		Format graphic
135	87	■	Stop graphic
136	88	!!	Note graphic
137	89	↕	Merge graphic
138	8A	↓	Down arrow
139	8B	↑	Up arrow
140	8C	←	Left arrow
141	8D	±	Plus-or-minus sign
142	8E	!	Inverted exclamation point
143	8F	?	Inverted question mark
144	90	ˆ	A circumflex
145	91	˘	A grave
146	92	˙	A acute
147	93	¨	A umlaut
148	94	˜	A tilde
149	95	←	Left arrow
150	96	Å	A angstrom
151	97	↓	Down arrow
152	98	Æ	Ligature AE
153	99	¸	C cedilla
154	9A	‡	Double dagger
155	9B	•	Bullet

Table F-1. The WISCII I Character Set (continued)

Decimal	Hex	Character	Explanation
156	9C	Ê	E circumflex
157	9D	É	E grave
158	9E	Ê	E acute
159	9F	Ë	E umlaut
160	A0	à	a circumflex
161	A1	á	a grave
162	A2	â	a acute
163	A3	ä	a umlaut
164	A4	ã	a tilde
165	A5	→	Right arrow
166	A6	Å	a angstrom
167	A7	†	Dagger
168	A8	æ	Ligature ae
169	A9	ç	c cedilla
170	AA	□	Ballot box
171	AB		Required space
172	AC	ê	e circumflex
173	AD	é	e grave
174	AE	ë	e acute
175	AF	ë	e umlaut
176	B0	Ġ	G hacek
177	B1	IJ	IJ ligature
178	B2	ı	Dotted I
179	B3	î	I circumflex
180	B4	í	I grave
181	B5	ï	I acute
182	B6	ï	I umlaut
183	B7	L·L	Catalonian LL ligature
184	B8	Ñ	N tilde
185	B9	Ô	O circumflex
186	BA	Ó	O grave
187	BB	Ô	O acute
188	BC	Ö	O umlaut
189	BD	Õ	O tilde
190	BE	Œ	OE ligature
191	BF	Ø	Slashed O
192	C0	ġ	g hacek
193	C1	ij	ij ligature
194	C2	ı	Dotless i
195	C3	î	i circumflex
196	C4	í	i grave
197	C5	ï	i acute

Table F-1. The WISCII I Character Set (continued)

Decimal	Hex	Character	Explanation
198	C6	ï	i umlaut
199	C7	l·l	Catalonian ll ligature
200	C8	ñ	n tilde
201	C9	ô	o circumflex
202	CA	ò	o grave
203	CB	ó	o acute
204	CC	ö	o umlaut
205	CD	õ	o tilde
206	CE	œ	oe ligature
207	CF	ø	Slashed o
208	D0	Þ	Icelandic thorn
209	D1	Ð	Icelandic Eth
210	D2	Ý	Y acute
211	D3	Ş	S cedilla
212	D4	‘	Open quote
213	D5	Û	U circumflex
214	D6	Ù	U grave
215	D7	Ú	U acute
216	D8	Ü	U umlaut
217	D9	©	Copyright symbol
218	DA	®	Registered trademark symbol
219	DB	℞	Prescription symbol
220	DC	ª	a superior (for number abbreviation)
221	DD	«	European open quotes
222	DE	§	Section symbol
223	DF	¶	Paragraph symbol
224	E0	þ	Icelandic thorn
225	E1	ð	Icelandic eth
226	E2	ý	y acute
227	E3	ş	s cedilla
228	E4	’	Close quote
229	E5	û	u circumflex
230	E6	ù	u grave
231	E7	ú	u acute
232	E8	ü	u umlaut
233	E9	™	Trademark symbol
234	EA	⌘	International monetary symbol
235	EB	↔	Double arrow
236	EC	º	o superior (for number abbreviation)

Table F-1. The WISCII I Character Set (continued)

Decimal	Hex	Character	Explanation
237	ED	»	European close quotes
238	EE	ß	Beta
239	EF	.	Centered dot
240	F0	£	Pound sterling
241	F1	f	Florin
242	F2	¥	Yen
243	F3	¼	1/4
244	F4	½	1/2
245	F5	¾	3/4
246	F6	ˆ	Accent circumflex
247	F7	˘	Accent grave
248	F8	˙	Accent acute
249	F9	¨	Accent umlaut (diaeresis)
250	FA	˜	Accent tilde
251	FB	¸	Accent cedilla
252	FC	ˇ	Accent hacek
253	FD	˘	Accent breve

F.2.2. Control Characters

To generate control characters, press CONTROL and another key simultaneously. The same control character results whether the other key is shifted or unshifted, except for CONTROL + [, where [must be shifted. Table F-2 contains the control characters from 01H to 1BH.

Table F-2. Control Characters

Hex Value	Keystroke Sequence
01	CONTROL + A
02	CONTROL + B
03	CONTROL + C or SHIFT + CANCEL
04	CONTROL + D
05	CONTROL + E
06	CONTROL + F
07	CONTROL + G
08	CONTROL + H or BACKSPACE
09	CONTROL + I or TAB
0A	CONTROL + J

Table F-2. Control Characters (continued)

Hex Value	Keystroke Sequence
0B	CONTROL + K
0C	CONTROL + L
0D	CONTROL + M or RETURN
0E	CONTROL + N
0F	CONTROL + O
10	CONTROL + P
11	CONTROL + Q
12	CONTROL + R
13	CONTROL + S
14	CONTROL + T
15	CONTROL + U
16	CONTROL + V
17	CONTROL + W
18	CONTROL + X
19	CONTROL + Y
1A	CONTROL + Z
1B	CONTROL + [

F.3 SPECIAL CHARACTERS

Special characters include various codes such as function keys, cursor control keys, ERASE, HELP, CANCEL, and PRINT. Note, however, that SHIFT, CAPS LOCK, CONTROL, and 2ND have no codes assigned to them.

There are 64 possible logical function keys with codes 1F80H to 1FBF. Of these, only the first 32 function keys, 1F80H to 1F9FH, are defined. Codes 1F61H to 1F7AH are also undefined. Table F-3 contains the remaining special characters with their shifted and unshifted hex values.

Table F-3. Special Character Keys

Key	Unshifted	Shifted
Cancel	1FE0	0003
Help	1FE1	1FF1
Glossary	1FE2	1FF2
Print	1FE3	1FF3
North cursor	1FC0	1FD0
South cursor	1FC1	1FD1
East cursor	1FC2	1FD2
West cursor	1FC3	1FD3
Home	1FC4	1FD4
Execute	1FC5	1FD5
Insert	1FC6	1FD6
Delete	1FC7	1FD7
Previous	1FC8	1FD8

Table F-3. Special Character Keys (continued)

Key	Unshifted	Shifted
Next	1FC9	1FD9
Erase	1FCB	1FDB
Tab	0009	1FDC
Backtab	1FCD	1FDD

F.4 DEAD KEYS

Dead key sequences are keystrokes that combine with one or more other keystrokes to produce one printable character. Dead key codes are between 1E20 and 1EFF. These are intermediate codes. They are replaced with the code for the combined keystrokes when you press the final key in the sequence.

The restart key sequence exemplifies the use of dead keys. 2ND + COMMAND produces an intermediate 1E code. The intermediate code is replaced by the code for the restart when you press CANCEL. If you enter an invalid dead key sequence, the code of the last nondead key you press is what the keyboard buffer receives.

A keyboard must be in one of two dead key modes: visible mode or invisible mode. When you press dead keys in visible mode, codes corresponding to the keys are put into the buffer. The code can have a printable graphic associated with it. If not, a dead key blank is placed into the buffer. When you press dead keys in the invisible mode, the operating system receives the codes, but does not store them in the buffer. The only code the buffer receives is the combined code generated when you press the final key in the dead key sequence. The default keyboard mode for dead keys is invisible.

APPENDIX G SCREEN MANIPULATION

G.1 INTRODUCTION

The Wang BIOS recognizes a series of 1-byte control codes and ANSI standard escape sequences that initiate specific screen operations. Section G.2 lists the control codes, and Section G.3 lists the escape sequences.

G.2 CONTROL CODES

The BIOS recognizes the following 1-byte control codes:

<u>Code</u>	<u>Meaning</u>
07H	Sound the tone
08H	Nondestructive backspace
09H	Cursor right (non-destructive)
0AH	Line feed (scroll if necessary)
0CH	Screen clear and cursor home
0DH	Carriage return
01BH	Escape sequence mode

G.3 ESCAPE CODE SEQUENCES

This section explain the various escape sequences and includes the format of each sequence. Definitions of the parameters used in the sequence appear after the formats. Each sequence ends with an alphabetic character that identifies the specific escape sequence.

For all sequences, the parameters are ASCII encoded decimal numbers, containing the digits 0 through 9. The BIOS ignores all embedded blanks and control characters. The control code for entering escape sequence mode is 01BH.

G.3.1 Set Graphics RenditionFunction

This sequence causes subsequent output to be produced with the attributes you select. Table G-1 lists these attributes.

Format

ESC "[" Ps ";" ... ";" Ps "m"

Ps values from Table G-1; the default is 0

Table G-1. Graphics Rendition Escape Sequences

Value	Attribute
0	All attributes off
1	Bold on
2	Subscript on
3	Superscript on
4	Underscore on
5	Blinking on
6	Overscore on
7	Reversed video on
8	Concealed on
30	Set color 0 foreground
31	Set color 1 foreground
32	Set color 2 foreground
33	Set color 3 foreground
34	Set color 4 foreground
35	Set color 5 foreground
36	Set color 6 foreground
37	Set color 7 foreground
40	Set color 0 background
41	Set color 1 background
42	Set color 2 background
43	Set color 3 background
44	Set color 4 background
45	Set color 5 background
46	Set color 6 background
47	Set color 7 background
50	Set color 8 foreground
51	Set color 9 foreground
52	Set color 10 foreground
53	Set color 11 foreground
54	Set color 12 foreground
55	Set color 13 foreground
56	Set color 14 foreground
57	Set color 15 foreground

Table G-1. Graphics Rendition Escape Sequences (continued)

Value	Attribute
60	Set color 8 background
61	Set color 9 background
62	Set color 10 background
63	Set color 11 background
64	Set color 12 background
65	Set color 13 background
66	Set color 14 background
67	Set color 15 background
80	Set color 0 highlight
81	Set color 1 highlight
82	Set color 2 highlight
83	Set color 3 highlight
84	Set color 4 highlight
85	Set color 5 highlight
86	Set color 6 highlight
87	Set color 7 highlight
88	Set color 8 highlight
89	Set color 9 highlight
90	Set color 10 highlight
91	Set color 11 highlight
92	Set color 12 highlight
93	Set color 13 highlight
94	Set color 14 highlight
95	Set color 15 highlight

G.3.2 Switch Default Screen to Specified ScreenFunction

This escape sequence selects the specified screen as the default screen. If the specified screen is not found, the current default screen remains the source.

Format

ESC "/" Ps "s"

Ps 1 = the first screen, 2 = the second screen, etc.

G.3.3 Set or Clear Selected Modes and LED Control CodesFunction

This sequence sets or clears the modes listed in Table G-2.

Format

ESC "[" Ps ";" ... ";" Ps $\left\{ \begin{array}{l} \text{"h"} \\ \text{"l"} \end{array} \right\}$

Ps one of the values from Table G-2; the default is 0.

"h" sets the selected modes.

"l" clears the selected modes.

Table G-2. Screen Modes and LED Control Codes

Value	Mode
4	Insert mode. While insert mode is active, printed characters are inserted at the cursor position.
5	Cursor on/off. "l" sets the cursor on; "h" turns it off.
6	Soft scroll (low resolution card only)
7	40/80 column mode (low resolution card only). "h" selects 80 column mode; "l" selects 40 column mode.
8	Cursor attach mode. When the mode is set, the cursor moves with character output; when the mode is clear, the cursor stays at its current location.
9	End of line autowrap mode. When the mode is set, a right cursor movement from the right margin moves down to the left margin on the next line; when the mode is clear, the last character of a line is overwritten.
10	Control LED 0 (LOCK)
11	Control LED 1
12	Control LED 2
13	Control LED 3
14	Control LED 4
15	Control LED 5

NOTE:

Only 80-column monitors allow a choice of 40 or 80 characters per line. In 80 column mode, 4 colors can be displayed simultaneously. In 40 column mode, up to 16 colors can be displayed simultaneously.

G.3.4 Clear All or Part of the ScreenFunction

This sequence erases from the start of the screen to the cursor, from the cursor to the end of the screen, or the entire screen.

Format

ESC "[" Ps "J"

Ps one of the following values (the default is 0):

<u>Value</u>	<u>Action</u>
0	Erases from cursor to end of screen (including character at cursor).
1	Erases from start of screen to cursor.
2	Erases entire screen and home cursor.

G.3.5 Clear All or Part of Current LineFunction

This sequence erases from the cursor to the end of the line, from the start of the line to the cursor, or erases the line entirely.

Format

ESC "[" Ps "K"

Ps one of the following values (the default is 0):

<u>Value</u>	<u>Action</u>
0	Erases from cursor to end of line (including character at cursor).
1	Erases from start of line to cursor.
2	Erases entire line and position cursor to column one of that line.

G.3.6 Position Cursor at Row and ColumnFunction

This sequence positions the cursor at a specified row and column.

Format:

$$\text{ESC "[" Pr ";" Pc} \left\{ \begin{array}{l} \text{"H"} \\ \text{"f"} \end{array} \right\}$$

Pr a number from 1 to 24 designating a row; the default is 1, for the upper row.

Pc a number from 1 to 40 or 1 to 80 (depending upon the width of the screen) designating a column. The default is 1 for the left column.

NOTE:

"H" and "f" have the same results.

G.3.7 Move Cursor Up, Down, Right, or LeftFunction

These sequences move the cursor in a specified direction for a specified number of columns or rows.

Format

$$\text{ESC "[" Pn} \left\{ \begin{array}{l} \text{"A"} \\ \text{"B"} \\ \text{"C"} \\ \text{"D"} \end{array} \right\}$$

Pn the number of rows or columns that the cursor will move; the default is 1.

"A" moves the cursor up.

"B" moves the cursor down.

"C" moves the cursor right.

"D" moves the cursor left.

G.3.8 Save Cursor Position / Restore Cursor to Saved Position

Function

This sequence saves the current cursor position or restores the cursor to a previously saved position.

Format

ESC "[" $\left\{ \begin{array}{l} "s" \\ "u" \end{array} \right\}$

"s" saves the current cursor position.

"u" restores the cursor to the previously saved position.

NOTE:

If you use a second Save sequence before you restore the cursor to the previously saved position, the second Save sequence overwrites the information from the first sequence.

G.3.9 Insert Line(s) at Current Line

Function

This sequence inserts a specified number of lines at the current cursor line.

Format

ESC "[" Pn ";" Ps "L"

Pn the number of lines to be inserted; the default is 1.

Ps any number. If the number is 0, the default, the sequence moves the lower part of the screen down. If the number is any nonzero number, the sequence moves the upper part of the screen up.

G.3.10 Delete Line(s) at Current LineFunction

This sequence deletes a specified number of lines beginning at the current cursor line.

Format

ESC "[" Pn ";" Ps "M"

Pn the number of lines to be deleted; the default is 1.

Ps any number. If the number is 0, the default, the sequence moves the lower part of the screen down. If the number is any nonzero number, the sequence moves the upper part of the screen up.

G.3.11 Scroll Portion of the ScreenFunction

This sequence scrolls a specified portion of the screen up or down a specified number of lines.

Format

ESC "/" Ps ";" Pe ";" Pc ";" Pd "S"

Ps the starting line number of the lines to be scrolled; must be less than Pe. There is no default.

Pe the ending line number of the lines to be scrolled. There is no default.

Pc the number of lines to scroll the specified portion of the screen. The default is 1.

Pd any number. If the number is 0, the default, the screen scrolls down. If the number is any nonzero number, the screen scrolls up.

G.3.12 Delete Characters from Current Line

Function

This sequence deletes the specified number of characters from a line starting at the current cursor position.

Format

ESC "[" Pn "P"

Pn the number of characters you want to delete.

G.3.13 Reset to Startup Configuration

Function

This sequence resets the default screen to the startup configuration.

Format

ESC "[" "z"

G.3.14 Set Palette Entry Attributes

Function:

This sequence sets the specified attributes for the selected palette entry (for the low resolution card only).

Format:

ESC "/" Ps ";" Pi ";" Pr ";" Pg ";" Pb "P"

Ps a number from 0 to 15 indicating a palette entry. The following are the palette entries as they are specified at system start-up.

<u>Entry</u>	<u>Color</u>
0	Black
1	Dark gray
2	White (light gray)
3	White (used for high intensity or bold)
4	Blue
5	Light blue
6	Brown
7	Yellow
8	Green
9	Light green
10	Magenta
11	Light magenta
12	Cyan
13	Light cyan
14	Red
15	Light red

NOTE:

For the remaining parameters, a number from 0 to 7 deselects the attribute. A number from 8 to 15 selects the attribute. There are no defaults. Table G-3 lists the possible combinations of attributes, with 0 for attribute deselected and 8 for attribute selected.

Pi	intensity
Pr	red
Pg	green
Pb	blue

Table G-3. Combinations of Palette Attributes

Resulting Color	Intensity	Red	Green	Blue
Black	0	0	0	0
Dark grey	8	0	0	0
White (light gray)	0	8	8	8
White	8	8	8	8
Blue	0	0	0	8
Light blue	8	0	0	8
Brown	0	8	8	0
Yellow	8	8	8	0
Green	0	0	8	0
Light green	8	0	8	0
Magenta	0	8	0	8
Light magenta	8	8	0	8
Cyan	0	0	8	8
Light cyan	8	0	8	8
Red	0	8	0	0
Light red	8	8	0	0

APPENDIX H

PRINTER CHARACTER SETS, CODES, AND ESCAPE SEQUENCES

H.1 INTRODUCTION

This appendix gives programmers information about the Wang PC printers. It lists the character sets, control codes, escape sequences, and return codes for each printer. For the Daisy Wheel Printer character set, refer to Section H.2. For the Matrix Printer character set, refer to Section H.3.

You can use function call 44H (I/O control), subfunction 3, to determine whether or not the Daisy Wheel Printer driver is currently loaded. If it is loaded, the message, "PC DW20 printer driver," is read into memory. The format of the I/O control command is as follows:

Byte	0FFH (the command to display the identifying message)
Dword	segment and address of the location of the message
Word	Length of memory reserved for message. The length should be at least 20 bytes. On return, this word reflects the actual length of the message, but the returned length will not be longer than the length you entered.

H.2 THE DAISY WHEEL PRINTER

This section gives the character set, control codes, escape sequences, and return codes for the Daisy Wheel Printer. For further information, refer to The Wang Professional Computer Daisy Wheel Printer Manual.

H.2.1 The Daisy Printer Character Set

The character set used for text symbols on the Wang Daisy Wheel Printer is WISCII I, with FF unused. Table H-1 gives the WISCII I character set as used by the Wang Daisy Wheel Printer.

Table H-1. WISCII I Character Set

Low order 4 Bits	High order 4 bits hex- digit	0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111															
		O	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0000	O			SPACE	O	@	P	`	p	°	Â	â	Ğ	ğ	Þ	þ	£
0001	1			!	1	A	Q	a	q	◆	À	à	U	ij	Ð	ð	ƒ
0010	2			“	2	B	R	b	r	►	Á	á	İ	ı	Ý	ý	¥
0011	3			#	3	C	S	c	s	◀	Ä	ä	Î	î	Ş	ş	¼
0100	4			\$	4	D	T	d	t	→	Å	å	Ì	ì	'	'	½
0101	5			%	5	E	U	e	u	↵	←	→	Í	í	Û	û	¾
0110	6			&	6	F	V	f	v		Ä	ä	Ï	ï	Ù	ù	^
0111	7			'	7	G	W	g	w	■	↓	†	L·L	l·l	Ú	ú	`
1000	8			(	8	H	X	h	x	!!	Æ	æ	Ñ	ñ	Ü	ü	'
1001	9			)	9	I	Y	i	y	↑	Ç	ç	Ô	ô	©	™	“
1010	A			*	:	J	Z	j	z	↓	‡	□	Ò	ò	®	☒	~
1011	B			+	;	K	[	k	{	↑	●	REG. SPACE	Ó	ó	℞	↔	.
1100	C			,	<	L	\	l		←	Ê	ê	Ö	ö	ª	º	√
1101	D			-	=	M	]	m	}	±	È	è	Õ	õ	«	»	˘
1110	E			.	>	N	↑	n	~	ı	É	é	Œ	œ	§	ß	5 UNIT SPACE
1111	F			/	?	O	_	o	€	¿	Ë	ë	Ø	ø	¶	.	DEAD KEY

H.2.2 Daisy Printer Control Codes And Escape Sequences

The following 1-byte control codes are recognized by the Daisy Wheel Printer's drivers:

<u>Code</u>	<u>Meaning</u>
07H	Sound the tone
08H	Nondestructive backspace
09H	Horizontal tab
0AH	Line feed
0BH	Vertical tab
0CH	Form feed
0DH	Carriage return
01BH	Escape sequence mode

Table H-2 lists the standard escape sequences that the Daisy Wheel Printer's driver recognizes. The escape code (ESC) is 01BH. For all the following sequences, the parameter (P) is an ASCII-encoded signed decimal number, containing the digits 0 through 9, and + and -. Unless otherwise indicated, P represents thousandths of an inch. The default for unsigned numbers is positive. All blanks (020H) and control characters are ignored in an escape sequence.

Table H-2. Daisy Printer Escape Sequences

Sequence	Meaning
ESC "[" "z"	Resets printer
ESC "[" "a"	Restores defaults
ESC "[" "b"	Clears platen without sheet feed
ESC "[" Ps "c"	Sets left margin
ESC "[" Ps "d"	Sets line feed spacing
ESC "[" Ps "e"	Sets form length
ESC "[" Ps "i"	Issues partial line feed

Table H-2. Daisy Printer Escape Sequences (continued)

Sequence	Meaning
ESC "[" Ps "v"	Selects paper tray Ps
ESC "[" Pi "o"	Selects pitch in thousandths of an inch per character. If Pi is 0, selects proportional spacing
ESC "[" Ps ";"..." Ps "p"	Sets horizontal tab positions in thousandths of an inch from left margin
ESC "[" Ps ";"..." Ps "q"	Sets vertical tab positions in thousandths of an inch from top of page
ESC "[" ";"..." Ps "h"	Sets the modes selected by the value of Ps
ESC "[" Ps ";"..." Ps "l"	Clears the modes selected by the value of Ps

Value	Mode
0	Deselect (l) /select (h) the printer
1	Auto line feed after carriage return h turns on auto line feed; l turns off auto line feed
3	Deletes current contents of buffer on receipt of external top of form command

ESC "[" Ps ";"..." Ps "m"	Sets print attribute selected by the value of Ps
ESC "[" Ps ";"..." Ps "n"	Clears print attribute selected by the value of Ps. For example, the sequence for bold print with underscore is is 1BH [1;4m.

Value	Meaning
0	All attributes
1	Overstrike
2	Subscript
3	Superscript
4	Underscore
6	Overstrike with slash
7	Double underscore

H.2.3 Daisy Printer Return Codes

Table H-3 lists the hex return codes for the Daisy Wheel Printer.

Table H-3. Daisy Wheel Printer Return Codes

Hex	Meaning
01	Cover open
02	Paper out
04	Ribbon out
08	Printer deselected
10	Top of Form pressed
20	Feeder empty
40	Paper jammed in feeder
80	Printer power off

H.3 THE MATRIX PRINTER

This section gives the character set, control codes, escape sequences, and return codes for the Matrix Printer. For further information, refer to The Wang Professional Computer Matrix Printer Manual.

H.3.1 The Matrix Printer Character Set

Table H-4 shows the Matrix Printer character set with the corresponding hex codes. Table H-5 contains the names of the characters, plus the decimal values for use in print programs.

Table H-4. Matrix Printer Characters and Hexadecimal Codes

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0		!	0	@	P	`	p	£		à	á	â	ä	å	£
1		"	1	A	Q	a	q	"		á	í	í	ÿ	ä	f
2		#	2	B	R	b	r	'		ä	í	í	ÿ	ä	¥
3		\$	3	C	S	c	s	,		ä	í	í	ÿ	ä	¥
4		%	4	D	T	d	t	'		ä	í	í	ÿ	ä	¥
5		&	5	E	U	e	u	,		ä	í	í	ÿ	ä	¥
6		'	6	F	V	f	v	,		ä	í	í	ÿ	ä	¥
7		(	7	G	W	g	w	,		ä	í	í	ÿ	ä	¥
8		)	8	H	X	h	x	,		ä	í	í	ÿ	ä	¥
9		*	9	I	Y	i	y	,		ä	í	í	ÿ	ä	¥
A		+		J	Z	j	z	,		ä	í	í	ÿ	ä	¥
B		,		K	[	k	{	,		ä	í	í	ÿ	ä	¥
C		-		L	\	l	}	,		ä	í	í	ÿ	ä	¥
D		.		M	]	m	~	,		ä	í	í	ÿ	ä	¥
E		/		N	^	n		,		ä	í	í	ÿ	ä	¥
F				O	_	o		,		ä	í	í	ÿ	ä	¥

Table H-5. Matrix Printer Character Names and Corresponding Codes

DEC	HEX	CHR	DEC	HEX	CHR	DEC	HEX	CHR	DEC	HEX	CHR	DEC	HEX	CHR	DEC	HEX	CHR
32	20	Space	64	40	@ Sign	96	60	Open Quote	160	A0	Blank	192	C0	g hacek	224	E0	thorn
33	21	!	65	41	A	97	61	a	161	A1	a grave	193	C1	ij ligature	225	E1	eth
34	22	"	66	42	B	98	62	b	162	A2	a acute	194	C2	dotless i	226	E2	y acute
35	23	#	67	43	C	99	63	c	163	A3	a umlaut	195	C3	i circumflex	227	E3	s cedilla
36	24	\$	68	44	D	100	64	d	164	A4	a tilde	196	C4	i grave	228	E4	Down Arrow
37	25	%	69	45	E	101	65	e	165	A5	E umlaut	197	C5	i acute	229	E5	u circumflex
38	26	&	70	46	F	102	66	f	166	A6	a angstrom	198	C6	i umlaut	230	E6	u grave
39	27	'	71	47	G	103	67	g	167	A7	plus/minus	199	C7	ll ligature	231	E7	u acute
40	28	(	72	48	H	104	68	h	168	A8	ae ligature	200	C8	n tilde	232	E8	u umlaut
41	29	)	73	49	I	105	69	i	169	A9	c cedilla	201	C9	o circumflex	233	E9	Left Arrow
42	2A	"	74	4A	J	106	6A	j	170	AA	Inv Exclaim.	202	CA	o grave	234	EA	Monetary
43	2B	+	75	4B	K	107	6B	k	171	AB	Inv Question	203	CB	o acute	235	EB	Right Arrow
44	2C	.	76	4C	L	108	6C	l	172	AC	e circumflex	204	CC	o umlaut	236	EC	o superior
45	2D	-	77	4D	M	109	6D	m	173	AD	e grave	205	CD	o tilde	237	ED	a circumflex
46	2E	.	78	4E	N	110	6E	n	174	AE	e acute	206	CE	oe ligature	238	EE	Beta
47	2F	Slash	79	4F	O	111	6F	o	175	AF	e umlaut	207	CF	slashed o	239	EF	dot
48	30	Zero	80	50	P	112	70	p	176	BD	G Hacek	208	DD	THORN	240	FD	Pound Sterling
49	31	One	81	51	Q	113	71	q	177	B1	IJ Ligature	209	D1	ETH	241	F1	Florin
50	32	Two	82	52	R	114	72	r	178	B2	Dotted I	210	D2	Y Acute	242	F2	YEN
51	33	Three	83	53	S	115	73	s	179	B3	I Circumflex	211	D3	S Cedilla	243	F3	1/4
52	34	Four	84	54	T	116	74	t	180	B4	I Grave	212	D4	A Acute	244	F4	1/2
53	35	Five	85	55	U	117	75	u	181	B5	I Acute	213	D5	U Circumflex	245	F5	3/4
54	36	Six	86	56	V	118	76	v	182	B6	I Umlaut	214	D6	U Grave	246	F6	A Circumflex
55	37	Seven	87	57	W	119	77	w	183	B7	LL Ligature	215	D7	U Acute	247	F7	A Grave
56	38	Eight	88	58	X	120	78	x	184	B8	W Tilde	216	D8	U Umlaut	248	F8	A Umlaut
57	39	Nine	89	59	Y	121	79	y	185	B9	O Circumflex	217	D9	A Tilde	249	F9	A Angstrom
58	3A	Colon	90	5A	Z	122	7A	z	186	BA	O Grave	218	DA	degree	250	FA	AE Ligature
59	3B	Semicolow	91	5B	Left Bracket	123	7B	Left Brace	187	BB	O Acute	219	DB	cent	251	FB	C Cedilla
60	3C	Less Than	92	5C	Back Slash	124	7C	Vertical Bar	188	BC	O Umlaut	220	DC	a superior	252	FC	E Circumflex
61	3D	Equal Sign	93	5D	Right Bracket	125	7D	Right Brace	189	BD	O Tilde	221	DD	bullet	253	FD	E Grave
62	3E	Greater Than	94	5E	Up Arrow	126	7E	Approximate	190	BE	OE Ligature	222	DE	section	254	FE	E Acute
63	3F	Question	95	5F	Underscore	127	7F	DELETE	191	BF	Slashed O	223	DF	Paragraph	255	FF	DELETE

DEC=DECIMAL
 HEX=HEXADECIMAL
 CHR=CHARACTER

H.3.2 Matrix Printer Control Codes and Escape Sequences

The control codes for the Matrix Printer are the same as those for the Daisy Printer (refer to Section H.2.2). Table H-6 lists the escape sequences for the Matrix Printer. The escape code (ESC) is 01BH. For all the following sequences, the parameter (P) is an ASCII-encoded signed decimal number, containing the digits 0 through 9, and + and -. Unless otherwise indicated, P represents thousandths of an inch. The default for unsigned numbers is positive. All blanks (020H) and control characters are ignored in an escape sequence.

Table H-6. Matrix Printer Escape Sequences

Sequence	Meaning
ESC "[" "z"	Restores defaults and flushes buffer
ESC "[" "a"	Restores defaults and flushes buffer
ESC "[" Ps "c"	Set left margin in number of 10-pitch characters/inch
ESC "[" Ps "d"	Set line feed spacing
ESC "[" Ps "e"	Set form length. Ps is rounded to an even inch increment.
ESC "[" Ps "i"	Issue partial line feed
ESC "[" Pi "o"	Select pitch. Pi is 16.5 pitch, 10 pitch or 5 pitch in thousands of an inch per character.
ESC "[" Ps ";" ... ";" Ps "h"	Set suppress auto line feed after carriage return. Ps = 1.
ESC "[" Ps ";" ... ";" Ps "l"	Clear suppress auto line feed after carriage return. Ps = 1.
ESC "[" Ps ";" ... ";" Ps "m"	Sets print attribute selected by the value of Ps
ESC "[" Ps ";" ... ";" Ps "n"	Clears print attribute selected by the value of Ps

Table H-6. Matrix Printer Escape Sequences (continued)

Value	Meaning
0	All attributes (except that "m" does not set subscripts and superscripts)
1	Overstrike
2	Subscript
3	Superscript
4	Underscore
6	Autoslash
7	Double underscore

H.2.3 Matrix Printer Return Codes

Table H-7 lists the hex return codes for the Matrix Printer.

Table H-7. Matrix Printer Return Codes

Hex	Meaning
02	Paper out
04	Fault
08	Printer deselected

APPENDIX I IBM PC TO WANG PC CODE CONVERSION

I.1 INTRODUCTION

You can easily convert programs in BASIC from the IBM Personal Computer to the Wang Professional Computer. The following sections give guidelines for the conversion.

I.2 KEYBOARD, PRINTER, AND SCREEN PROTOCOLS

No syntax modifications are required to run programs written in IBM BASIC on the Wang PC. The Microsoft BASIC language interpreter is functionally compatible with the IBM PC Advanced BASIC language interpreter. However, differences do exist in the keyboard and display protocols. When you are converting existing programs or developing new programs, we suggest you use variables to represent key and display values in the code.

For example, suppose the program is to display a vertical bar on each of these machines; in the program, you could define a variable VERTBAR\$. On the IBM PC, you would let VERTBAR\$ equal CHR\$(179). However, on the Wang PC, you would set it equal to CHR\$(134); then, to display the vertical bar you simply code PRINT VERTBAR\$;. This method eliminates the need to change every statement in the software responsible for the display of text.

This technique can also be applied to the keyboard. Suppose the program is to test for the pressing of the PF01 key (when disabled as a soft key). In the program, define a variable PF01KEY\$. On the IBM PC you set PF01KEY\$ equal to CHR\$(059), but on the Wang PC it would equal a CHR\$(128). Then, the variable receiving the key value from the INKEY\$ statement could be compared to the PF01KEY\$ variable, instead of comparing it to a hard coded keystroke value particular to one machine.

To take this technique a step further, a standard header could be built for each machine including all of the key and display values that exist on a given machine. This standard header could then be merged into each program developed to improve readability and transportability of your software.

I.2.1 Keyboard Values

Key values on the IBM PC are equal to those on the Wang PC. This comparison is limited to the keys available on both machines as there are some keys available on the Wang PC keyboard that are not available on the IBM PC keyboard, and vice versa. Refer to the discussion of the keyboard in The Wang Professional Computer Introductory Guide.

For certain keys or key combinations that cannot be stored as a standard WISCII code, the BASIC INKEY\$ statement returns an extended code. The value CHR\$(031) is returned to indicate that an extended code follows. If the first character received is a CHR\$(031), then a second code is pending in the keystroke buffer. Execution of another INKEY\$ statement returns this second code immediately. The following is a list of the WISCII codes (in decimal) for this second character, and the associated keys.

<u>Second Code</u>	<u>Meaning</u>
067	Shift/Cancel
128-143	PF01 to PF16 (Indent through Goto)
144-159	PF17 to PF32 (Shift/Indent through Shift/Goto)
192	Cursor Up
193	Cursor Right
194	Cursor Down
195	Cursor Left
196	Home
197	Execute
198	Insert
199	Delete
200	Prev
201	Next
203	Erase
205	Backtab
208	Shift/Cursor Up
209	Shift/Cursor Right
210	Shift/Cursor Down
211	Shift/Cursor Left
212	Shift/Home
213	Shift/Execute
214	Shift/Insert
215	Shift/Delete
216	Shift/Prev
217	Shift/Next

<u>Second Code</u>	<u>Meaning</u>
219	Shift/Erase
220	Shift/Tab
221	Shift/Backtab
224	Cancel
225	Help
226	GL(glossary)
227	Print
241	Shift/Help
242	Shift/GL(glossary)
243	Shift/Print

I.3 TRANSPORTING TEXT FILES FROM THE IBM PC TO THE WANG PC

Media compatibility exists between the Wang and IBM PCs. The Wang PC can read SSDD or DSDD 5.25 inch diskettes formatted on the IBM PC. Because both systems use virtually the same operating system, the diskette formats are fully compatible. Therefore, the Wang PC can read all source programs and data files created by the IBM PC-DOS, without any difficulty. You should save BASIC source programs that are to be transported in ASCII format. The Wang PC BASIC Compiler cannot read source files in condensed format.

Another method of transporting source programs and data files between the two machines is available if both systems are equipped with asynchronous communications hardware and software. Both Wang and IBM offer asynchronous communications capabilities. However, this is an option on the IBM PC. On the Wang PC, the RS-232-C communications port is standard.

Use the IBM Asynchronous Communications Support Software to transfer text files to the Wang PC. The following are some constraints imposed by the protocol used in the IBM Asynchronous Communications Support Software:

- The files must be textual, and cannot contain more than 254 characters per line.
- BASIC source programs must be in ASCII format.
- Each line in the file must end with CR/LF.
- Each file must terminate with a CONTROL + Z.
- A full duplex protocol is employed [using XON (CONTROL + Q) and XOFF (CONTROL + S)] to prevent communication buffer overflow during data reception.

The following sample BASIC program illustrates how you can transport text files from the IBM PC to the Wang PC using asynchronous communications. The program assumes a 300 bits per second (bps) line, even parity, seven data bits, and an input buffer of 256 bytes. Characters received are echoed on the video monitor to verify receipt of valid characters.


```
10 CLS:WIDTH 80: KEY OFF: DEFINT A-Z
20 LOCATE 10,10: INPUT "(T)transmit or (R)receive "; XMIT$
30 LOCATE 12,10: INPUT "ASCII Filename "; NAM$
40 IF XMIT$ = "t" OR XMIT$ = "T" GOTO 150
50 REM LINES 60-130 RECEIVE FILE FROM COM PORT
60 OPEN NAM$ FOR OUTPUT AS #1
70 OPEN "com1:300,E,7,1" AS #2
80 IF LOC(2)=0 THEN GOTO 80
90 WHILE NOT EOF(2)
100 LINE INPUT #2,A$
110 PRINT #1, A$
120 WEND
130 CLOSE: GOTO 220
140 REM LINES 150-210 TRANSMIT FILE TO COM PORT
150 OPEN NAM$ FOR INPUT AS #1
160 OPEN "com1:300,E,7,1" AS #2
170 WHILE NOT EOF(1)
180 LINE INPUT #1, A$
190 PRINT #2, A$
200 WEND
210 CLOSE
220 LOCATE 20,30: PRINT "Transfer Complete"
230 END
```

APPENDIX J

CONTENTS OF THE WANG PC SYSTEM SOFTWARE DISKETTE

This appendix lists the contents of the Wang PC System Software Diskette shipped with each system by Wang Laboratories, Inc., January 24, 1983. Track 0, sector 1 of the diskette contains the start-up sector. Two copies of the FAT and one of the disk directory follow the start-up sector (refer to Section A.4 for a discussion of the disk directory and the FATs). The following are the file names on the System Software diskette with the corresponding version numbers (N/A means not applicable), and descriptions of their functions.

<u>File Name</u>	<u>Version</u>	<u>Description</u>
BIOS.SYS	1	Basic I/O System
MSDOS.SYS	2	Disk Operating System
CONFIG.SYS	N/A	Standard Configuration File (for MENU.COM)
CONFIG.BAK	N/A	Modified Configuration File (for COMMAND.COM)
DW20DRV.R.COM	1	DW20 Printer Driver
CONFGEDT.COM	1	Configuration File Editor
COMMAND.COM	2	Command Processor
CHKDSK.COM	2.1	Check Disk Utility
FORMAT.COM	2	Disk Initialization Utility
FILCOM.COM	1	Compare Files Utility
PRINT.COM	1	Print Spooler Utility
UTILITY.COM	1	System Utilities Program Loader
MENU.COM	1	Menu Display Program
MENUDRV.R.COM	1	Loader for MENU.COM and Files Called from Menus
MENUICMP.COM	1	Menu Editor
WPCONV.COM	1	Text File to WP Document Conversion Utility
CONVRSRC.WPS	1	WP Document Conversion Messages
KEYBOARD.WPS	1	Keyboard Translation Table
BASIC.EXE	1	GW-Basic Interpreter
UTILITY.MSG	1	System Utilities Messages
MENU.MSG	1	Date and Time Screen and Menu Messages
UTILITY.DAT	1	Utility Menu Data for MENU.COM
MENU.DAT	1	Main Menu Data for MENU.COM
APPMENU.DAT	1	Applications Menu Data for MENU.COM
PRGDEV.MEN.DAT	1	Program Development Menu Data for MENU.COM
TCMENU.DAT	1	Communications Menu Data for MENU.COM
APPMENU.HLP	1	Applications Menu Help Screens

<u>File Name</u>	<u>Version</u>	<u>Description</u>
MENU.HLP	1	Main Menu Help Screens
TCMENU.HLP	1	Communications Menu Help Screens
PRGDVMEN.HLP	1	Program Development Menu Help Screens
UTILITY.HLP	1	System Utilities Menu Help Screens
PRNDW20.BCF	1	DW20 Batch Command File for CONFGEDT.COM
DAISY06	1	Character Set Table for Daisy Printer

APPENDIX K
ABBREVIATIONS GLOSSARY

<u>Abbreviation</u>	<u>Meaning</u>
AF	Auxiliary Carry Flag
ANSI	American National Standards Institute
ASCII	American National Standard Code for Information Interchange
ASCIZ	ASCII string with a hex zero used as an end-of-string indicator
AUX	Auxiliary Device
BIOS	Basic I/O System
BPB	BIOS Parameter Block
CF	Carry Flag
CHR	Character
CON	Console Device
CRC	Cyclic Redundancy Check
CS	Code Segment (register)
DEC	Decimal
DF	Direction Flag
DMA	Direct Memory Access
DOS	Disk Operating System
DPB	Device Parameter Block
DS	Data Segment (register)
DTA	Disk Transfer Address
DWORD	Double Word
ES	Extra Segment (register)
ESC	Escape
FAT	File Allocation Table
FCB	File Control Block
H (suffix)	Hexadecimal
HEX	Hexadecimal
IF	Interrupt Flag
Int	Software Interrupt
IOCTL	I/O Control (DOS function call)
IP	Instruction Pointer (register)
IRET	Return from Interrupt (Assembler instruction)
LED	Light-emitting Diode
OF	Overflow Flag
PF	Parity Flag
PRN	Printer Device
PROM	Programmable Read-only Memory
PSH	Program Segment Header
QID	Queue Identification Number

<u>Abbreviation</u>	<u>Meaning</u>
SF	Sign Flag
SP	Stack Pointer (register)
SS	Stack Segment (register)
STI	Standard Input Device
STO	Standard Ouptut Device
TF	Trap Flag
WISCII	Wang International Standard Code for Information Interchange
ZF	Zero Flag

INDEX

A

A (Assemble) command, 4-8 to 4-10
Abbreviations glossary, K-1, K-2
Absolute disk read interrupt,
 C-4
Absolute disk write interrupt,
 C-5
Address parameter, 4-5
Allocate DMA channel BIOS
 function, D-2
Allocate memory function call,
 C-20
Ampersand (&) command character,
 3-11
ASCII character string, C-11
ASCIZ character string, C-6
Asterisk (*) command character,
 3-10
Auxiliary input function call,
 C-8
Auxiliary output function call,
 C-9
AVAILDEV command, A-9

B

BACKSPACE command, 1-5
BIOS, D-1 to D-6
 disk drive control blocks, D-6
 return codes, D-3
 screen information block, D-5,
 D-6
 software interface, D-1 to D-3
 system configuration table, D-4
 to D-6
 trappable events, D-3, D-4
BIOS functions, D-1, D-2
 Allocate DMA channel, D-2
 Clear event trap, D-1
 Free DMA channel, D-2
 Send bytes to keyboard, D-2
 Send character to screen, D-2
 Set/Clear CONTROL +S/Q, D-2
 Set DMA channel, D-2
 Set event trap, D-1

 Set pointer to configuration
 table, D-1
 Set pointer to system name,
 D-1
BREAK command, A-9
Buffered keyboard input function
 call, C-9
BUFFERS command, A-9
Byte parameter, 4-5
Build BPB driver function, E-12,
 E-13

C

C (Compare) command, 4-11
Center mode, 1-1, 1-5
CENTER MODE command, 1-5
Change current directory function
 call, C-16
Change incompatible configuration
 parameters function call,
 C-15
Change write protection function
 call, C-18
Character input with buffer
 flush function call, C-9
Character sets
 daisy wheel printer, H-1, H-2
 matrix printer, H-5 to H-7
Check keyboard status function
 call, C-9
Class, 2-3 to 2-6
Clear Current Line escape
 sequence, G-5
Clear event trap BIOS function,
 D-1
Clear Screen escape sequence, G-5
Close file function call
 (pre-2.0), C-10
Close a file function call (2.0),
 C-17
Code conversion, IBM PC to Wang
 PC, I-1 to I-4
.COM files, A-15
Command characters
 library manager
 Linker, 2-12, 2-13

INDEX (continued)

COMMAND.COM, A-7, A-10, A-11
 CONFIG.EDT, 1-4
 CONFIG.SYS, A-9, A-10
 Console input without echo
 function call, C-8
 Control codes
 daisy wheel printer, H-3
 matrix printer, H-8
 screen, G-1
 CONTROL + characters, F-7, F-8
 COPY LINES command, 1-6
 CP/M Calling conventions, C-24
 Create file function call
 (pre-2.0), C-11
 Create a file function call
 (2.0), C-16
 Create new program segment
 function call, C-12
 Current directory function call,
 C-20
 Current disk function call, C-11
 CURSOR EAST, CURSOR WEST command,
 1-6
 CURSOR NORTH, CURSOR SOUTH
 command, 1-6

D

D (Dump) command, 4-12, 4-13
 Daisy wheel printer, H-1 to H-5
 character set, H-1, H-2
 control codes, H-3
 escape sequences, H-3, H-4
 return codes, H-5
 Dead keys, F-9
 Debugger, 4-1 to 4-39
 initialization, 4-2, 4-3
 invoking, 4-1, 4-2
 messages, B-6
 using, 4-3, 4-4
 Debugger commands, 4-7 to 4-39
 A (Assemble), 4-8 to 4-10
 C (Compare), 4-11
 D (Dump), 4-12, 4-13
 E (Enter), 4-14, 4-15
 F (Fill), 4-16
 G (Go), 4-17, 4-18
 H (Hexarithmic), 4-19
 I (Input), 4-20
 L (Load), 4-21, 4-22

M (Move), 4-23,
 N (Name), 4-24 to 4-26
 O (Output), 4-27
 Q (Quit), 4-28
 R (Register), 4-29 to 4-31
 S (Search), 4-32
 T (Trace), 4-33, 4-34
 U (Unassemble), 4-35, 4-36
 W (Write), 4-37, 4-38
 X (Change I/O Device), 4-39
 Debugger Parameters, 4-5
 <address>, 4-5
 <byte>, 4-5
 <device>, 4-5
 <drive>, 4-5
 <filespec>, 4-5
 <list>, 4-6
 <range>, 4-6
 <register-name>, 4-6
 <sector>, 4-6
 <string>, 4-6
 <value>, 4-7
 Delete a directory entry function
 call, C-17
 DELETE CHARACTER command, 1-6
 Delete characters from line
 escape sequence, G-9
 Delete file function call, C-11
 DELETE LINE command, 1-6
 Delete lines escape sequence, G-8
 DELETE MANY LINES command, 1-7
 DEVICE command, A-9
 Device drivers, E-1 to E-15
 data block formats, E-11 to
 E-14
 device interface, E-1 to E-2
 format of, E-2 to E-4
 functions of, E-7 to E-9
 installation of, E-4 to E-7,
 E-14, E-15
 static request header, E-9,
 E-10
 Device driver functions, E-7 to
 E-9
 Build BPB, E-12, E-13
 Flush calls, E-14
 Init, E-14
 Media check, E-12
 Nondestructive read no wait,
 E-11
 Status calls, E-14

INDEX (continued)

- Device parameter, 4-5
- Direct console input function call, C-9
- Direct console I/O function call, C-9
- Disk directory, A-25, A-26
- Disk error handling, A-17
- Disk reset function call, C-9
- Disk space allocation, A-25
- Disk Transfer Address (DTA), A-20
- Diskette layouts, A-21 to A-25
- Disk I/O requests (BIOS), DOS
 - Command Processor, A-7, A-10, A-11
 - contents of DOS disk, A-7, A-8
 - disk error handling, A-17
 - initialization, A-8 to A-10
 - memory usage, A-11, A-13
 - program initialization, A-13 to A-17
 - technical information, A-7 to A-17
- Drive designation parameter, 2-6
- Drive parameter, 4-5
- DSALLOCATE switch, 2-13, 2-14
- Duplicate a file handle function call, C-20

E

- E (Enter) command, 4-14, 4-15
- Editing modes, 1-1, 1-5,
- Editor
 - buffer size, 1-2
 - commands, 1-5 to 1-12
 - configuration parameters, 1-4
 - help screens, 1-2
 - invoking, 1-2, 1-3
 - line numbers, 1-2
 - window, 1-1
- Editor commands, 1-5 to 1-12
 - BACKSPACE, 1-5
 - CENTER MODE, 1-5
 - COPY LINES, 1-6
 - CURSOR EAST, CURSOR WEST, 1-6
 - CURSOR NORTH, CURSOR SOUTH, 1-6
 - DELETE CHARACTER, 1-6
 - DELETE LINE, 1-6
 - DELETE MANY LINES, 1-7

- END SESSION AND SAVE CHANGES, 1-7
- ERASE LINE, 1-7
- ERASE TO END, 1-7
- GO TO FIRST LINE, 1-8
- GO TO LAST LINE, 1-8
- GO TO LINE, 1-8
- INSERT LINE, 1-8
- INSERT MODE, 1-8
- LOAD PARTIAL FILE, 1-9
- LOAD TEXT, 1-9
- MOVE LINES, 1-9
- NEXT SCREEN, 1-10
- PREVIOUS SCREEN, 1-10
- PRINT LINES, 1-10
- REPLACE, 1-10
- SAVE PARTIAL FILE, 1-11
- SEARCH, 1-11
- SET TABS, 1-11
- SET WIDTH, 1-12
- SHIFT + CURSOR EAST, SHIFT + CURSOR WEST, 1-12
- SHIFT + NEXT, SHIFT + PREV, 1-12
- TAB, 1-12
- END SESSION AND SAVE CHANGES command, 1-7
- ERASE LINE command, 1-7
- ERASE TO END command, 1-7
- Error messages, see messages
- Escape sequences, printers
 - daisy wheel printer, H-3, H-4
 - matrix printer, H-8
- Escape sequences, screen, E-1 to E-11
- .EXE files, A-15 to A-17
- Execute a program function call, C-21, C-22
- Extended file control block, A-20
- Exit from a process function call, C-22

F

- F (Fill) command, 4-16
- FAT ID byte, A-21 to A-24
- Fatal error abort vector interrupt, C-2, C-3
- File Allocation Table (FAT), A-27
- File Control Block, A-18 to A-20 extended, A-20
- File handle, C-11

INDEX (continued)

File management, A-17 to A-27
File size function call, C-12
File times function call, C-23
Filename parameter, 2-6
Extension parameter, 2-6
Filespec parameter, 2-6, 4-5
FILES command, A-9
Find first function call, C-22
 C-23
Find next function call, C-23
Flag codes, 4-30
Flags
 8086, A-3, A-4
 Debugger, 4-30
Floating cursor mode, 1-1, 1-5
Flush calls driver function, E-14
Force a duplicate of a handle
 function call, C-20
Free allocated memory function
 call, C-21
Free DMA channel BIOS function,
 D-2
Function calls, C-5 to C-23
 2.0 function calls, C-14 to
 C-23
 pre-2.0 function calls, C-8 to
 C-14
 Xenix compatible function
 calls, C-16 to C-23
Function request interrupt, C-1

G

G (Go) command, 4-17, 4-18
Get current DTA function call,
 C-14
Get date function call, C-14
Get disk free space function
 call, C-15
Get DOS critical section flag
 function call, C-15
Get DOS version number function
 call, C-14
Get interrupt vector function
 call, C-15
Get time function call, C-14
Get verify flag function call,
 C-23
GO TO FIRST LINE command, 1-8
GO TO LAST LINE command, 1-8
GO TO LINE command, 1-8
group, 2-3 to 2-6

H

H (Hexarithmic) command, 4-19
Help screens, 1-2
HIGH switch, 2-14

I

I (Input) command, 4-20
IBM PC to Wang PC code
 conversion, I-1 to I-4
 keyboard, printer, and
 screen protocols, I-1 to
 I-3
 keystroke values, I-2, I-3
 transporting text files, I-3,
 I-4
Init driver function, E-14
INSERT LINE command, 1-8
Insert Lines escape sequence, G-7
INSERT MODE command, 1-8
Instruction Pointer
International information
 function call, C-16
Interrupt vectors, A-12, A-13
Interrupts (DOS), C-1 to C-5
 Absolute disk read, C-4
 Absolute disk write, C-5
 Fatal error abort vector, C-2,
 C-3
 Function request, C-1
 Program terminate, C-1
 SHIFT + CANCEL exit address,
 C-2
 Terminate address, C-2
 Terminate but stay resident,
 C-5
Invisible dead keys, F-9
I/O control for devices function
 call, C-18 to C-20

K

Keep process function call, C-14
Keyboard input function call, C-8

L

L (Load) command, 4-21, 4-22
LED's (light emitting diodes),
 G-4

INDEX (continued)

Libraries prompt, 2-11, 2-12
 Library file prompt, 3-8
 Library Manager, 3-1 to 3-12
 command characters, 3-9 to 3-12
 command prompts, 3-7 to 3-9
 features, 3-1
 messages, B-4 to B-6
 operation, 3-2 to 3-4
 running, 3-5 to 3-7
 system requirements, 3-4
 Line numbers, 1-2
 LINENUMBERS switch, 2-14
 Linker, 2-1 to 2-15
 combine types, 2-4
 command characters 2-12, 2-13
 command prompts 2-10 to 2-12
 files, 2-6, 2-7
 messages, B-1 to B-3
 operation, 2-1, 2-2, 2-3 to 2-6
 running, 2-8, 2-9
 switches, 2-13 to 2-15
 system requirements, 2-2
 Linker switches, 2-13 to 2-15
 -DALLOCATE, 2-13
 -HIGH, 2-14
 -LINENUMBERS, 2-14
 -MAP, 2-14
 -PAUSE 2-14, 2-15
 -STACK, 2-15
 List file Library Manager prompt,
 3-9
 List file Linker prompt, 2-11
 list parameter, 4-6
 LOAD PARTIAL FILE command, 1-9
 LOAD TEXT command, 1-9
 Logical keyboard protocol, F-1 to
 F-9
 control characters, F-7, F-8
 dead keys, F-9
 simple characters, F-1 to F-8
 special characters, F-8, F-9
 WISCII I character set, F-2 to
 F-7

M

M (Move) command, 4-23
 Make directory function call,
 C-16
 MAP switch, 2-14

Matrix printer, H-5 to H-9
 character set, H-5 to H-7
 control codes, H-8
 escape sequences, H-8
 return codes, H-9
 Media check driver function, E-12
 Memory usage, A-11 to A-13
 Messages
 Debugger, B-6
 Library Manager, B-5, B-6
 Linker, B-1 to B-4
 Minus sign (-) command character,
 3-10
 Modify allocated block function
 call, C-21
 Module, 3-1
 Move Cursor escape sequence, G-6
 Move file read/write pointer
 function call, C-17
 MOVE LINES command, 1-9

N

N (Name) command, 4-24 to 4-26
 NEXT SCREEN command, 1-10
 Nondestructive read no wait
 driver function, E-11

O

O (Output) command, 4-27
 Object file, 3-1
 Object module, 3-1
 Object modules prompt, 2-10
 Open file function call (pre-
 2.0), C-10
 Open file function call (2.0),
 C-16
 Operation prompt, 3-8, 3-9

P

Palette entries, G-10
 Palette entry attributes, G-10,
 G-11
 Parse file name function call,
 C-13
 PAUSE switch, 2-14, 2-15
 Plus sign (+) command character,
 3-10

INDEX (continued)

Position cursor escape sequence,
G-6
Pre-2.0 function calls, C-8 to
C-14
PREVIOUS SCREEN command, 1-10
PRINT LINES command, 1-10
Print string function call, C-9
Printer output function call, C-9
Printers, H-1 to H-9
character sets, H-1, H-2, H-5
to H-7
control codes, H-5, H-9
daisy wheel printer, H-1 to H-5
escape sequences, H-3 H-4, H-8
matrix printer, H-5 to H-9
return codes, H-5, H-9
Program Development menu, 1-3
program initialization, A-13 to
A-17
Program terminate function call,
C-8
Program terminate interrupt, C-1

Q

Q (Quit) command, 4-28

R

R (Register) command,
4-29 to 4-30
Random block read function call,
C-12
Random block write function call,
C-13
Random read function call, C-12
Random write function call, C-12
Range parameter, 4-6
Read a file or device function
call, C-17
Register-name parameter, 4-6
Remove directory function call,
C-16
Rename function call (2.0), C-23
Rename file function call
(pre-2.0), C-11
REPLACE command, 1-10, 1-11
Reset Startup Configuration
escape sequence, G-9
Restore Cursor Position escape
sequence, G-7

Retrieve the return code of a
child process, C-20
Run file prompt, 2-10, 2-11

S

S (Search) command, 4-32
SAVE PARTIAL FILE, 1-11
Save/Restore Cursor Position
escape sequence, G-7
Screen output function call, C-8
Screen manipulation, G-1 to G-11
control codes, G-1
escape sequences, G-1 to G-11
Scroll portion of screen
escape sequence, G-8
SEARCH command, 1-11
Search for the first entry
function call, C-10
Search for the next entry
function call, C-10
Sector parameter, 4-6
Segment, 2-3 to 2-6
Segment registers
8086, A-2, A-3
Debugger, 4-2
Select disk function call, C-9
Semicolon (;) command character,
3-11
Send bytes to keyboard BIOS
function, D-2
Send character to screen BIOS
function, D-2
Sequential read function call,
C-11
Sequential write function call,
C-11
Set/Clear CONTROL + S/CONTROL + Q
BIOS function, D-2
Set/Clear LED codes escape
sequence, G-4
Set/Clear Modes escape sequence,
G-4
Set CONTROL + C trapping function
call, C-15
Set date function call, C-14
Set disk transfer address function
call, C-11
Set DMA channel BIOS function,
D-2
Set event trap BIOS function, D-1

INDEX (continued)

Set Graphics Rendition escape sequence, G-2, G-3
Set interrupt vector function call, C-12
Set Palette Entry Attributes escape sequence, G-9 to G-11
Set pointer to configuration table BIOS function, D-1
Set pointer to system name BIOS function, D-1
Set random record field function call, C-12
Set/reset verify flag function call, C-14
SET TABS command, 1-11
Set time function call, C-14
SET WIDTH command, 1-12
SHELL command, A-10
SHIFT + CANCEL
 command character, 3-12
 exit address, C-1, C-2
 interrupt, C-2
SHIFT + CURSOR EAST, SHIFT + CURSOR WEST commands, 1-12
SHIFT + NEXT, SHIFT + PREV commands, 1-12
Simple characters, F-1 to F-8
Software/BIOS interface, D-1 to D-3
Special characters, F-8, F-9
Stack pointer, 4-2
STACK switch, 2-15
Startup sector, A-22 to A-24
Status calls driver function, E-14
String parameter, 4-6
Switch Default Screen escape sequence, G-3
System Configuration Table, D-4 to D-6
 Screen Information Block, D-5 to D-6
 Disk Drive Control Blocks, D-6
System software diskette, J-1, J-3

T

T (Trace) command, 4-33, 4-34
TAB command, 1-12
Terminate address interrupt, C-2

Terminate but stay resident interrupt, C-5
Transporting text files, IBM PC to Wang PC, I-3, I-4

U

U (Unassemble) command, 4-35, 4-36

V

Value parameter, 4-7
Visible dead keys, F-9

W

Window, 1-1
W (Write) command, 4-37, 4-38
Write a file function call, C-17
WSCII I character set, F-1 to F-8, H-2

X

X (Change I/O Device) command, 4-39
XENIX compatible function calls, C-16 to C-23
XENIX operating system, C-6

2.0 function calls, C-14 to C-23
8086 Processor, A-1
 address generation, A-6
 architecture, A-1
 flags, A-1, A-3, A-4
 memory organization, A-5 to A-7
 registers, A-1, A-2
 segmentation, A-2, A-3
 storage organization, A-5
 & command character, 3-11, 3-12
 * command character, 3-10
 - command character, 3-10
 + command character, 3-10

Help Us Help You . . .

We've worked hard to make this document useful, readable, and technically accurate. Did we succeed? Only you can tell us! Your comments and suggestions will help us improve our technical communications. Please take a few minutes to let us know how you feel.

How did you receive this publication?

- | | |
|--|--------------------------------------|
| ☐ Support or Sales Rep | ☐ Don't know |
| ☐ Wang Supplies Division | ☐ Other _____ |
| ☐ From another user | _____ |
| ☐ Enclosed with equipment | _____ |

How did you use this Publication?

- | | |
|--|--|
| ☐ Introduction to the subject | ☐ Aid to advanced knowledge |
| ☐ Classroom text (student) | ☐ Guide to operating instructions |
| ☐ Classroom text (teacher) | ☐ As a reference manual |
| ☐ Self-study text | ☐ Other _____ |

Please rate the quality of this publication in each of the following areas.

	EXCELLENT	GOOD	FAIR	POOR	VERY POOR
Technical Accuracy — Does the system work the way the manual says it does?	☐	☐	☐	☐	☐
Readability — Is the manual easy to read and understand?	☐	☐	☐	☐	☐
Clarity — Are the instructions easy to follow?	☐	☐	☐	☐	☐
Examples — Were they helpful, realistic? Were there enough of them?	☐	☐	☐	☐	☐
Organization — Was it logical? Was it easy to find what you needed to know?	☐	☐	☐	☐	☐
Illustrations — Were they clear and useful?	☐	☐	☐	☐	☐
Physical Attractiveness — What did you think of the printing, binding, etc?	☐	☐	☐	☐	☐

Were there any terms or concepts that were not defined properly? ☐ Y ☐ N If so, what were they? _____

After reading this document do you feel that you will be able to operate the equipment/software? ☐ Yes ☐ No
☐ Yes, with practice

What errors or faults did you find in the manual? (Please include page numbers) _____

Do you have any other comments or suggestions? _____

Name _____ **Street** _____

Title _____ **City** _____

Dept/Mail Stop _____ **State/Country** _____

Company _____ **Zip Code** _____ **Telephone** _____

Thank you for your help.

WANG

Fold



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS

PERMIT NO. 16

LOWELL, MA

POSTAGE WILL BE PAID BY ADDRESSEE

**WANG LABORATORIES, INC.
CHARLES T. PEERS, JR., MAIL STOP 1226A
ONE INDUSTRIAL AVENUE
LOWELL, MASSACHUSETTS 01851**



Cut along dotted line.

Fold

WANG

The completed order form should be mailed to:

WANG LABORATORIES, INC.
Supplies Division
51 Middlesex St.
No. Chelmsford MA 01863

To Order by Phone, Call:

(800) 225-0234

From Mass., Hawaii, and Alaska

(617) 256-1400**TELEX 951-743****Order Form for Wang Manuals and Documentation**

① Customer Number (If Known) _____				
② Bill To: _____			Ship To: _____	
_____			_____	
_____			_____	
_____			_____	
③ Customer Contact: _____			④ Date _____ Purchase Order Number _____	
() () _____ Phone Name			_____	
⑤ Taxable ☐ ⑥ Tax Exempt Number ☐ ⑦ Credit This Order to _____ Yes ☐ A Wang Salesperson _____ No ☐ Please Complete Salesperson's Name _____ Employee No. _____ RDB No. _____				
⑧ Document Number	Description	Quantity	⑨ Unit Price	Total Price
⑩ _____ Authorized Signature			Date _____	
☐ Check this box if you would like a free copy of the Corporate Publications Literature Catalog (700-5294)			Sub Total	
			Less Any Applicable Discount	
			Sub Total	
			Local/State Tax	
			Total Amount	

Ordering Instructions

1. If you have purchased supplies from Wang before, and know your Customer Number, please write it here.
2. Provide appropriate Billing Address and Shipping Address.
3. Please provide a phone number and name, should it be necessary for WANG to contact you about your order.
4. Your purchase order number and date.
5. Show whether order is taxable or not.
6. If tax exempt, please provide your exemption number.
7. If you wish credit for this order to be given to a WANG salesperson, please complete.
8. Show part numbers, description and quantity for each product ordered.
9. Pricing extensions and totaling can be completed at your option; Wang will refigure these prices and add freight on your invoice.
10. Signature of authorized buyer and date.

Wang Supplies Division Terms and Conditions

1. **TAXES** — Prices are exclusive of all sales, use, and like taxes.
2. **DELIVERY** — Delivery will be F.O.B. Wang's plant. Customer will be billed for freight charges; and unless customer specifies otherwise, all shipments will go best way surface as determined by Wang. Wang shall not assume any liability in connection with the shipment nor shall the carrier be construed to be an agent of Wang. If the customer requests that Wang arrange for insurance the customer will be billed for the insurance charges.
3. **PAYMENT** — Terms are net 30 days from date of invoice. Unless otherwise stated by customer, partial shipments will generate partial invoices.
4. **PRICES** — The prices shown are subject to change without notice. Individual document prices may be found in the Corporate Publications Literature Catalog (700-5294).
5. **LIMITATION OF LIABILITY** — In no event shall Wang be liable for loss of data or for special, incidental or consequential damages in connection with or arising out of the use of or information contained in any manuals or documentation furnished hereunder.

WANG

Fold



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO. 16 NO. CHELMSFORD, MA.

POSTAGE WILL BE PAID BY ADDRESSEE

WANG LABORATORIES, INC.
Supplies Division
c/o Order Entry Dept.
M/S 5511
51 Middlesex St.
No. Chelmsford, MA 01863



Fold



WANG LABORATORIES, INC., ONE INDUSTRIAL AVENUE, LOWELL, MA 01851 • TEL: 617/454-5000, TWX 710-343-6769, TELEX 94-7421

WANG

Printed in U.S.A.
700-8018
8-83-7.5M

THE WANG PROFESSIONAL COMPUTER

Assembly
Language
Reference
Manual

WANG

Assembly Language Reference Manual

The Wang Professional Computer Assembly Language Reference Manual

1st Edition — May, 1983
Copyright © Wang Laboratories, Inc., 1983
700-8315

WANG

WANG LABORATORIES, INC., ONE INDUSTRIAL AVENUE, LOWELL, MA 01851 • TEL. (617) 459-5000, TWX 710-343-6769, Telex 94-7421

Disclaimer of Warranties and Limitation of Liabilities

The staff of Wang Laboratories, Inc., has taken due care in preparing this manual; however, nothing contained herein modifies or alters in any way the standard terms and conditions of the Wang purchase, lease, or license agreement by which this software package was acquired, nor increases in any way Wang's liability to the customer. In no event shall Wang Laboratories, Inc., or its subsidiaries be liable for incidental or consequential damages in connection with or arising from the use of the software package, the accompanying manual, or any related materials.

NOTICE:

All Wang Program Products are licensed to customers in accordance with the terms and conditions of the Wang Laboratories, Inc. Standard Program Products License; no ownership of Wang Software is transferred and any use beyond the terms of the aforesaid License, without the written authorization of Wang Laboratories, Inc., is prohibited.

**WANG**

PREFACE

The Wang Professional Computer Assembly Language Guide provides an overview of the Wang Professional Computer (PC) assembler. In addition, it describes the arrangement of memory; the syntax and elements of the source program; and the directives you need for writing modules, functions, procedures, macros, conditionals, and for formatting the program. The last chapters examine the listing file, error messages, and the cross-reference file.

This manual does not explain how to program in assembly language nor does it cover the instruction set. For those who do not know how to program in 8086 assembly language, the following books are recommended.

IAPX 86,88 User's Manual. Santa Clara, CA: Intel Corporation Literature Dept., 1981.

An Introduction to ASM 86. Santa Clara, CA: Intel Corporation Literature Dept., 1981.

Morse, Stephen, P. The 8086 Primer. Rochelle Park, NJ: Hayden Publishing Co., 1980.

Rector, Russell and George, Alexy. The 8086 Book. Berkley, CA: Osbourne/McGraw-Hill, 1980.

The 8086 Family User's Manual. Santa Clara, CA: Intel Corporation, 1979.

8086/8087/8088 Macro Assembly Language Reference Manual. Santa Clara, CA: Intel Corporation, 1980.

The following books explain how to use the Wang Professional Computer.

The Wang Professional Computer Introductory Guide (700-8020)

The Wang Professional Computer Program Development Guide (700-8018).

This manual contains twelve chapters and four appendices. Chapter 1 provides a brief overview of the features of the assembly language and the assembler. It also explains how to invoke the assembler.

Chapter 2 explains how memory is arranged. It names and describes the function and purpose of each of the 13 registers and describes how to address an element in memory, how to use the segment registers, and how to address segments. In addition, it explains the purpose of the flag register and the way each flag is used.

Chapter 3 explains the elements of the source program, naming each of the many kinds of statements, identifiers, assembler directives, labels, symbols, numbers and their possible notations, strings, constants, expressions, and operands that are available to the Wang assembler. Chapter 4 names the operands associated with assembly language and describes their functions.

Chapters 5 through 9 describe the different types of assembler directives. Chapter 5 lists the data directives through which you define and initialize data. Chapter 6 describes the various assembler directives for writing program modules and functions and explains how to use them. Chapter 7 presents the assembler directives used for macros and repeat blocks. Chapter 8 contains information about conditional directives and how to use them. Chapter 9 discusses the listing directives that help you to format the page of your program or to suppress text.

Chapters 10 through 12 describe the tools that you can use to understand and debug your source program. Chapter 10 contains a detailed study of the listing file (the assembler file that lists the machine code), the address of each element of your program, and the symbols used in the program. Chapter 11 contains a list of error messages and their meanings. Chapter 12 describes the cross-reference listing file, CREF, and explains what it does and how to use it.

There are four appendices: Appendix A gives a listing of ASCII character codes; Appendix B gives a functional list of all assembler directives; Appendix C is a supplement to Chapter 9 and contains a listing of all error messages in numerical rather than in alphabetical order; and Appendix D provides a listing of the instruction set, the flags that each instruction effects, a description of what the instruction does, and examples of the use of each instruction.

CONTENTS

CHAPTER 1 INTRODUCTION

1.1	Assembly Language Features	1-1
1.2	How the Assembler Operates: An Overview	1-2
1.3	How to Invoke the Assembler	1-3
	Calling the Assembler through the DOS	
	Processor Command	1-4
	Creating an Assembly Language Menu Selection	1-7
	Control Characters	1-8
1.4	Creating an Executable File	1-10

CHAPTER 2 THE ARRANGEMENT OF MEMORY

2.1	Segments and Base Addresses	2-1
2.2	Register Structure	2-2
	Instruction Pointer	2-2
	General Purpose Registers	2-2
	Pointer and Index Registers	2-3
	Segment Registers	2-3
2.3	Addressing Modes	2-3
	Direct Offset Addressing	2-3
	Indirect Addressing	2-4
	Register Offset Addressing	2-4
	Addressing through Base and Index Registers	2-4
	Addressing through Base and Index Registers	
	Plus an Offset	2-4
2.4	Port Addressing	2-5
2.5	Flags	2-5

CHAPTER 3 THE SOURCE PROGRAM

3.1	Legal Characters and Names	3-1
3.2	Format of the Source Program	3-2
3.3	Statement Line Format	3-5
3.4	Types of Source Program Statements	3-5

CONTENTS (continued)

3.5	Elements of the Command Statement	3-6
	Identifiers	3-6
	Variables	3-6
	Labels	3-7
	Numbers and Their Notation	3-7
	Strings	3-8
	Constants	3-8
	Comments	3-8
	Expressions	3-9
	Assembler Directives	3-9
	Operands	3-9
CHAPTER 4	OPERATORS	
4.1	Arithmetic Operators	4-1
4.2	Relational Operators	4-2
4.3	Logical Operators	4-3
4.4	Attributable Operators	4-4
	SEG	4-5
	OFFSET	4-5
	TYPE	4-6
	LENGTH	4-6
	SIZE	4-7
	.TYPE	4-7
4.5	Synthetic Operators	4-8
	PTR	4-8
	THIS	4-9
	Segment Override	4-9
	SHORT	4-10
	HIGH and LOW	4-11
4.6	Record Specific Operators	4-11
	Shift Count	4-11
	WIDTH	4-12
	MASK	4-13
4.7	Order of Precedence for Operators	4-14
CHAPTER 5	DATA DIRECTIVES	
5.1	The ASSUME Directive	5-1
5.2	The Data Allocation Statement	5-2
5.3	EQU	5-4
5.4	Equal Signs	5-4
5.5	The EVEN Directive	5-5

CONTENTS (continued)

5.6	The INCLUDE Directive	5-5
5.7	The LABEL Directive	5-6
5.8	The NAME Directive	5-6
5.9	The ORG Directive	5-7
5.10	Procedures	5-7
5.11	The .RADIX Directive	5-9
5.12	The RECORD Directive	5-9
5.13	The Structure Directive	5-11
CHAPTER 6	USING PROGRAM MODULES AND FUNCTIONS	
6.1	Using the Same Symbol in Multiple Modules	6-1
	The PUBLIC Directive	6-1
	The EXTR Statement	6-2
6.2	Combining Segments	6-3
	Segment Name	6-3
	Align Type	6-4
	Combine Type	6-4
	Class Name	6-6
6.3	Using Groups	6-6
CHAPTER 7	MACRO AND REPEAT BLOCKS	
7.1	Macros	7-1
	Creating a Macro	7-1
	Invoking a Macro	7-2
	Deleting a Macro	7-3
	The LOCAL Statement	7-3
7.2	Repeat Blocks	7-4
	Repeat Block	7-5
	Indefinite Repeat Block	7-5
	Indefinite Repeat Character	7-6
7.3	Special Macro Operators	7-7
	Ampersand (&)	7-7
	Angle Brackets (< >)	7-7
	Two Semicolons (;)	7-8
	Exclamation Mark (!)	7-8
	Percent Sign (%)	7-8
CHAPTER 8	CONDITIONAL DIRECTIVES	
8.1	The Ten Conditional Directives	8-2

CONTENTS (continued)

CHAPTER 9 LISTING DIRECTIVES

9.1	Format Directives	9-1
	The PAGE Directive	9-1
	The TITLE Directive	9-2
	The SUBTTL Directive	9-2
9.2	%OUT Directive	9-3
9.3	Text Suppression Directives	9-4
	The .LIST and .XLIST Directives	9-4
	The .SFCND, .LFCOND, and .TFCOND Directives	9-5
	The .XALL, .LALL, and .SALL Directives	9-6
	The .CREF and .XCREF Directives	9-6

CHAPTER 10 LISTING FILE

10.1	The Listing File	10-1
10.2	The First Section of the Listing File	10-2
10.3	The Symbol Table	10-6
	The Listing of Macros	10-7
	The Listing of Structures and Records	10-7
	The Listing of Segments and Groups	10-9
	The Listing of Symbols	10-11

CHAPTER 11 ERROR MESSAGES

11.1	Assembler Error Messages	11-1
11.2	I/O Handler Errors	11-8
11.3	Runtime Errors	11-9

CHAPTER 12 THE CROSS-REFERENCE FILE

12.1	Creating a Cross-Reference File	12-1
12.2	Invoking CREF	12-2
	Invoking CREF: Method One	12-2
	Invoking CREF: Method Two	12-2
12.3	Command Characters	12-3
12.4	The Format of the Cross-Reference Listing File	12-4
12.5	Error Messages	12-5
12.6	The Format of CREF	12-6

CONTENTS (continued)

APPENDICES

Appendix A	ASCII Character Codes	A-1
Appendix B	Table of Assembler Directives and Operators	B-1
	Memory Directives	B-1
	Macro Directives	B-2
	Conditional Directives	B-3
	Listing Directives	B-3
	Attribute Operators	B-3
	Precedence of Operators	B-4
Appendix C	Error Messages	C-1
Appendix D	The 8086 Instruction Set	D-1

INDEX	INDEX-1
-------------	---------

FIGURES

Figure 1-1	The Files Produced by the Assembler	1-3
Figure 1-2	The Interactive Menu Design Package Screen	1-7
Figure 2-1	The General Purpose Registers	2-3
Figure 2-2	Flags	2-6
Figure 4-1	The Shift Count of a Record	4-12
Figure 4-2	The Width of a Record Field	4-13
Figure 4-3	The Mask of a Field	4-14
Figure 5-1	The Placement of Bits in a Record	5-10
Figure 8-1	The Nesting of Conditionals	8-2
Figure 10-1	The Shift Count and Mask of a Record	10-9

TABLES

Table 4-1	Logical Operations	4-4
Table 5-1	The Five Data Types	5-2
Table 9-1	The Use of .TFCOND with /X	9-6

CHAPTER 1

1.1 ASSEMBLY LANGUAGE FEATURES

The Wang Professional Computer (PC) assembly language incorporates many features usually found only in larger assemblers: macro assembly, conditional assembly, and a variety of assembler directives.

The PC assembly language produces relocatable object code that you can load anywhere in memory, allowing the program to execute in the most efficient location. Because the code is relocatable, you can create programs in modules that are units you can assemble and test individually.

Through the assembly language's macro facility, you can use the same block of code in many programs, eliminating the need to recode frequently used instructions. Each time you need a macro, you insert a call in the program. The assembler then searches for the macro with that name and expands the code into the file.

You can nest macros, i.e., you can insert a macro within another macro. Only the size of memory limits the number of macros that you can nest within each other. The macro facility also includes repeat, indefinite repeat, and indefinite repeat character directives for repeating the macros.

The assembly language supports a set of conditional directives that make the testing of conditions easier. The conditional directives available to the assembly language can test for blank or nonblank arguments, for defined or undefined symbols, for equivalence, for first or second assembler passes, or for the similarity of two strings. As a result of the test, the program can branch, leaving unneeded portions of code unassembled. You can nest conditional directives to 255 levels.

The PC assembler can correct some typographical errors. For example, if you enter an operand with no defined type for an instruction that accepts only one type of operand, the assembler returns an error message but generates the "correct" code.

In the following example, the assembler scans the instruction and finds two different size elements as the source and the destination. AL is a byte of register A, while WORDLBL appears to be an error. From the instruction, BL, a byte register, seems to be a word long. The assembler modifies the operand to fit the register size, making WORDLBL into a byte.

```
MOV AL, WORDLBL
```

1.2 HOW THE ASSEMBLER OPERATES: AN OVERVIEW

The Wang assembler is a two-pass assembler. Once you submit the source file, the assembler reads it twice. During the first pass, the assembler evaluates the statements, expands macro call statements, calculates the amount of code the assembler will generate, and builds a symbol table where it assigns values to all symbols, variables, labels, and macros.

During the second pass, the assembler fills in the symbol table and emits the relocatable object code into an object file with the default extension name .OBJ. The Linker creates the executable file from the object file by defining all undefined references and giving a nonrelocatable address to all relocatable code.

You can assemble the source file without creating an .OBJ file. The assembler performs all of the other assembly steps, but does not send the object code to the disk. It displays only erroneous source statements on the terminal screen. In this way, you can assemble modules quickly, test them, and correct errors before putting the object code on the disk.

Upon command, the assembler creates two other files: a listing file and a cross-reference file. The listing file has the default extension .LST. It contains a copy of the source program; the offset address of each instruction; the machine code translation of each statement in hexadecimal notation; a symbol table with the values of all symbols, labels, and variables; and a listing of all macros that occur within the program. The assembler lists all macros in the symbol table. For more information on the listing file, refer to Chapter 10, The Listing File.

The cross-reference file with its default extension of .CRF contains a compact representation of variables, labels, and symbols. When the cross-reference program processes the cross-reference file, it converts the file into an expanded symbol table that lists all the variables, labels, and symbols in alphabetical order. Each element is followed by the source program line number in which each symbol is defined, and the line numbers in which each symbol is used throughout the entire program. The default extension name for the final cross-reference file is .REF. For more information on the cross-reference file, refer to Chapter 12, The Cross-Reference File.

Figure 1-1 shows all of the files produced by the assembler and how they relate to one another.

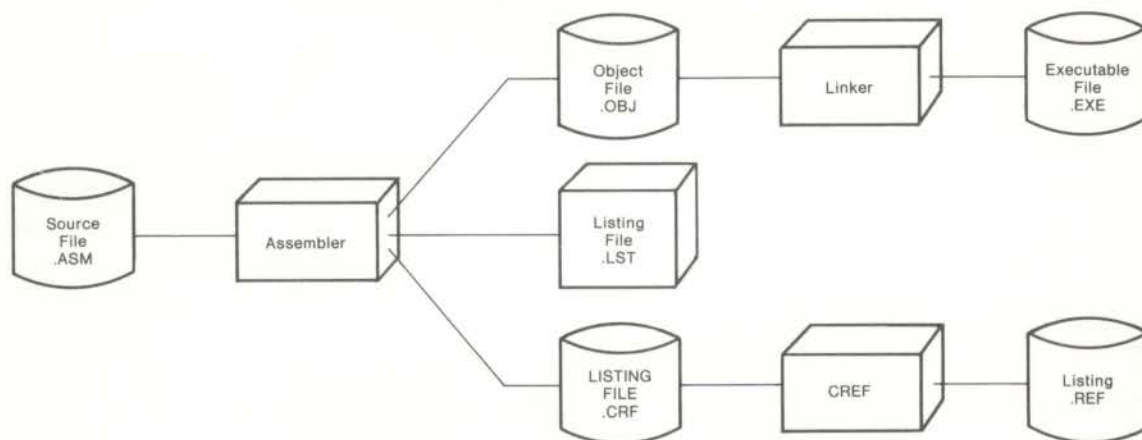


Figure 1-1. The Files Produced by the Assembler

1.3 HOW TO INVOKE THE ASSEMBLER

To invoke the assembler, perform the following procedure.

1. Write the source file using the PC Editor.
2. Select either the DOS Command Processor or Other from the Main System Menu. If you have previously created an assembly language menu selection on the Program Development Menu, you can invoke the assembler after selecting the Program Development Menu.

1.3.1 Calling the Assembler through the DOS Command Processor

From the System Menu, select the option, DOS Command Processor. Once the system prompt appears, you can invoke the assembler. Enter the command MASM. This causes the processor to load the assembler into memory. After you enter MASM, you can enter all the file names on the same line and then press the RETURN key (Method One), or you can press the RETURN key immediately, wait for a series of prompts to appear, and respond to them one by one (Method Two).

Method One

If you choose to write the file names on the same line and then press the RETURN key, the syntax is as follows.

MASM source file, object file, listing file, cross-reference file

Upon this command, the assembler assembles your source program and creates an object, listing, and cross-reference file. In the following example, the assembler assembles a source file, an object file, a listing file, and a cross-reference file, all named PROG1.

MASM PROG1, PROG1, PROG1, PROG1

Default Extension Names

Each file has a default extension name. If you do not supply your own extension, the assembler uses the default name. The default file names are as follows.

File Name	Default Extension
SOURCE	.ASM
OBJECT	.OBJ
LIST	.LST
CROSS-REFERENCE	.CRF

Switches

There are three switches that control some additional assembly features. They are as follows.

/D	Produces a source listing on both assembler passes. To activate this switch, you must create a source listing file.
/O	Outputs the listing file in octal radix.
/X	Suppresses the listing of conditionals whose values are false. Does not affect any code controlled by the .SFCOND or .LFCOND directives. (For more information, refer to Section 9.3.2, The .SFCOND, .LFCOND, and .TFCOND Directives.)

If you wish to use one or more switches with the list of file names, enter the switch's name either after the appropriate file or at the end of the list of file names, separating each switch's name with a slash. In the following example, the switches follow the appropriate file; in the second example, the switches are at the end of the file listing.

```
MASM PROG1, PROG1, PROG1/D/X, PROG1
MASM PROG1, PROG1, PROG1, PROG1/D/X
```

Method Two

A second way of invoking the assembler is to enter the command MASM, immediately followed by a carriage return. The processor loads the assembler into memory. The assembler returns four prompts, one at a time. You answer each prompt with the name of the file you wish to create, using up to three switches and a carriage return. If you want to use a switch, you must enter the file name. You cannot enter only a switch as a response to a prompt. For example, as the response for the list file, you can enter:

```
PROG1.LST/D/O
```

If you are creating a file called PROG1, the following prompts appear after you write MASM.

Prompt 1 Source filename [.ASM]:

You can respond to this prompt by entering:

- The file name alone, causing the file to use the specified name and the default extension .ASM, or
- The file name and an extension, if the extension is different from the default.

Prompt 2 Object filename [PROG1.OBJ]:

You can respond to this prompt by:

- Pressing RETURN, so that the file has the same file name as the source program with the default extension .OBJ;
- Entering the file name alone, so that the file has the specified name and the extension .OBJ; or
- Entering the file name and an extension, so that the object file has the specified file name and extension.

Prompt 3 Source listing [NUL.LST]:

You can respond to this prompt by:

- Pressing RETURN, so that the assembler creates no listing file;
- Entering a file name alone, so that the listing file has the specified file name with the default extension .LST; or
- Entering a file name and an extension name which create a listing file with the specified name and extension.

Prompt 4 Cross reference [NUL.CRF]:

You can respond to this prompt by:

- Pressing RETURN, so that the assembler does not create a cross-reference file;
- Entering a file name alone, which creates a file of the specified name and the default extension .CRF; or
- Entering a file name and an extension name, which create a file of the specified name and extension.

The "Other" Option

You can invoke the assembler through the "Other" option of the Main System Menu. Once you choose Other and press the EXECUTE key, the following prompt appears.

File spec: _____

On the line, enter:

MASM.EXE

The following prompt appears:

Source file [.ASM]:

Respond to this prompt as you did to those described for Method Two.

1.3.2 Creating an Assembly Language Menu Selection

You can create your own assembly language menu selection on the Program Development Menu. In this way, you can call upon assembly language through the use of menus alone.

To create this selection, follow the instructions in The Wang Professional Computer Introductory Guide, 700-8020.

The procedure is as follows.

1. Select System Utilities from the Main System Menu.
2. Select Modify System Menus from the System Utilities Menu. The Interactive Menu Design Package screen (Figure 1-2) appears.

Wang Professional Computer
Interactive Menu Design Package
Release

Select desired option

 - Create new menu
 ■ Edit existing menu

Menu File Id: _____
On Drive: -

EXECUTE-Proceed
CANCEL -Terminate
HELP -Instructions

Figure 1-2. The Interactive Menu Design Package Screen

3. From the Interactive Menu Design Package screen, select the Edit existing menu option. When you press RETURN, the cursor positions itself on the blank space next to Menu File Id. Because you are editing the Program Development Menu, enter:

PRGDVMEN.DAT

If your operating system diskette is in Drive A, as an example, type in A, as shown.

On Drive: A

Once you have entered "assembly language" as a new menu item, answer the following prompts as shown.

- File Id of module to invoke MASM.EXE
- On drive B (if you put the assembler in Drive B only)
- Module type:
 program

After you have responded to the prompts, exit from the Design Package and select Assembly Language. The following prompt appears.

Source file [.ASM]

Respond to this prompt as you did to those described for Method Two.

1.3.3 Control Characters

You can replace file names with three other responses: a carriage return, a comma, or a semicolon. The effect of each response is as follows.

Carriage return	causes the assembler to accept the default name for that file only.
Semicolon	causes the assembler to accept the default file names for all of the remaining files.
Comma	causes the assembler to accept the default for that particular file for Method One only (in which you list all four file names after the MASM command).

The Carriage Return

The carriage return accepts the default name for the file. In the following example, PROG1.OBJ becomes the name of your file when you press RETURN.

Object filename [PROG1.OBJ]: (RETURN)

The Semicolon

Use a single semicolon followed by a carriage return at any time after the source file prompt to select the default responses for all of the remaining prompts. After using the semicolon, there are no more prompts. Therefore, if you want to skip over one prompt only, use the carriage return.

In the following example, once you enter the semicolon to respond to the object file prompt, no further prompts appear. The assembler assembles the source and object files. The defaults for the list and cross-reference files are NUL files. Therefore, the assembler creates no list or cross-reference file.

Source filename [.ASM]: PROG1]
Object filename [PROG1.OBJ] ; (RETURN)

The Comma

Use the comma with Method One only. The comma causes the assembler to accept the default name for that file only. In the following example, the assembler creates PROG1.ASM, PROG1.OBJ, PROG1.LST, and PROG1.CRF.

MASM PROG1.ASM,,PROG1, PROG1

There are two other controls that help you in the creation of your assembly language program. Each causes you to exit from the mode that you are in and to return to the system prompt. They are:

CONTROL + C	Returns you to command level. If you are in the assembly program, CONTROL and C causes assembly to stop. Once back at command level, you can invoke the assembler again.
SHIFT + CANCEL	Returns you to command level.

1.4 CREATING AN EXECUTABLE FILE

After you have entered the last file name, the assembler begins to assemble your program. When it is finished, a list of warning errors and severe errors appears on the screen. You can go back into editing mode to correct the errors.

If no error messages appear, the assembler has assembled your program successfully. You must link your program before you can run it. To call upon the Linker, return to the system prompt and enter the word LINK, followed by your file name and RETURN.

```
B: LINK PROG1
```

The linker responds with a series of prompts.

```
Run File [PROG1.EXE]:  
List File [NUL.MAP]:  
Libraries [ .LIB]:
```

Answer these prompts by supplying the name of your program.

```
Run File [PROG1.EXE]: PROG1  
List File [NUL.MAP]: PROG1  
Libraries [ .LIB]: PROG1
```

For more information concerning the Linker and its utilities, refer to The Wang Professional Computer Program Development Guide, 700-8018.

When the Linker is finished, it sends out a series of error messages. To run your program, you must correct the problems in the Wang PC Editor. Then you can reassemble and relink your program.

To run your program, write its name after the system prompt. If you called your program PROG1, you would enter it after the system prompt and press RETURN as follows.

```
B:PROG1
```

CHAPTER 2

THE ARRANGEMENT OF MEMORY

2.1 SEGMENTS AND BASE ADDRESSES

The processor in the Wang Professional Computer is designed to address up to 1 megabyte of memory (1,048,576 bytes). A pair of addresses is required to address memory: a base and an offset. Through the use of these addresses, the PC can access up to 256K (4*64K) of memory at any one time. By changing the base values, you can access the full megabyte of memory.

A segment is composed of up to 64k bytes of continuous memory. The assembler assigns a base address to each segment. Segments can be adjacent, apart, partially overlapped, or fully overlapped.

The base address is a 20-bit number. The high 16 bits of the base address are stored in segment registers. The low nibble (four bits) of the address is zero. The processor shifts the 16-bit address in the segment register four binary positions to the left to obtain the 20-bit address. For example, if CS contains 4321H, CS points to the segment with the base location 43210H. Because the low nibble is zero in hexadecimal notation, the address of a segment is always divisible by 16 (just as a number that ends in zero is always divisible by 10 in decimal notation).

A physical segment is made up of one or more logical segments. The logical segments are related to the physical segments. The processor can access up to four segments at a time. Each logical segment is addressed from a base address. Each begins with a SEGMENT statement and ends with an ENDS statement. Each segment is associated with a segment register that bears the same name as the segment and stores the base address of the segment. The segment registers are called:

- Code segment (CS) register
- Data segment (DS) register
- Stack segment (SS) register
- Extra segment (ES) register

2.2 REGISTER STRUCTURE

The registers are readily accessible to the CPU during execution. When you use registers efficiently, you save computer time. In the Wang PC, there are 13 registers:

- the instruction pointer (IP)
- four general purpose registers: AX, BX, CX, DX
- two pointers: the stack pointer (SP) and the base pointer (BP)
- two index registers: source index (SI) and destination index (DI)
- four segment registers: code segment (CS), data segment (DS), stack segment (SS), and extra segment (ES)

2.2.1 Instruction Pointer

All instruction fetches come from the current code segment. The IP register contains the offset from the base of the current code segment of the next instruction to be executed. However, you cannot access the IP through any instruction. Its presence is invisible to the programmer.

2.2.2 General Purpose Registers

The general purpose registers are intermediate storage locations for constants, variables, and data. The general purpose registers make these elements more immediately accessible to the processor.

You can access the general purpose registers in either words or bytes. When you call up the whole word, one of the following registers is called: AX, BX, CX, and DX. "X" in this context refers to a 16-bit value.

8-bit registers are useful for manipulating one byte values. The low bytes, containing the 8 least significant bits of AX, BX, CX, and DX, respectively, are called AL, BL, CL, and DL. The high byte registers, which contain the most significant bits, are called AH, BH, CH, and DH. Figure 2-1 illustrates the words and bytes in the general purpose registers.

Word	High	Low
AH	AL	AX
BH	BL	BX
CH	CL	CX
DH	DL	DX

Figure 2-1. The General Purpose Registers

2.2.3 Pointer and Index Registers

The pointer and index registers are four 16-bit registers that can participate in most of the same arithmetic and logical operations as the general registers. Generally, however, these registers contain offset addresses for addressing within a segment. For this reason, you use them within a segment for indirect addressing. While you use the stack pointer (SP) implicitly for stack operations, you use the source index (SI) and the destination index (DI) registers for string operations.

2.2.4 Segment Registers

There are four segment registers, each of which holds the start address of the segment of the same name. This address is called the base address. They are not available for any arithmetic or logical operation.

2.3 ADDRESSING MODES

When you write the source program, you should know the different ways to store data in memory and how to access it. The manner in which you store and access data is called the addressing mode. There are five addressing modes in the Wang PC assembly language.

2.3.1 Direct Offset Addressing

The offset of the addresses is within the elements themselves. For example:

```
MOV AX, BX; load BX into AX
MOV AX, WORD1; copies the contents of variable WORD1 into AX
```

2.3.2 Indirect Addressing

The address of the address to be accessed is stored in the register. For example:

```
MOV AX, [BX]; load the word whose address is in BX into AX.
```

2.3.3 Register Offset Addressing

The address uses a value in memory plus an offset. For example:

```
MOV AX, [BP+2]; add 2 to the address contained in the base pointer and
move the contents of the new address into AX.
```

2.3.4 Addressing through Base and Index Registers

The sum of a base register and an index register form an offset. When this offset is added to the appropriate segment register, the address is formed. With BX, the default segment is DS; with BP, it is SS. For example:

```
MOV AX, [BX][SI]
MOV AX, [BX][DI]
MOV AX, [BP][SI]
MOV AX, [BP][DI]
```

In each of the preceding commands, the base register is added to the index register, and the value is placed in AX. All of the allowable combinations of index and base registers are indicated here. You cannot use any other combination.

2.3.5 Addressing through Base and Index Registers Plus an Offset

The offset and the two registers are added together to form the offset address. This address is added to the segment address to form the program address of the element. For example:

```
MOV AX, VAR_1 [BX][SI]
MOV AX, VAR_1 [BX][DI]
MOV AX, [BX][SI+10]
MOV AX, VAR_1 [BP][DI]
```

The contents of the two registers plus the contents of the variable are added together and then moved to AX.

2.4 PORT ADDRESSING

Using the OUT instruction, you can use two different addressing modes to access ports located in the I/O space: direct port addressing and indirect port addressing. Direct port addressing through its 8-bit immediate operand allows access to ports 0-255. Indirect port addressing uses a 16-bit number stored in the DX register. Through indirect port addressing, you can access any port from 0 to 65,535.

2.5 FLAGS

In addition to the registers, the Wang PC central processor uses six 1-bit status flags and three 1-bit control flags. The status flags reflect the outcome of mathematical operations. They are set to 1 to reflect one condition and cleared to 0 to reflect the opposite condition.

Carry flag (CF)	Set if the operation results in a carry out of (from addition) or a borrow into (from subtraction) the high-order bit of the result.
Auxiliary carry flag (AF)	Set if the operation results in a carry out (from addition) or a borrow into (from subtraction) the low-order four bits of the result.
Overflow flag (OF)	Set if there has been an overflow, that is, a significant digit has been lost because the size of the result exceeded the capacity of its destination.
Zero flag (ZF)	Set if the result of the operation is zero.
Sign flag (SF)	Set if the operation generates a negative result.
Parity flag (PF)	Set if there was an even number of 1-bits, e.g., 3 is odd with an even parity, while 8 is even with an odd parity. This flag checks data transmission errors.

The three control flags can alter processor operations, depending upon whether they are set or clear. The flags are as follows.

Direction flag (DF)	The flag controls the direction of string operation instructions. The clearing of the direction flag causes the strings to process from low to high bits (to auto-increment). Setting the bit causes strings to process from high to low order bits (to auto-decrement).
Interrupt-enable flag (IF)	When set, the flag enables external interrupts; when cleared, the flag disables them. IF has no affect on nonmaskable interrupts.
Trap flag (TF)	If the trap flag is set, the processor goes into single-step mode for debugging. An internal interrupt is generated after each instruction so that you can study one instruction at a time.

Figure 2-2 shows the use of the control and status flags.

Control Flags	TF	TRAP
	DF	DIRECTION
	IF	INTERRUPT-ENABLE
Status Flags	OF	OVERFLOW
	SF	SIGN
	SF	ZERO
	AF	AUXILIARY CARRY
	PF	PARITY
	CF	CARRY

Figure 2-2. Flags

CHAPTER 3

THE SOURCE PROGRAM

The source program is the program that you create. It is composed of mnemonic instructions, operands, directives, and symbolic names stored as ASCII text files. The assembler translates your source program into an object program composed of binary code, the only code the computer can read.

3.1 LEGAL CHARACTERS AND NAMES

The legal characters in the Wang PC assembly language are:

- A-Z
- a-z
- 0-9
- special characters ? . @ _ \$

In addition to the general special characters that you can use in any name, the following group of special characters serve the following purposes.

Special Character	Function
Colon :	an operator to override the original segment for one operation only.
Period .	operator for field, record, or structure name. The separator between a structure name and a field name.
Square brackets []	in register and memory addressing, indicates the value in the register is an address, not data.
Parentheses ()	used in DUP expression and to change precedence of evaluation of operators (as in mathematical expressions).
Angle brackets < >	operator in initialization of values in records, structures, around parameters, in IRP macro blocks; indicates literals.

Special Character	Function
Dollar sign \$	symbolizes the location counter that keeps track of the current offset in the segment being assembled.
Comma ,	delimits the individual fields in an instruction so that the assembler recognizes them as separate fields.
Single and double quotations (' ') (" ")	delimits strings of text so that the assembler interprets everything within the quotations as being literal.
semicolon ;	symbolizes that you accept all remaining default files in Method One of invoking the assembler.

You can use names to represent code, data, and constants. A numeral between 0 and 9 cannot appear as the first character of a name, but must appear as the first character of a numeric value. If you use a period in a name, it must be the first character. You can use as many letters as you wish for a name within the confines of memory, but the assembler reads only the first 31 letters.

3.2 FORMAT OF THE SOURCE PROGRAM

The source program can begin with an optional program name and ends with a program END statement. The following program outline shows the optional NAME directive and the beginning of the main module accompanied by its final END statement.

```

NAME PROG2
.
.
.
START:  MOV AX, ABC
.
.
.
END

```

The source program consists of one or more segment blocks that begin with a SEGMENT statement and a segment name and end with an ENDS statement. A segment is a block of code addressed through one segment register and is 64K or fewer bytes long. The segment blocks can appear in any order in your source file. The SEGMENT-ENDS directive tells the assembler in which segment register it can find the base address for each symbol or line of code. The syntax of the SEGMENT-ENDS directive is as follows.

```

segment name SEGMENT [align] [combine] ['class']
      .
      .
      .
segment name ENDS

```

At runtime, all instructions that generate code and data are in separate segments. Your program can be a segment, part of a segment, several segments, parts of several segments, or a combination of these elements. Without a SEGMENT statement, your program generates an "invalid object" error at link time.

The segment name is a unique, legal name. The align type can be PARA, BYTE, WORD, or PAGE. (Refer to Chapter 6 for definitions of align type, combine type, and class name.) The combine type can be PUBLIC, COMMON, AT expression, STACK, MEMORY, or blank (which defaults to Not Combinable, also called Private).

The Linker uses class name to load segments together. It loads all segments belonging to the first class that it encounters together, and then all the segments belonging to the next class, and so on, until all of the segments have been loaded.

You can nest segments. Either of the following two arrangements is legal.

```

A SEGMENT
  .
  .
B SEGMENT
  .
  .
B ENDS
  .
  .
A ENDS

```

```

A SEGMENT
  .
  .
A ENDS
B SEGMENT
  .
  .
B ENDS
A SEGMENT
  .
  .
A ENDS

```

The following nesting pattern is illegal.

```

A SEGMENT
  .
  .
B SEGMENT
  .
  .
A ENDS
  .
  .
B ENDS

```

The following example consists of a data segment that defines two variables, each a word long and initialized to 0.

```
FIRST_DATA SEGMENT
```

```
    VAR1 DW 0
    VAR2 DW 0
```

```
FIRST_DATA ENDS
```

The source program begins with an optional NAME directive, followed by an optional ASSUME statement that gives the base segment address to symbols. If the ASSUME statement is not present, you must attach a segment prefix to each symbol every time it is used. For more information concerning the ASSUME statement, refer to Section 5.1, The Assume Statement.

Following the ASSUME statement are one or more SEGMENT-ENDS statements. The main module begins with a SEGMENT statement. Each module ends with an ENDS statement. The source file ends with an END statement followed by an optional label that gives the start address of the main module.

```
NAME PROG1
ASSUME DS: VAR1, CS:VAR2
SEG1      SEGMENT
           statement
```

```
    .
    .
    .
    .
```

```
SEG1      ENDS
```

```
MAIN_SEG  SEGMENT
           START: segment
```

```
    .
    .
    .
    .
```

```
MAIN_SEG  ENDS
           END
```


3.3 STATEMENT LINE FORMAT

Assembler language directive statements consist of four fields: name, action, expression, and comment. The format is as follows.

Name	Action	Expression	;Comment
VAR1	DB	OD5EH	;this is a comment
VAR2	DW	10,20	;this is another comment

Instruction statements usually consist of three fields: action, express, and comment. The format is as follows.

Label	Action	Expression	;Comment
PARS1	MOV CX,VAR1		;this is a comment
	ADD AX, BX		;copy the contents of register BX into ;register AX

3.4 TYPES OF SOURCE PROGRAM STATEMENTS

A source program is composed of two types of statements: action and assembler directives. The name field, when present, is the first entry on the statement line. The name can begin in any column, although normally names begin in column one. The action statement names the operation to be executed. If the name field is blank, the action field is the first entry in the statement format. It can appear in Column 1 and run to the end of the line. The following are examples of action statements.

```
INC BH ;add one to register BH
MOV AX, BX ;move the contents of register BX to register AX
```

Assembler directives give instructions to the assembler for input and output, format control, memory organization, conditional assembly, listing and cross-reference control, and definitions. Unlike instruction statements, assembler directives do not generate machine instructions in the assembled code.

In the following examples, the first assembler directive names the program, the second names the segment, and the third assigns a value to a symbol.

```
NAME PROG_2
CODE SEGMENT
VAR_1 EQU 2
```

Appendix B lists all assembly directives available in the Wang PC assembly language.

3.5 ELEMENTS OF THE COMMAND STATEMENT

You should write the command line so that it is comprehensible. Although you can write command lines in a free-form style, they are easier to read if you write them in a column format. The format of the command statement is as follows.

```
LABEL:      MNEMONIC argument,...,argument    ;comment
```

The command line is composed of many different elements: variables, labels, constants, numbers, comments, and expressions that are composed of operands and operators, strings, and symbols.

3.5.1 Identifiers

An identifier is a symbolic name that represents variables, labels and constants. The identifier is composed of up to 31 letters, numbers, and underscore characters. Although you can use more than 31 characters when naming an identifier, the assembler disregards all characters after 31.

3.5.2 Variables

A variable is an element of data with a symbolic name. A variable has three attributes: type, segment, and offset. By placing your variables at the beginning of your code, you allow the assembler to recognize their attributes before they are used in code. In this way, the assembler generates shorter instructions.

Type

The type of a variable refers to its size. It can either be a byte (DB), a word (DW), a double word (DD), eight bytes (DQ), or ten bytes (DT), or a structure.

Segment

The segment attribute refers to the segment in which you define the variable and, consequently, to the base address of the variable: the data segment, the code segment, the stack segment, or the extra segment. Through the ASSUME directive or a segment override operator, the assembler couples the variable name with the segment name.

Offset

The offset attribute is a 16-bit unsigned value that refers to the distance (in bytes) from the start of the segment in which the variable is defined to where it is used. The following example illustrates the attributes with which variables are defined.

```
DATA    SEGMENT
        WORDVAR DW
        BEGIN    DB 5
```

The segment attribute for the variable BEGIN is data, the type attribute is byte (DB), and the offset attribute is two since two bytes were defined before it in the segment.

3.5.3 Labels

Labels, like variables, have three attributes: segment, offset, and type. The segment attribute is the paragraph number of the beginning of the segment in which the label is defined. Labels are always assumed to be in a CS register.

The offset of a label is defined the same way as the offset of a variable. (Refer to Subsection 3.5.2.) The type attribute indicates whether the label is NEAR or FAR. NEAR labels can be referenced only within the same segment or group in which they are defined, while FAR labels can be referenced from the same or other segments. NEAR identifies a pointer length of 2 bytes; FAR identifies a pointer length of 4 bytes.

3.5.4 Numbers and Their Notation

Numbers, except numbers including decimals, must be written with an indication of whether they are hexadecimal, binary, or octal. The following examples demonstrate how to write numbers.

```
0101011B = Binary
735Q     = Octal
735O     = Octal
951      = Decimal
951D     = Decimal
OFFH     = Hexadecimal
25.23E-7 = Real: floating point
8217R    = Real
```

To avoid confusion, begin with a number rather than with a letter when using hexadecimal notation. A leading zero as in OFFH tells the assembler that the characters form a hexadecimal number and not a word.

3.5.5 Strings

Strings are characters enclosed in delimiters, usually single quotation marks. The following are examples of strings.

```
'a string'
'goodbye'
```

3.5.6 Constants

There are two types of constants available in the Wang PC assembly language: whole number constants and string constants. Unlike variables, constants have fixed values.

A whole number constant is a nonfractional number between zero and 65,535. A string constant is composed of one or two letters enclosed in apostrophes. String constants are valued the same as their ASCII equivalents. For example, 'A' is 41H, the ASCII code for A.

3.5.7 Comments

The comment begins with a semicolon and can run to the end of the line on which it begins. To expand the comment to another line, another initial semicolon is required. The comment usually documents and explains programs. In the assembly process, the assembler ignores the comment field.

The comment line is written as follows.

```
MOV AX, BX ;This instruction copies the contents of BX to AX.
;This is a comment line
```

Another way of entering a comment is through the COMMENT directive. This directive permits you to use multiple lines without using the semicolon. The syntax of the COMMENT directive is:

```
COMMENT 'comment'
```

You surround the comment with delimiters, any non-blank character, on each side. The assembler interprets everything inside the delimiters as a comment. In the following example, the comment line is on two lines:

```
COMMENT 'This is a very,
very long comment.'
```


3.5.8 Expressions

An expression is the value upon which an instruction or directive performs its functions. Every expression consists of at least one or more operands. Operators join multiple operands, resulting in what looks like a mathematical expression. The following are examples of expressions.

```
MOV AX, AGAIN*2
ADD BX, REAL*(ALEX-BETA)
A EQU (MASK B) XOR (10 and MASK B1)
```

3.5.9 Assembler Directives

Assembler directives give instructions to the assembler for input and output, format control, memory organization, conditional assembly, listing and cross-reference control, and definitions. Assembler directive do not generate machine instructions in the assembled code.

The first of the following assembler directives names the program; the second names the segment; and the third gives value to a variable:

```
NAME PROG2
CODE SEGMENT
VAR1 EQU 2
```

3.5.10 Operands

Operands are the values that follow the instruction. The instruction directs the processor to act upon the operand in some way.

Operands can be constants, data items, symbols, registers, labels, variables, structures, or offsets. You can index them through the base or index registers or with constants. In a single operand operation, the instruction indicates what the processor should do to the operand. In the following instruction, for example, the processor adds one to the index register.

```
INC DI
```

In a double operand operation, one using two operands, the operand to the right is the source, and the operand to the left is the destination. The processor acts upon the source first and the destination next.

In the following command, the operands are BX and AX. BX is the source. When the operation is complete, the value of BX remains exactly what it was before the operation began; the value of AX, the destination, matches that of BX.

```
MOV AX, BX
```

There are three types of operands: immediate, register, and memory.

Immediate Operands

Immediate operands are constant values. In a double operand operation, an immediate operand can be the source operand only. In the following examples, the constant 9 is an immediate operand. In the first example, it is a constant. In the second example, it is a symbol for a constant.

```
MOV AX, 9
```

```
VAR1 EQU 9
MOV AX, VAR1
```

Register Operands

When the operand is a register, it is called a register operand. In the following double operand operation, both operands are registers.

```
MOV AX, BX
```

Memory Operands

A memory operand directs the processor to an address to find some data or instruction. A memory operand is an offset from a base address. There are three kinds of memory operands.

- indexed memory operands
- direct memory operands
- structure operands

Direct memory operands, consisting of a single offset value, can be either the source or the destination. They do not use registers, but consist of labels, simple variables, and offsets. In the following example, VAR1 represents an address rather than a symbolic constant.

```
MOV AX, VAR1
```

Indexed memory operands use the base and index registers, constants, displacement values, and variables, often in combination. By combining indexed operands, you create an address expression.

Indexed memory operands use square brackets to indicate that the expression evaluates to an address rather than data. In the following examples, VAR1 contains an address. In the first example, the processor fetches the data from the address indicated in VAR1 and copies it into BX. In the second example, the processor fetches the data 5 bytes further than the address stored in VAR1 and copies it into BX.

```
MOV BX [VAR1]  
MOV BX [VAR1+5]
```

A structure operand is an operand used exclusively with a structure. Once you define a structure, you can create a structure operand. Structure operands allow you to access any value by field name indexed through BP. For instance, through the use of structure operands, you can access parameter values passed on the stack, not just the value at the top. To learn more about structure operands, refer to Section 5.13, The STRUCTURE Directive.

CHAPTER 4 OPERATORS

There are four kinds of operators: arithmetic, relational, logical, and attributal. Arithmetic, relational, and logical operators combine or compare operands. Attribute operators override the attributes of operators, return the value of the attributes, or isolate fields of records.

4.1 ARITHMETIC OPERATORS

Eight arithmetic operators plus two shift operators provide the common mathematical functions. The arithmetic operators combine operands to form an expression that results in a data item or an address. With two exceptions, operands must be constants.

- For plus only, one operand must be constant.
- For minus, the left operand can be a variable or both operands can be variables, but the right operand must not be a variable if the left is constant.

The eight arithmetic operators are as follows.

*	Multiply (operand * operand). The operands must be absolute numbers. Example: MOV AL, 3*2
/	Divide (operand / operand). The operands must be absolute numbers. Example: CMP CX, 100H/12
MOD	Modulo (operand MOD operand) Divide the left operand by the right operand and return the value of the remainder (modulo). Both operands must be absolute. In the following example, you divide 17 into 100, returning decimal 15 into AX. Example: MOV AX, 100 MOD 17

unary-	(-operand). The value following the minus sign is negated.
+	Add. (operand + operand). An operand evaluates to an absolute number, a variable, or a label name. One operand must be constant; one may be a variable.
-	Subtract the right operand from the left operand (operand - operand). The left operand can be a variable, or both operands can be variables. But the right operand can be a variable only if the left operand is also a variable and in the same segment.
SHR	Shift right (operand SHR count) SHR is preceded by a numeric operand that evaluates to an absolute number and is followed by an integer that specifies the number of bit positions the value is to be right shifted. Each position shifted right divides the number in half. In the following example, the processor shifts 1100000 five places to the right, moving 11B into AX. Example: MOV AX, 1100000B SHR 5
SHL	Shift left (operand SHL count). SHL is preceded by a numeric expression that evaluates to an absolute number and is followed by an integer that specifies the number of bit positions the value is to be left shifted. With each bit position shifted left, the value of the original number doubles. In the following example, 0110B shifts left five positions, moving 011000000B into AX. Example: SHL AX, 0110B SHL 5

4.2 RELATIONAL OPERATORS

The relational operator is a logical operator that determines whether a particular relationship exists between two operands. You use relational operators most often with conditional directives and conditional instructions. The syntax for the relational operator is as follows.

operand relational operator operand

The operands of relational operations can be absolute numbers or variable or label names defined in the current module. The result of a relational operation is an absolute number: it returns all 1's for true and all 0's for false. The relational operators are as follows.

EQ	Equal. Returns true if the operands equal each other.
NE	Not equal. Returns true if the operands do not equal each other.
LT	Less than. Returns true if the left operand is less than the right operand.
LE	Less than or equal. Returns true if the left operand is less than or equal to the right operand.
GT	Greater than. Returns true if the left operand is greater than the right operand.
GE	Greater than or equal. Returns true if the left operand is greater than or equal to the right operand.

4.3 LOGICAL OPERATORS

AND, OR, AND XOR compare the binary values of corresponding bit positions of each operand. They return a value according to whether the state is true or false. The syntax for OR, XOR, or AND is:

operand .logical operator. operand

.NOT. is a binary operator that inverts the bits of the byte or the word operand. The syntax of NOT is:

.NOT.operand

The logical operators are as follows.

NOT	Logical NOT. A binary operator that inverts the bits of the operand. It sets a bit if that bit position in the operand is clear, and it clears a bit if that position in the operand is set.
AND	Logical AND. Compares the bit positions of two operands and returns the result to the destination operand. It sets a bit if both corresponding bits of the original operands are set. It clears the bits if either one or both are clear in the original operands. Both operands must be treated as an unsigned number.
OR	Logical OR. Compares the bit positions of two operands and returns the result to the destination operand. It sets a bit in the result if either or both of the two corresponding bits of the original operands are set. It clears the bits if neither of the two corresponding bits of the original is set. Both operands must be treated as an unsigned number.

XOR	Exclusive OR. Compares the bit positions of two operands and returns the result to the destination operand. It sets a bit if one of the corresponding bits of the original operands is set and one is clear. It clears the bits if the corresponding bits if the original operands contain the same value. Both operands must be treated as an unsigned number.
-----	---

Table 4-1 illustrates the value of the logical operators when X and Y vary in value from true to false.

Table 4-1. Logical Operations

X	Y	X.AND.Y	X.OR.Y	.NOT.X	X.XOR.Y
TRUE	TRUE	TRUE	TRUE	FALSE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE	TRUE
FALSE	TRUE	FALSE	TRUE	TRUE	TRUE
FALSE	FALSE	FALSE	FALSE	TRUE	FALSE

4.4 ATTRIBUTAL OPERATORS

The attributal operators override an operand's attributes. The operators are SEG, OFFSET, TYPE, SIZE, and LENGTH. They tell what the components of the memory address are and return their values. For this reason, they are often called value returning operators. The six analytic operators are as follows.

SEG	Returns the base address of the segment, label, or variable.
OFFSET	Returns the offset of the variable or label.
TYPE	Returns the number of bytes of a variable.
LENGTH	Returns the number of type units (BYTE, WORD, DWORD, QWORD, TBYTE) allocated in all. If the DUP expression is not present, it returns a 1.
SIZE	Returns the total number of bytes allocated for a variable, which amounts to the product of LENGTH * TYPE.
.TYPE	Returns the mode and indicates whether the expression is external.

4.4.1 SEG

The SEG operator returns the base address of the segment enclosing the label or of the variable that immediately follows the operator. The syntax of SEG is:

```
SEG label  
SEG variable
```

In the following example, the processor returns the segment base address and stores it in AX.

```
MOV AX, SEG VAR1
```

4.4.2 OFFSET

The OFFSET operator returns a constant equal to the offset of the variable or label from the beginning of the segment. The syntax of OFFSET is:

```
OFFSET label  
OFFSET variable
```

In the following example, the processor returns the offset for VAR1 and stores it in AX.

```
MOV AX, OFFSET VAR1
```

The following rules apply to the use of OFFSET.

- OFFSET returns a relocatable value if the label or variable is relocatable.
- OFFSET is unnecessary with DW and DD because the assembler applies an implicit OFFSET to these variables in address expressions.
- When you use an ASSUME with a GROUP directive, OFFSET returns the segment offset unless you use the group versions of the segment override operator. In the following example, you declare GOB to be part of a GROUP, and you want the group offset of GOB.

```
MOV BX, OFFSET DGROUP:GOB
```

4.4.3 TYPE

If the operand is a variable, the TYPE operator returns a value equal to the number of bytes of the variable type as follows.

```

BYTE = 1
WORD = 2
DWORD = 4
QWORD = 8
TBYTE = 10
STRUC = the number of bytes declared by STRUC

```

If the operand is a label, the TYPE operator returns NEAR or FAR. The syntax for TYPE is as follows.

```

TYPE label
TYPE variable

```

In the following example, the TYPE operator returns the number of bytes of VAR1 to AL.

```

MOV AL, TYPE VAR1

```

4.4.4 LENGTH

The LENGTH operator returns the number of type units (BYTE, WORD, DWORD, QWORD, TBYTE) for the variable of its argument. LENGTH accepts only one variable as its argument. You can use it for setting a counter for a loop that accesses the elements in an array. The syntax of LENGTH is:

```

LENGTH variable

```

If you define the variable with a DUP expression, LENGTH returns the number of type units duplicated, that is, the number that precedes the first DUP in the expressions. For example, the length returned in the following example is 100.

```

VAR3 DW 100 DUP(1)

```

If you do not define the variable with a DUP expression, LENGTH returns a 1.

In the following example, VAR1 is an array with 100 elements duplicated only once. Consequently, the LENGTH operator returns 100 and stores 100 in CX.

```

VAR1 DW 100 (DUP)1
MOV CX, LENGTH VAR1

```

4.4.5 SIZE

The SIZE operator returns the total number of bytes allocated for a variable. SIZE is the product of the value of LENGTH times the value of TYPE (SIZE = LENGTH * TYPE). The syntax of SIZE is:

SIZE variable

In the following example, the variable is two bytes long, and the length is 100 (100 * 2 = 200). Thus, the size is 200.

```
VAR1 DW 100 (DUP) 1
MOV BX, SIZE VAR1 ;get total bytes in array
```

4.4.6 .TYPE

TYPE is written with a beginning period to distinguish the .TYPE operator from the TYPE operator. Use the .TYPE operator to gain information about the type of data with which you are dealing. .TYPE informs you of two data characteristics: the mode and whether the data is external. The syntax of .TYPE is:

.TYPE variable

Upon finding the .TYPE directive, the assembler returns a byte of information. The least significant two bits of this byte indicate the mode.

- 0 the mode is Absolute.
- 1 the mode is Program Related.
- 2 the mode is Data Related.
- 3 the mode is undefined.

The most significant bit that the assembler returns to you is the external bit. If the high bit is:

- 0 the expression is not external.
- 1 the expression contains an external.

The defined bit is 20H (the sixth bit from the 0 bit). If the defined bit is:

- 0 the expression is undefined or external.
- 1 the expression is locally defined.

You use the .TYPE operand with macros, external programs that are available for repeated use. Often you must test an argument type to make a decision about how to continue the program. For example:

```

EXCEL    MACRO X
          LOCAL Z
Z        =  .TYPE X
IF       Z

```

In the example, `.TYPE` tests the mode and type of `X` through a conditional directive. Depending upon the evaluation, the assembler either assembles or omits the block of code, beginning with `IF Z`.

4.5. SYNTHETIC OPERATORS

There are six synthetic operators: `PTR`, `THIS`, Segment Override, `SHORT`, `HIGH`, and `LOW`. They build memory-address operands from their components. Synthetic operators are called override operators because they override the segment, offset, type, or distance of variables and labels. The synthetic operators are as follows.

<code>PTR</code>	Overrides the type (BYTE, WORD, DWORD, etc.) or the distance (NEAR or FAR) of an operand.
<code>THIS</code>	Creates an operand with the distance or type you specify.
Segment Override	Overrides the assumed segment of an address expression (a label, variable, or memory operand).
<code>SHORT</code>	Overrides the distance attribute for <code>JMP</code> instructions within 127 bytes.
<code>HIGH</code> or <code>LOW</code>	Isolates the high or low byte of a word.

4.5.1 PTR

The `PTR` operator overrides the type (BYTE, WORD, DWORD) or the distance (NEAR or FAR) of an operand. The `PTR` operator temporarily redefines an expression that is defined in another place. The syntax of the `PTR` operator is as follows.

```

attribute    PTR    expression

```

The attribute is the new type or distance; the expression is the operand whose attribute you wish to override. It can be a variable name, a label name, an address or register expression, or an integer that represents an offset.

In the following example, the PTR operator allows the variable, VAR1, previously defined as a word, to be referenced as a byte for this instruction.

```
VAR1    DW
MOV BYTE PTR VAR1, AL
```

The PTR operator can also resolve ambiguous references. The assembler generates an error when you try to move a byte variable into a word register. If you are not sure of the size of the variable, you can use the PTR to make sure that both the source and the destination are the same size. You can also use PTR to override the type attribute if, for example, you wish to access a word variable as type byte.

4.5.2 THIS

The THIS operator is very similar to the PTR operator. The THIS operator builds a memory address of a specific type without actually leaving any specific memory for it. Through the THIS operator, you create an operand in which you specify the type of distance. The syntax of the THIS operand is as follows.

```
THIS distance
THIS type
```

THIS distance creates an operand with the distance attribute you specify, an offset equal to the current location counter, and the segment base address of the enclosing segment.

THIS type creates an operand with the type attribute you specify, an offset equal to the current location counter, and the segment base address of the enclosing segment.

The following examples illustrate the use of the THIS operators. In the first example, you define TAG as a byte. In the second example, you define SPOTCHECK as NEAR.

```
TAG EQU THIS BYTE
SPOTCHECK = THIS NEAR
```

4.5.3. Segment Override

The assembler gets segment address information from two sources. The ASSUME statement indicates the address of each item of data; default registers provide addresses for anonymous references with no variable names. The default registers are the current CS for labels and the current DS or ES registers for variables. To override the default registers, you need the segment override operator for forward references.

The segment override operator is a colon. The segment override indicates to the assembler that no matter what instructions it previously received concerning the segment of a variable, label, memory operand, or expression, it must use the segment register that precedes the colon. A segment override serves as an ASSUME statement for that command only.

You can use the segment override operator with segment register names (CS,DS,SS,ES), with segment names, or with group names. The syntax of the segment override expression can be any of the following.

```
Segment register: address expression
Segment name:    address expression
Group name:      address expression
```

The segment register named in the format statement is one of the four segment registers; the segment name is a name defined in the SEGMENT directive; and the group name is a name defined in the GROUP directive. The address expression can be a label, a variable, or a memory operand.

In the following examples, VAR2 becomes associated with the extra segment register, and FAR_LABEL becomes associated with segment name, CSEG.

```
MOV AX, ES:VAR2
MOV CSEG:FAR_LABEL, AX
```

4.5.4 SHORT

The SHORT operator overrides FAR distance attributes of labels used as targets for the JMP instruction. The SHORT operator tells the assembler that the distance between the JMP statement and the label specified is within than 127 bytes.

The major advantage of using the SHORT operator is to save one byte. When a label is forward referenced, it carries a 2-byte pointer. Because a label within a range of 256 bytes requires only a single-byte pointer, the SHORT operator saves the extra byte. If the label is not within 256 bytes, however, an error occurs. The syntax of the SHORT operator is:

```
SHORT label
```

In the following example, the label REPEAT is less than 127 bytes away from the SHORT operator.

```
JMP SHORT REPEAT
.
.
.
REPEAT:
```

4.5.5 HIGH and LOW

The HIGH and LOW operators provide 8080 assembly language compatibility. HIGH and LOW are byte isolation operators. HIGH isolates the high 8 bits of an absolute 16-bit value or address expression while LOW isolates the low 8 bits. The syntax of the HIGH and LOW operators is as follows.

```
HIGH expression
LOW expression
```

In the following commands, the high byte of WORD_VALUE is moved to the high order byte of register AH; in the second command, the low order byte of VAR_1 is moved to the low byte of register AL.

```
MOV AH, HIGH WORD_VALUE
MOV AL, LOW VAR_1
```

4.6 RECORD SPECIFIC OPERATORS

All of the record specific operators isolate record fields, consecutive bits within a word or byte. The record specific operators are as follows.

Shift Count	Counts the number of bits from the low end of the record to the low end of the field.
WIDTH	Counts the width of the field or record in bits.
MASK	Determines the maximum value of the field (if all the bits in it are on and if all the bits in the other fields of the word are off).

Before you can use record specific operators, you must create a record by using the RECORD directive. A record is 1 to 16 bits long, and it is defined by fields that are also 1 to 16 bits long. To create a record defined by fields, you need the RECORD directive, which breaks a word up into fields used with any of the record specific operators. For information concerning the record directive, refer to Chapter 5, Data Directives.

4.6.1 SHIFT COUNT

Once you have created a record, you can use the record specific operators. The shift count operator helps you to know where each field begins. The shift count tells you how many bits the lowest bit of a field is from the the lowest bit in the byte or word. The shift count is the number of bits the field must be right shifted for the lowest bit of the field to be set in the lowest bit of the record byte.

For example, VAR2 is a 16-bit record that contains three fields: FIELD1, FIELD2, and FIELD3. The record is diagrammed in Figure 4-1.

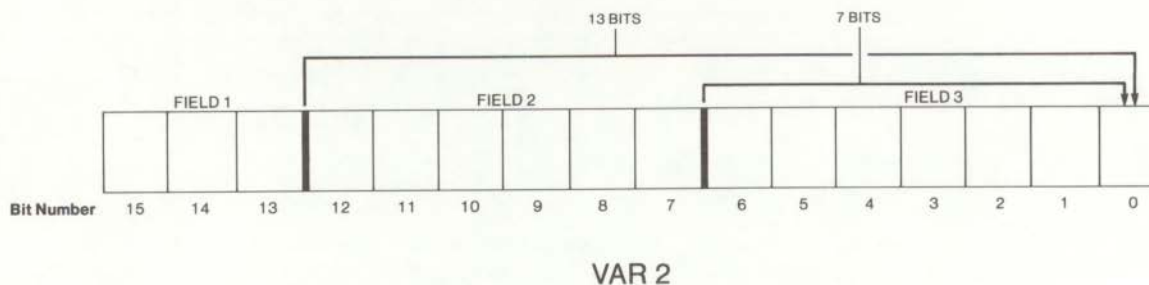


Figure 4-1. The Shift Count of a Record

From the lowest bit in FIELD1 to the lowest bit of the whole word is 13 bits. Therefore, FIELD1 has a shift count of 13. From the lowest bit in FIELD2 to the lowest bit of the whole word is 7 bits. Therefore, FIELD2 has a shift count of 7. The lowest bit of FIELD3 is the lowest bit of the word. Therefore, the shift count of FIELD3 is 0.

The syntax of shift count is:

record field name

When you want to isolate the value in one of these fields, you enter its name as an operand. In the following example, you move FIELD2 into CL, thus isolating its value.

```
MOV DX, VAR2
MOV CL, FIELD2
```

4.6.2 WIDTH

You can use the WIDTH operator with either the field name or the record name. When you use the WIDTH operator with the field name, the assembler returns the number of bits in that particular field. When you use the WIDTH operator with a whole record, the assembler returns the number of bits in the record.

The syntax of WIDTH is:

```
WIDTH record field name
WIDTH record
```


Figure 4-2 illustrates the width of the fields within the record. In this case, the WIDTH of the record is 16 bits, one word.

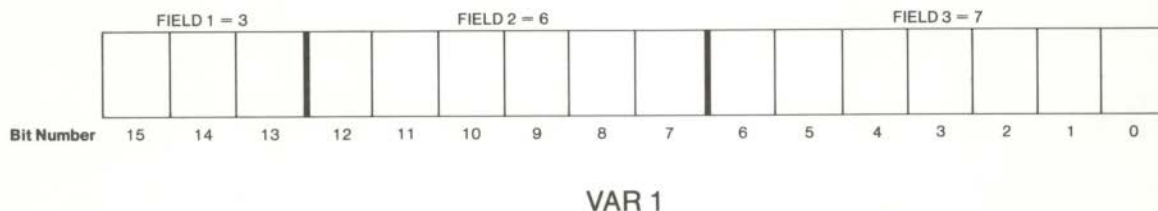


Figure 4-2. The Width of a Record Field

The WIDTH of FIELD1 equals 3; the WIDTH of FIELD2 equals 6; and the WIDTH of FIELD3 equals 7. In the following example, the command moves the WIDTH of FIELD2 into the low byte of the C register.

```
MOV CL, WIDTH FIELD2.
```

4.6.3 MASK

The MASK operand isolates a field and turns on all of the bits in that field. In other words, the value that the field takes on after being masked is the maximum value the bits in that position can have. All of the other bits in the record are turned off and have a zero value.

In Figure 4-3, the byte variable VAR3 is composed of two fields: FIELD1 and FIELD2. FIELD2 is masked. Consequently, all of the bits in FIELD1 are set, and all of the bits in FIELD2 are clear. The MASK of FIELD2 equals 15, or the maximum number that bits 0, 1, 2, and 3 can equal in binary notation if they are all set.

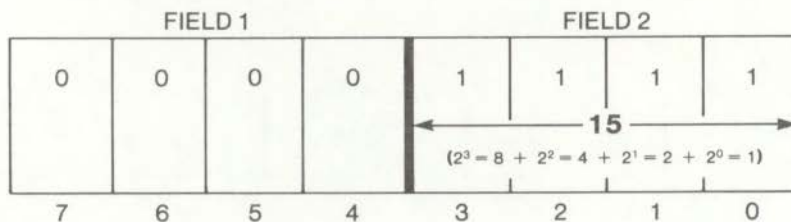


Figure 4-3. The Mask of a Field

The syntax of the MASK operator is:

MASK record-field name

The following operation isolates FIELD2, turns on all of its bits, and AND's it with bits 0, 1, 2, and 3 in VAR2.

```
MOV DL, VAR2
AND DL, MASK FIELD2
```

4.7 ORDER OF PRECEDENCE FOR OPERATORS

As in algebra, there is an order of precedence in the way that expressions are evaluated. The processor evaluates the operations of higher precedence first; it evaluates operations of equal precedence from left to right. Parentheses alter the order of precedence.

The following list contains the order of precedence from highest precedence to the lowest. All operators in a single item have the same precedence.

1. LENGTH, SIZE, WIDTH, MASK
 Items inside parenthesis, angle brackets, or square brackets
 Structure variable operand: variable.field
2. Segment override operator

3. PTR, OFFSET, SEG, TYPE, THIS
4. HIGH, LOW
5. *, /, MOD, SHL, SHR
6. +, - (both unary and binary)
7. EQ, NE, LT, LE, GT, GE
8. Logical NOT
9. Logical AND
10. Logical OR, XOR
11. SHORT, .TYPE

CHAPTER 5

DATA DIRECTIVES

Data directives define and initialize data by type and value. Through the data directives, you can reserve space for data, assign value to data, set and redefine constants and labels, create data structures, define the attributes of a name, change the default radix of numbers, and change the value of the location counter. The data directives are as follows.

ASSUME

The Data Allocation Statement:

DB
DD
DQ
DT
DW

EQU

=

EVEN

INCLUDE

LABEL

NAME

ORG

PROC...ENDP

.RADIX

RECORD

STRUC ...ENDS

5.1 THE ASSUME DIRECTIVE

The ASSUME statement tells the assembler which logical segment (or group) a segment register points to. You can enter from one to four register names in the ASSUME statement. The syntax of the ASSUME statement is as follows.

```
ASSUME segment register:segment name [,segment register:segment name]...
```

The segment names can be:

- The name of a segment declared in the SEGMENT directive.
- The name of a group declared with the GROUP directive.
- An expression: either SEG variable name or SEG label name.
- NOTHING.

In the following example, the ASSUME statement tells the assembler that it should address DATA1 with offsets from DS, CGROUP with offsets from CS, STACK1 with offsets from the SS, and that there should be no address in ES.

```
ASSUME DS:DATA1, CS:CGROUP, SS:STACK1, ES:NOTHING
```

NOTHING tells the assembler that that particular segment register points to NOTHING. Each time you access a variable through that register, you must use a segment prefix. NOTHING is the default value. For this reason, it is important to use the ASSUME statement before you attempt to access any memory instructions.

If you do not use an ASSUME statement or an ASSUME NOTHING, the assembler cannot find the location of data unless you use a register prefix for each variable, symbol, or label in your program.

In the following example, the prefixes DS: and ES: indicate that the assembler should address VAR1 with offsets from the DS register and VAR2 with offsets from the ES register in this one instance only.

```
DS:VAR1
ES:VAR2
```

5.2 DATA ALLOCATION STATEMENT

The data allocation directive tells the assembler how much space to allocate for data, and what the initial value of the variable is.

There are five data types, as Table 5-1 illustrates.

Table 5-1. The Five Data Types

Data Type	Size in Bytes	Meaning of Symbol
DB	1	Byte
DW	2	Word
DD	4	Double word
DQ	8	Quadruple word
DT	10	Ten bytes

The syntax of the data allocation directive is as follows.

```
[variable name] DB (or DW, DD, DQ, DT) initial value [,initial value]
```

The field following the DB specifies how many bytes, words, double words, quadruple words, or ten bytes that the assembler should reserve and what their initial values should be. There are many kinds of statements that can follow the DB. They include:

- A constant or expression. For example, the following statement allocates space for one byte of data and initializes it with the value of 4.

```
DB 2+2
```

- A question mark, used for reserving space without initializing it to any particular value. For example, the following statement reserves 10 bytes of data with no particular value.

```
DT ?
```

- An address expression (for DW and DD only). For example, the word "TABLE" in the following example is a label representing an address.

```
DB PTR DD TABLE
```

- An ASCII string that initializes consecutive bytes in a DB statement. In this case, the characters are stored sequentially with the first character assigned to the lowest address. For example, the following statement reserves 14 bytes for 'MARK MIKE PAUL'.

```
MULT_CHAR DB 'MARC MIKE PAUL'
```

- The expression DUP plus an expression in parentheses. DUP indicates that the assembler should initialize each of the memory locations in the data block to the value in parentheses. In the following example, you request 200 words initialized to alternating 0's and 1's.

```
DW 100 DUP (0, 1)
```

- A number of constants. These constants initialize as many memory locations as are required with the stated value. In the following example, the statement initializes three bytes with the values 1, 2, and 3.

```
DB 1, 2, 3
```

- A combination of the preceding. The following example shows four words initialized with 2, 4, an undetermined value, and 02FH hexadecimal.

```
DB 2, 4, ?, 02FH
```

5.3 EQU

EQU equates the names of symbols with constants, expressions, mnemonics, addresses, or registers. There are a number of rules regarding the use of EQU.

- The expression that follows EQU cannot be external.
- Once you use EQU to define a name, you cannot redefine the name.
- You cannot use EQU within a structure.
- The name cannot have a value before you use EQU to define it. If the symbol already has a value, you use the equal sign (=) to redefine it.

The syntax of EQU is:

```
symbol EQU expression
```

Through EQU, the symbolic name becomes another name for the symbol in the expression. The expression can be:

- a name of a symbol
- an index reference
- an instruction name
- a record expression
- a 16-bit absolute constant
- a floating-point value
- a segment prefix and operands

In the following examples, SYMB1 is equated with the constant 50, and SYMB2 is equated with an index reference, the base pointer plus 4.

```
SYMB1 EQU 50
SYMB2 EQU [BP+4]
```

5.4 EQUAL SIGNS

You can use equal signs to define and redefine an expression. The main difference between the equal signs and EQU is that equal signs allow you to redefine a label without causing an assembly error. The syntax is:

```
symbol = expression
```

The expression must be a valid expression. In the following example, SYMB becomes equated with 50.

```
SYMB = 50
```


5.5 The EVEN DIRECTIVE

It is often necessary to use even boundaries to address an element so that it begins on a word boundary. The EVEN directive causes the assembler to assign an even address to an instruction. If the instruction already has an even address, the assembler ignores the EVEN directive. If you use EVEN with a byte-aligned segment, the assembler issues an error message.

The syntax of the EVEN directive is:

EVEN

5.6 THE INCLUDE DIRECTIVE

The INCLUDE directive inserts a source code from an alternate assembly source file into the current source file. The INCLUDE directive allows you to repeat frequently used code. The syntax of the INCLUDE directive is as follows.

INCLUDE file name

The file name is any valid file specification. If the device designation is different than the currently logged drive or device, the source file name specification must include it.

When the assembler finds the INCLUDE directive, it searches for the file named and assembles it into the current program immediately following the INCLUDE directive. When the assembler reaches the end of the file, it resumes assembling the current program. If the assembler cannot find a program with the name indicated in the INCLUDE statement, it sends out an error message and the assembly aborts.

Nested INCLUDES are legal. The file inserted with an INCLUDE statement can contain an INCLUDE statement of its own. This is not a recommended procedure because of the amount of memory taken up when you nest INCLUDE statements.

In the following example, the assembler includes the file RECORD.TST in the current source file.

INCLUDE RECORD.TST

5.7 THE LABEL DIRECTIVE

The LABEL directive causes the assembler to associate the current segment offset with a label name. The attributes of name are segment, offset, and type. The syntax of LABEL is as follows.

```
label name LABEL type
```

You can use the label name for either code or data. The type entry must be BYTE, WORD, DWORD, structure-name, or record-name for data areas. When you use a LABEL for code, you must address it through the CS register. The type must be NEAR or FAR for executable code. In the following examples, VAR1 is NEAR while VAR2 is FAR.

```
VAR1 LABEL NEAR
VAR2 LABEL FAR
```

5.8 THE NAME DIRECTIVE

The NAME directive gives a module a name. Through the NAME directive, you can make a name represent code, data, or a constant. The name field can begin in any column although normally names begin in column one. The format of the NAME directive is:

```
NAME name of module
```

You cannot use a reserved word as the name of the module.

Each module has a name. If you do not use the NAME directive, the assembler uses the first six characters of the TITLE statement for the module name if they are legal. If you do not use the TITLE statement, or if the first six characters of the TITLE statement are not legal, the assembler uses the source file name as the module name. Use one of the following formats to make a name represent code.

```
NAME: directive, instruction, or nothing
NAME LABEL NEAR or FAR
EXTRN NAME: NEAR or FAR
```

Use one of the following formats to make a name represent data.

```
NAME LABEL size
NAME expression
EXTRN NAME: size
```

The size refers to the data type, BYTE, WORD, etc. Use one of the following formats to make a name represent a constant.

```
NAME EQU constant
NAME = constant
NAME SEGMENT attributes
NAME GROUP segment names
```

For further information about any of the individual directives attached to NAME, refer to each directive separately, i.e., EQU, SEGMENT, GROUP etc.

5.9 THE ORG DIRECTIVE

Through the ORG directive, you can reset the location counter to locate code or data at a particular offset in the segment. The syntax of the ORG directive is as follows.

```
ORG expression
```

The expression must evaluate to a two-byte absolute number. In the first of the following examples, the expression following ORG evaluates to 120H. In the second example, the dollar sign represents the current value of the location counter. The instruction causes the location counter to move 2 bytes forward.

```
ORG 120H
ORG $+2
```

5.10 PROCEDURES

A procedure is a sequence of instructions. Through the procedure, you can organize code by dividing it into blocks, each of which performs a distinct function. Although you can JMP into and out of a procedure, a procedure usually receives control through a CALL instruction and passes control back through a RET instruction. Two statements, PROC and ENDP, indicate the entry point and exit point of the procedure. The format of a procedure is as follows.

```

procedure name  PROC [NEAR/FAR]
    .
    .
    .
    RET
    .
    .
procedure name  ENDP

```

The procedure name labels the beginning address of the procedure. NEAR or FAR tells the assembler which CALL instruction to use. If the procedure is within 64K bytes of the CALL, the assembler saves only the offset in the IP on the stack. If the procedure is farther away than 64K bytes, the assembler saves two addresses: the offset in the IP and the segment address from the CS.

Just as there are two kinds of calls, there are two kinds of return instructions: NEAR and FAR. The NEAR restores the value of the original offset, and the FAR restores the values of the base address and the offset.

If you don't specify either FAR or NEAR, the default is NEAR. You use FAR if the procedure name is an operating system entry point or if you call the procedure from code that has another ASSUME CS value.

Combining the PUBLIC directive with the PROC statement permits you to make external calls or other external references to the procedure. You can call a procedure through either a JMP or CALL. You can also nest procedures. The following is an example of a simple procedure that adds two byte values together.

```

ADD_UP  PROC    NEAR
        MOV DH, AH
        ADD DH, BH
        RET
ADD_UP  ENDP

```

Parameters are the input values passed on to procedures. For example, in the preceding example, the procedure receives input values in AH and BH. There are three ways of passing parameters to a procedure: by placing values in registers as previously illustrated by making values in a variable in the data region available to the procedure, and by putting the values onto the stack and allowing the procedure to access them.

5.11 THE .RADIX DIRECTIVE

The .RADIX directive permits you to change the input radix from the default of 10 to any base in the range of 2 to 16 for types DB and DW. The syntax of the .RADIX directive is as follows.

```
.RADIX expression
```

The expression is always in decimal radix, regardless of the current input radix. The default input radix for all constants is decimal. The following directive causes the assembler to interpret all numbers that come after it as being in octal notation. Number 24 is equivalent to Decimal 20.

```
.RADIX 8
MOV BX, 24
```

The .RADIX directive does not affect variables defined as type DD, DQ, or DT. The assembler evaluates numeric values of this data type as decimal unless you append a data type suffix to the value. The .RADIX directive does not affect the values in the .OBJ, .LST, or .CRF output files.

5.12 THE RECORD DIRECTIVE

Before you can use record specific operators, you must create a record. A record is 1 to 16 bits long, made up of fields from 1 to 16 bits long. To create a record defined by fields, you need the RECORD directive, which breaks a word up into fields used with any of the record specific operators. The syntax of the RECORD directive is as follows.

```
record name RECORD field name: width [=expression],[...]
```

When you use field name as a value in an expression, its value is the shift count to move the field to the far right. Using the MASK operator with the field name returns a bit mask for that field.

Width is a constant in the range of 1-16 that specifies the number of bits in the field. If the total width of all declared fields is larger than 8 bits, the assembler uses 2 bytes; otherwise, it uses 1 byte.

The expression contains the initial or default value for the field. If the field is 7 or more bits wide, you can use an ASCII character as the expression. For example:

```
HIGH:7='Q'
```

In the following example, a record named REC1 is broken up into three records: HIGH, which is 4 bits long and initialized at 1; MID, which is 3 bits long and initialized at 2; and LOW, which is also 3 bits long and initialized at 5.

```
REC1    RECORD  HIGH:4=1, MID:3=2, LOW:3=5
```

The first field uses the most significant bits of the record. Subsequently declared fields use the succeeding bits to the right. If the fields do not total exactly 8 bits or 16 bits, the entire record is shifted to the right so that the last bit of the last field is the lowest bit of the record. Unused bits remain at the high end of the record.

Figure 5-1 illustrates the placement of bits for the following RECORD directive after the processor shifts the fields to the right.

```
REC2    RECORD  HIGH:5, MID:4, LOW:1
```

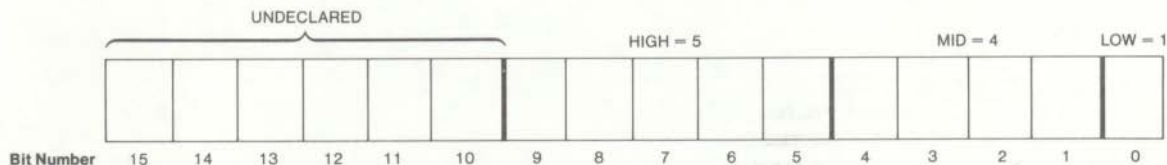


Figure 5-1. The Placement of Bits in a Record

To initialize records after the first RECORD directive, use a format similar to that of the data allocation statement.

```
[name] record name <[expression]  [,...]> or
[name] record name  repeat value  DUP([<expression> [,...]>)]
```

The name is optional. If you indicate a name, it refers to a label for the first byte or word of the record storage area. The record name is the name of the record in the RECORD directive.

The expression contains the values with which you want to initialize the record fields. Parentheses are required around the second expression following DUP. If you leave the expression blank, the value that you gave the field in the original RECORD directive applies. If no such value exists, the value is indeterminate. In the following example, RECNAME takes on the value 5.

```
REC1 RECNAME <5>
```

If you want to reinitialize fields, name the record and specify the value that you want the field to have. Commas indicate that you want the field to retain its present value. In the following example, the HIGH and the MID fields retain their indeterminate values while the LOW field becomes valued at 7.

```
REC1 <,,7>
```

You can use records in expressions as operands in the following form.

```
record name [value[,...]]
```

The value entry is the value you wish to place in the record field. The value entry is optional, but the angle brackets must appear even if no values are within them. In the following instruction, you use the field MID as an operand, masking it to set the like bits in DX.

```
AND DX, MASK MID
```

5.13 The STRUCTURE DIRECTIVE

A structure is a group of byte and word offsets that you can reference from any base. The STRUC directive is a very specialized type of data allocation statement. The assembler allocates no storage to the structure. The structure becomes associated with an area of memory and a base value when you reference it in an instruction.

At times, you may wish to define a complex type of data. For example, you may wish to define the dimensions of a cube and compose a data structure in which you can call up each dimension. You can define a data structure using the data allocation statement coupled with the STRUC directive.

The syntax of the STRUC directive is as follows.

```

Structure name      STRUC
                    .
                    .
                    .
[variable name] DB or DW or DD or DQ or DT Expression
                  Expression DUP expression [...]
```

Structure name ENDS

The following structure creates the dimensions of a cube.

```

CUBE      STRUCT
    L      DW  ?
    W      DW  ?
    H      DW  ?
CUBE      ENDS
```

You can either define the value of the variables within the STRUC as you create them, or you can define them afterward. The preceding structure does not define the value of the variables. If you do not define the variables within the structure, you must include a definition like the following one that gives the values of 1, 2, and 3 to L, W, and H respectively.

```
SQUARE CUBE <1,2,3>
```

The syntax of the statement that initializes and allocates structures after you have declared them is:

```
[name] structure name <expression, [...]>
```

The name refers to an optional name for the bytes that you defined in the structure. The structure name is the name you assigned from the STRUC/ENDS directives. The expression specifies the list of initializing or optional override values.

You can define the data types within the structure as two sorts: overridable or not overridable. You can redefine overridable types; you cannot redefine those that are not overridable.

A simple field with only one entry is overridable; a field with more than one entry or with a DUP expression is not overridable. Strings can be overridden by other strings. However, if they are not the same length, the assembler either pads or truncates the overriding string, depending upon whether it is shorter or longer than the original string.

The following structure presents two overridable fields and two fields that cannot be overridden.

```
S      STRUCT
FIELD1 DB 1,2; not overridable
FIELD2 DB 10 DUP(?) not overridable
FIELD3 DB 5; overridable
FIELD4 DB 'JOHN DOE'; overridable
S      ENDS
```

To override the value of the two overridable fields, FIELD3 and FIELD4, use the following statement.

```
NEW S  <,,7,'BOB HALLER'>
```

The commas command the assembler to leave FIELD1 and FIELD2 with their original values; the assembler replaces the 5 of FIELD3 with a 7, and the string 'JOHN DOE' of FIELD4 with 'BOB HALLER'. The assembler truncates the last two letters of HALLER because the overriding string is two characters longer than the original string.

The STRUC directive allows you to access any field within the variable through the special dot (.) operator. The format of structure operands is as follows.

```
variable.field
```

The variable is the user name given in the structure initialization statement. It can be anonymous, such as the name of an indexed memory operand. The field name is the user name given in the data allocation statement within the structure. The value of the field entry is the offset within the structure.

In the following example, you set up a structure, initialize GIRAFFE with the value 16 using the name LONG_NECK, and then use LONG_NECK.GIRAFFE as a structure operand.

```
ZOO      STRUC
GIRAFFE DB ?
ZEBRA DB ?
GAZELLE DB ?
ZOO      ENDS

LONG_NECK  ZOO <16, 5, 7>

MOV AL, LONG_NECK.GIRAFFE
```



CHAPTER 6

USING PROGRAM MODULES AND FUNCTIONS

The Wang PC assembly language is a flexible programming language. With it, you can combine separately assembled program modules into a single program. Your ability to combine different modules into consecutive memory locations allows you to use easily manageable program units that you can test and debug individually. You can also use high-level languages in modules and link them to your assembly language program.

There are many assembly directives especially designed to make elements of a module available to other modules. During assembly, the assembler allots space to these elements. All of the modules, however, are assembled separately. The Linker resolves all of the references and creates one file out of the separate modules. Each module has a name. To name a module, use the NAME directive described in Section 5.8.

6.1 USING THE SAME SYMBOLS IN MULTIPLE MODULES

The assembler codes local symbols with only an offset address, but it codes external symbols with an offset and a base address. You should distinguish local symbols from external symbols so that the assembler can code the symbols properly. The PUBLIC and EXTRN directives allow a symbol to be used in more than one module.

6.1.1 The PUBLIC Directive

The PUBLIC directive informs the assembler that the symbol that follows PUBLIC is available for use in other modules. The syntax of the PUBLIC statement is as follows.

```
PUBLIC symbol [,symbol]...
```

The symbol can be a number, a variable, or a label. It can be a label defined with the PROC directive. It cannot be a register name or a symbol defined with EQU by floating point numbers or by integers larger than 2 bytes. In the following example, you declare VAR1 to be PUBLIC.

```
VAR1 DW 50
PUBLIC VAR1
```

6.1.2 The EXTRN Statement

Once you define a symbol and declare it to be PUBLIC, you can use it in another module. To access the symbol from another module, use another special statement, the EXTRN statement. The EXTRN statement calls upon the PUBLIC symbol from another module. The syntax of the EXTRN statement is as follows.

```
EXTRN symbol:type [,symbol:type]
```

Symbol refers to a symbol that you define and declare to be PUBLIC in another module. Type can be any of the following types, but it must be valid for the symbol.

- BYTE, WORD, or DWORD
- NEAR or FAR for labels or procedures defined with a PROC directive
- ABS for pure numbers whose implicit size is word but includes byte

For example, if you wanted to call upon VAR1 from the first module, you would write:

```
EXTRN VAR1:BYTE
```

You must define the type, which is byte, when you use the EXTRN statement but not when you use the PUBLIC statement. You defined the variable type in the same module in which you used the PUBLIC statement so you do not need to redefine it.

Placement of the EXTRN statement affects assembly. When you place EXTRN inside a SEGMENT/ENDS pair, the assembler assumes that the external symbol belongs to that logical segment. Therefore, the assembler associates the segment name with the symbol and the segment base address. If EXTRN is outside a SEGMENT/ENDS pair, the assembler assumes that the symbol belongs to none of the logical segments of that module.

To help the Linker locate the address of an EXTRN symbol if you do not know it, you can use an ASSUME statement with a SEG operand or an explicit segment prefix. The following example returns the segment value of VAR1.

```
ASSUME DS:SEG VAR1
```

The Linker tries to resolve incorrectly defined or undefined symbols. It takes the offset relative to a given statement and searches for the symbol definition. If the definition is more than 64K bytes away, the linker cannot find it and returns an error message.

6.2 COMBINING SEGMENTS

To save processing time, to clarify your program, and to save space, you should combine the logical segments of two program modules whose total size is less than 64K bytes. When you combine the segments, the assembler addresses both segments through the same segment register.

To combine corresponding logical segments from two modules, you must use the same name for both segments or use different segment names but GROUP directives with the same group name. For example, if you name the data segment in the first module FUNCT_DATA, you must name the data segment in the second module FUNCT_DATA.

To use the SEGMENT statement for the combination of modules, you need to indicate the attributes of the SEGMENT in the modules that you combine. The syntax of this SEGMENT statement differs from the simple SEGMENT declarative statement. The syntax of the SEGMENT statement that combines logical segments from different modules is as follows.

```
segment name SEGMENT align type, combine type, 'class name'
```

6.2.1 Segment Name

The segment name is any legal name that you give the segment. Make sure the names of all segments you wish to combine are the same.

6.2.2 Align Type

The alignment tells the linker on what kind of boundary you want the segment to begin. The first address of the segment is determined by one of the following alignment types.

PAGE	The low byte of the address is 0.
PARA	The low nibble of the address is 0.
WORD	The low bit of the address is 0; the word is aligned on an even boundary.
BYTE	The address can occur on either an even or an odd boundary.

PARA, an address divisible by 16, is the default. You should avoid using the BYTE align type because it is slow and often results in boundary errors.

6.2.3 Combine Type

The combine type can be any of the following.

PUBLIC	Combines data segments or code segments with common base addresses.
STACK	Combines stack segments with common base addresses.
MEMORY	Causes segments to be placed at highest available segments in memory.
AT paragraph address	Permits labels and variables to be defined at fixed offsets within fixed areas of storage; places segment at the specified paragraph address.
COMMON	Overlaps loaded segments so that the length of the common area is the length of the longest segment.
no entry	Creates private segments that may share names, but each has its own base address.

The PUBLIC Combine Type

PUBLIC and STACK are the two most common combine types. Use the PUBLIC combine type for segments that are addressed from the same segment register and that occupy adjacent memory units. PUBLIC allows code and data segments from different modules to be combined.

Because of the PUBLIC combine type, when the linker links the following modules, all of the variables in the modules are addressed from the same data segment address.

```

MODULE_1

DATA_SEG    SEGMENT          PUBLIC
      DATA_A    DB          "A,B,C,D"
DATA_SEG    ENDS

MODULE_2

DATA_SEG    SEGMENT          PUBLIC
      DATA_B    DB          1,2,3
      DATA_C    DB          ?
DATA_SEG    ENDS

```

Stack Combine Type

To combine stacks from two different modules, use the STACK combine type. When the linker finds the STACK combine type, it links separate stack segments from separate stack modules. This new stack region is as large as the combined space of all the separate stack regions from all the separate modules.

In the following program, the LABEL directive addresses the top of the stack. The name of the stack segment in each module is identical. The linker links these separate modules into one segment addressed from one stack segment register.

```

MODULE_1

DATA_STACK  SEGMENT          STACK
      DB      10    DUP(?)
TOP_S       LABEL            BYTE

MODULE_2

DATA_STACK  SEGMENT          STACK
      DB      25    DUP(?)
TOP_STACK   LABEL            BYTE

```


6.2.4 The Class Name

The class name indicates that the assembler should place certain previously combined regions next to each other in memory. The assembler does not address them from a common base, however. In the following example, the same class name causes the assembler to place the two segments in adjacent areas of memory.

```
DATA_STACK SEGMENT WORD STACK 'A_REGION'
DATA_SEG SEGMENT WORD PUBLIC 'A_REGION'
```

The use of the class name 'A_REGION' within single quotations causes the linker to place the two segments adjacent to one another in memory.

6.3 USING GROUPS

A group is a set of segments that the assembler addresses from the same segment register. The segments share a base address. When the total of all the segments occupy less than 64K bytes of memory, you can combine them to make your program more efficient. When you use groups, you can use the offset alone to address each item; furthermore, one register contains the address of all of the segments.

You can use a group name:

- as an immediate value, e.g., MOV AX, DGROUP;
- in an ASSUME statement, e.g., ASSUME DS:DGROUP; or
- as an operand prefix to combine segments, e.g., DW DGROUP:VAR1.

The syntax of the group statement is as follows.

```
group name GROUP segment name [,segment name]...
```

The segment name can be:

- assigned by a SEGMENT directive and it can be a forward reference, or
- an expression: SEG variable or SEG label.

The following group statement combines the code, data, and stack segments into one group, addressable from the address of the label.


```
WHOLE_GROUP  GROUP  D_SEG, C_SEG, S_SEG
```

Once you define a group, you must load its address into a segment register. In the following example, the processor loads the group address into the DS register.

```
MOV AX, WHOLE_GROUP  
MOV DS, AX
```

You must also identify the group address to the assembler as follows.

```
ASSUME DS:WHOLE_GROUP
```

The offset of a variable or label from its group base address is different from the offset from its segment base address. The OFFSET operator returns the OFFSET from the segment even though the variable is grouped, unless you use a group override. The syntax of the group override is as follows.

```
OFFSET  group name:symbol
```

For example, if VAR1 were a variable in SEG1, and you wish to get the offset address from the group of which it is a part, use the following statement.

```
OFFSET  WHOLE_GROUP:VAR1
```


CHAPTER 7

MACRO AND REPEAT BLOCKS

7.1 MACROS

A macro is a block that you code once and repeat within the module without recoding. A macro is a procedure that you can repeat in a variety of modules. Once coded, you can invoke a macro at appropriate places in the module, causing its expansion.

7.1.1 Creating a Macro

The first step in using macros within your program is to write the macro code. A macro definition associates a name with a block of instructions which may use dummy arguments as place holders. The assembler replaces these dummy arguments with actual parameters. The number of dummy arguments is only limited by the length of the macro statement line.

The macro name and the dummy arguments are formed according to the rules for any other symbolic name. Each macro begins with a MACRO statement and ends with an ENDM statement. Commas separate the dummy arguments. The syntax of a macro is shown as follows.

```
macro name      MACRO      [dummy,...]
                  .
                  .
                  .
                  .
                  ENDM
```

The following is an example of a small macro that adds two numbers together and moves them to a third location. This operation requires three dummy arguments.

```
ADD1      MACRO XX,YY,ZZ
           PUSH AX
           MOV AX,  XX
           ADD AX,  YY
           MOV ZZ,  AX
           POP AX
           ENDM
```

If you don't pass any parameters to the macro block, you can use a macro call with no dummy arguments. The syntax of the parameterless macro statement is as follows.

```
macro name MACRO
```

7.1.2 Invoking a Macro

After you code your macro, you can invoke it. When the assembler encounters a macro call, it expands the macro call statement by bringing in and assembling the appropriate macro block at that spot within the program unit. To invoke a macro, enter its name and parameters into your program. The syntax of the call statement is as follows.

```
macro name [parameter,...]
```

For example, suppose you wish to use the macro in Subsection 7.1.1., ADD1, in your program. Enter a call using variable names rather than dummy arguments.

```
ADD1    VAR1, VAR2, VAR3
```

When the assembler encounters the macro call, it searches for a macro called ADD1 in the symbol table it builds in the first pass. If it finds a macro of the same name in the table, the assembler expands the code into your program, substituting the actual parameters for the dummy list in corresponding order. In this case, it puts the following code into your program.

```
PUSH AX
MOV AX, VAR1
MOV AX, VAR2
MOV VAR3, AX
POP AX
```

The assembler is very flexible in interpreting the parameters. If the number of parameters in the macro call is greater than the number of dummy arguments in the macro dummies, the assembler ignores the extra actual parameters. If there are fewer parameters in the macro call than there are arguments in the macro, the assembler interprets the extra dummy arguments as being null.

If you place angle brackets around a set of parameters, the assembler interprets them as being just one parameter. In the first of the following two examples, the assembler interprets ADD1 as having three parameters; in the second example, the assembler interprets ADD1 as having only one parameter.

```
ADD1 1,2,3
ADD1 <1,2,3>
```


7.1.3 Deleting a Macro

The PURGE directive deletes macros from the text of your program. Purging a macro is helpful if you need the space for more current macros. After you call up the file using the INCLUDE directive, you can edit out the macros that you do not need through the PURGE directive. In this way, you retain only the macro(s) that you wish to use. The syntax of the PURGE directive is as follows.

```
PURGE      macro name
```

The following example uses the INCLUDE and the PURGE directive to edit out one macro but retains the other macros in the file.

```
INCLUDE    MACRO.LIB
PURGE      MAC1
```

If you invoke MAC1 after using the preceding sequence, the assembler returns a syntax error because MAC1 is not in the file.

7.1.4 The LOCAL Statement

If you wish to repeat code within a macro, you can retain a unique address for each repetition of each symbol through the LOCAL statement. For example, if you want to repeat ADD1 from Subsection 7.1.1 three times and retain the values of VAR1, VAR2, and VAR3 in each repetition, you would use the LOCAL statement to do so. The LOCAL directive must precede all other types of statements in the macro block. The syntax of the LOCAL statement is as follows.

```
LOCAL dummy [,dummy...]
```

The following example shows a macro block in which the LOCAL statement causes each of the variables, A, B, C, D, and E to have new values each time they pass through the loop. The assembler gives each repetition of each variable a unique offset. For example, on the first round A takes the offset 0000 and the label ??0000, the form that each unique label of the variables in the listing file takes. On the second round, A takes the offset 0009 and the label ??0005. In each repetition, the labels retain their unique addresses and values. In the first listing file, the assembler substitutes 0C00H and OBEH for the dummy arguments NUM and Y. In the second repetition, it gives the dummy arguments the parameters of 03C0H and OFFH.

The Macro Block:

```

FUN SEGMENT
    ASSUME CS:FUN,DS:FUN
FOO MACRO  NUM,Y
    LOCAL  A,B,C,D,E
A:  DB  7
B:  DB  8
C:  DB  Y
D:  DW  Y+1
E:  DW  NUM+1
    JMP A
    ENDM
LAZY PROC
FOO      0C00H, OBEH
FOO      030H, 0FFH
    RET
LAZY ENDP

```

The first listing of the listing file:

```

0000      07      +  ??0000:  DB      7
0001      08      +  ??0001:  DB      8
0002      BE      +  ??0002:  DB     OBEH
0003      00BF     +  ??0003:  DW     OBEH+1
0005      0C01     +  ??0004:  DW     0C00H+1
0007      EB F7    +          JMP     ??0000

```

The second listing of the listing file:

```

0009      07      +  ??0005:  DB      7
000A      08      +  ??0006:  DB      8
000B      FF      +  ??0007:  DB     0FFH
000C      0100     +  ??0008:  DW     0FFH+1
000E      03C1     +  ??0009:  DW     03C0H+1
0010      EB F7    +          JMP     ??0005
0012                      FUN     ENDS
                                END

```

7.2 REPEAT BLOCKS

In the Wang PC assembly language, repeat blocks allow you to repeat code automatically. Repeat blocks differ from macro blocks in that macros are part of the code of the program. Repeat blocks, however, can be used inside macro blocks, where they are called up with the macro itself.

The parameters of the macro block are more flexible than those of a repeat block. You can pass parameters to the macro block when you call it; hence, you can change the parameters of a macro. In the repeat block, you assign the parameters as part of the code block. Therefore, you must know the parameters in advance.

7.2.1 Repeat Block

The syntax of the Repeat Block (REPT) is similar to that of a macro. You introduce it by a REPT statement, and it ends with an ENDM as shown.

```
REPT expression
.
.
.
.
ENDM
```

The expression that follows REPT indicates the number of times the processor should repeat the statements between the REPT statement and the ENDM statement. The processor evaluates the expression as a 16-bit unsigned number. The assembler generates an error if the expression that follows REPT contains an external symbol or an undefined operand.

In the following REPT block, the value of X increases by one each time the code repeats. The code repeats ten times as indicated by the statement: REPT 10. Thus, 10 bytes are allocated, with initial values from one to ten.

```
X=0
REPT 10
X=X+1
DB X
ENDM
```

7.2.2 Indefinite Repeat Block

The Indefinite Repeat Block (IRP) is similar to the REPT, but this block repeats for each parameter. The syntax of the IRP is as follows.

```
IRP, dummy, <parameters>
.
.
.
.
ENDM
```

Parameters, enclosed in angle brackets, are any legal symbols, strings, numerics, or character constants. As the Repeat Block executes, each parameter in sequence gives the value to the dummy argument. Each repetition substitutes the next parameter of each dummy in the block. If the parameter is null (<>), the block processes once with a null parameter.

In the following example, the IRP repeats for each parameter 1 through 10. Thus, 10 bytes are allocated, with initial values from 1 to 10.

```
IRP X, <1,2,3,4,5,6,7,8,9,10>
DB      X
ENDM
```

The syntax of the IRP may be slightly different inside a macro block. The dummy argument is a variable that can receive its value from the parameters of the macro call. In the following example, X receives its value from the parameters of the macro call, MAC1.

```
MAC1      MACRO X
IRP      Y, <X>
DB      Y
ENDM
ENDM
```

The parameters of the macro call supplies the value of the X, as shown.

```
MAC1 <1,2,3,4,5,6,7,8,9,10>
```

All the items are passed as a single parameter. When the assembler assembles the IRP, the macro expansion is:

```
IRP Y, <1,2,3,4,5,6,7,8,9,10>
```

7.2.3 Indefinite Repeat Character

The Indefinite Repeat Character (IRPC) is slightly different from the REP or the IRP. The syntax of the IRPC is as follows.

```
IRPC dummy, string
```

The assembler repeats the statements in the block once for each character in the string. As in the IRP, the assembler substitutes the next character in the parameters of the macro call for the dummy argument. In the following example, X takes on the values of 1 to 10.

```
IRPC      X, 0123456789
DB      X+1
```


7.3 SPECIAL MACRO OPERATORS

There are several special operators that you use with macros: the ampersand, angle brackets, the semicolon, the exclamation mark, and the percent sign.

7.3.1 Ampersand

In order to use a quoted string as a dummy parameter, you must place an ampersand (&) immediately before the string. The ampersand is used in macro expansion to concatenate text or symbols. Form a symbol from text and a dummy as follows.

```
ERRGEN  MACRO X
ERROR&X: PUSH BX
ABX  MOV BX, '&X'
AB&X JMP  ERROR
      ENDM
```

When you call ERRGEN A, the assembler generates the following code in your program.

```
ERRORA: PUSH BX
ABX  MOV BL, 'A'
ABA  JMP  ERROR
```

In this example, the assembler substitutes the quoted string 'A' for '&X'.

When you nest macros, you need to supply as many ampersands as there are levels of nesting, as follows.

```
ERRGEN  MACRO X
      IRP  Z, <1,2,3>
X&&Z DB  Z
      ENDM
      ENDM
```

Because the macro includes a repeat block, you need three ampersands. The assembler removes an ampersand each time it repeats the block of code.

7.3.2 Angle Brackets

The Wang PC assembly language treats all text between angle brackets as a single literal. If you put the parameters following the IRP directive inside angle brackets, the assembler interprets them in the following ways.

- As a single parameter, even if you use commas to separate the items in the list.
- As literal characters, not symbols or indicators. For example, if you use a semicolon within angle brackets, the assembler interprets it as a character rather than as an indicator preceding a comment.

When using nested macros, you need to use as many sets of angle brackets around the parameters as there are levels of nesting.

7.3.3. Two Semicolons

The assembler does not expand a comment preceded by two semicolons (;;) in a MACRO or repeat block. .LALL causes the assembler to list all comments not preceded by ;; during macro expansion. .XALL, the default, causes the assembler to disregard comments when expanding macro code.

Listing comments during macro expansion can cause out-of-memory errors. If you are using .LALL, use a double semicolon (;;) before comments, unless you specifically wish to have a comment in the expansion. In the following example, the assembler overlooks the comment that follows the double semicolons.

```
;; This is a comment.
```

7.3.4 Exclamation Mark

An exclamation mark (!) indicates that the next character in an argument is literal. Therefore, !; causes the assembler to interpret the semicolon as a literal rather than as an operator to indicate comments.

7.3.5 Percent Sign

You use the percent sign (%) only in a macro argument to convert the expression that follows it to a number in the current radix. Without the percent sign, the assembler substitutes the text of the macro argument for the symbolic parameter. When the percent sign is present, the assembler replaces the dummy argument for the numeric equivalent. The format of the percent operator is the following.

```
%expression
```

In the following example, the command `STRING %NUMBER` causes the assembler to substitute the value of `NUMBER` (1234) for `NUM` in the concatenated string `'abcd&NUM'`. The result that the assembler generates is `'abcd1234'`.

```
NUMBER EQU 1234
STRING  MACRO    NUM
        DB 'abcd&NUM'
        ENDM
        STRING %NUMBER
```


CHAPTER 9

LISTING DIRECTIVES

There are three types of listing directives: those that control the format of the file, those that serve as programming aids, and those that control what is assembled in the file. The format control directives control page breaks and the appearance of titles, the programming aid directive causes text to be listed on the screen during assembly, while the assembly control directives help you to control the amount of space used in memory.

9.1 FORMAT DIRECTIVES

There are three format directives: PAGE, SUBTTL, and TITLE. They do what their names indicate. The PAGE directive sets up page breaks or the size of the page, the TITLE directive causes a title to appear on each page, and the SUBTITLE directive causes subtitles to appear on each page.

9.1.1 The PAGE Directive

The PAGE directive has two very distinct functions depending upon the initial syntax of the command. PAGE followed by no arguments or with an optional + causes the assembler to start a new output page. The syntax of this PAGE command is either PAGE + or PAGE.

When the assembler encounters the PAGE + command, it puts a form feed character into the listing file at the end of the page; it increments the major page number, i.e., the chapter or unit number; and it resets the minor page number to 1, i.e., the page within the chapter. When the assembler encounters the PAGE command without the +, it puts a form feed character into the listing file at the end of the page and increments the minor portion of the page number.

The PAGE directive followed by length and width arguments sets up either the length or the width of the page or both. The syntax of the command is as follows.

PAGE [length] [width]

If you include the value of length, that value becomes the new page length as measured by the lines of the page. The page length starts at 10 or greater and goes to 255 or less. The default page length is 50 lines per page.

The value of width, if included, becomes the new page width as measured in columns. The page width can be no greater than 132, and the default is 80. The following command, for example, sets the length of the page at 58 lines and the width of the page at 60 columns.

PAGE 58, 60

9.1.2 The TITLE Directive

The TITLE directive directs the assembler to list a title on the first line of each page. The title can be up to 60 characters long. The first six characters of the title are used as the module name unless a NAME directive specifies a different name. The syntax of the TITLE directive is as follows.

TITLE text

For example, the following TITLE directive causes the name PROG1 to appear at the top of every page of the listing. If the NAME directive does not specify otherwise, the module name is now PROG1.

TITLE PROG1

9.1.3 The SUBTTL Directive

The SUBTTL directive causes the assembler to list a subtitle on the line after the title. The subtitle can be up to 60 characters long. The assembler truncates more than 60 characters.

CHAPTER 8

CONDITIONAL DIRECTIVES

There are ten conditional directives in the Wang Professional Computer assembly language: IF, IFE, IF1, IF2, IFDEF, IFNDEF, IFIDN, IFDIF, IFB, IFNB. You can use a conditional directive to test the condition of blocks of code in much the same way that you can in higher level languages. The condition is either true or false. The value of true is any number other than zero; the value of false is zero. If the condition is true, control passes to the next instruction in the conditional block; if it is false, control passes to an ELSE statement, which processes an alternative block of code, or to the code following the ENDIF statement if there is no ELSE. All arguments to a conditional must be known on pass 1. Otherwise, errors occur.

All types of conditional blocks have the same format:

```
IF    argument
.
.
.
[ELSE
.
.
.]
ENDIF
```

Each IF statement must have a matching ENDIF statement to terminate the conditional just as each ENDIF statement must have a matching IF statement. If an IF statement does not match an ENDIF statement, the assembler generates an error. If an ENDIF statement does not have an accompanying IF statement, the assembler generates a "Not in conditional block" error.

The ELSE directive is optional. The ELSE block is an alternative block of code that the processor processes if the original condition tested is false. You can only use one ELSE block for one IF statement. The ELSE block always refers to the most recent IF statement that is not followed by an ENDIF.

You can nest conditionals up to 255 levels. The following figure illustrates the nesting of conditionals:

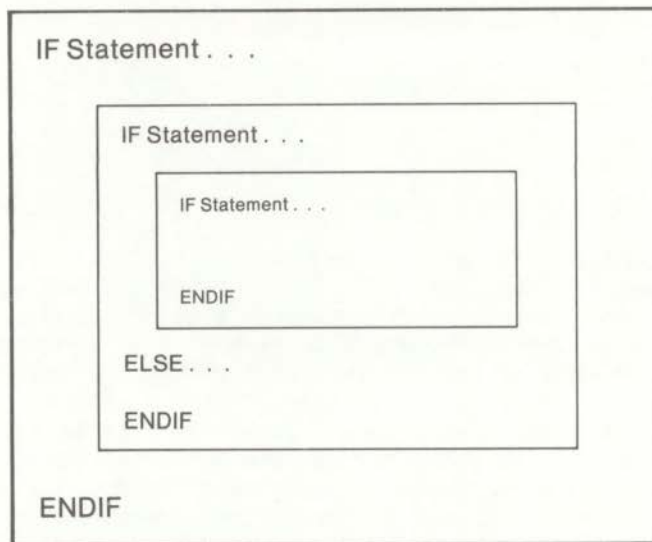


Figure 8-1. The Nesting of Conditionals

The expressions following the IF and IFE statements must evaluate to either true or false upon the first pass of the assembler. Therefore, the values in the expression following IF or IFE must be predefined and absolute.

The assembler can define the conditional expressions that are found within a macro block upon either the first or the second pass. Therefore, IFDEF and IFNDEF may contain symbolic expressions.

8.1 THE TEN CONDITIONAL DIRECTIVES

In the Wang PC assembly language, there are ten conditional directives, each of which tests for slightly different conditions. They are as follows.

IF expression	If the expression is true, the assembler assembles the statements within the conditional block.
IFE expression	If the expression is false, the assembler assembles the statements within the conditional block.
IF1	If the assembler is in pass 1, it assembles the statements in the conditional block.

IF2	If the assembler is in pass 2, it assembles the statements in the conditional block.
IFDEF symbol	If the symbol is either defined or external, the assembler assembles the statements in the conditional block.
IFNDEF symbol	If the symbol is not defined and not external, the assembler assembles the statements in the conditional block.

All of the following conditional directives normally appear in macro blocks. The arguments must appear in angle brackets as shown. They are usually dummy arguments that are replaced by parameters passed by the macro call.

IFIDN <argument 1>, <argument 2>	If the string in <argument 1> is identical to the string in <argument 2>, the assembler assembles the conditional block. The macro call must pass two identical parameters to meet this condition.
IFDIF <argument 1>, <argument 2>	If the string in <argument 1> is different from the string in <argument 2>, the assembler assembles the conditional block. The macro call must pass two different parameters to meet this condition.
IFB <argument>	If there is no argument in the angle brackets < > or if argument is blank, the assembler assembles the conditional block. When you call the macro, a parameter replaces the argument. If the macro call does not specify a parameter, the expression is blank, and the assembler assembles the block.

IFNB <argument>

If there is an argument in the angle brackets and if the macro call specifies a parameter, the argument is neither blank nor null, and the assembler assembles the conditional block.

Each time the assembler encounters SUBTTL followed by a name, it replaces the old subtitle with the new one. If you wish to turn off the subtitle for part of the output, enter a SUBTTL with a null string in place of the title. The syntax of the subtitle directive is as follows.

SUBTTL text

For example, if you have named a portion of your program Special I/O Routine, enter the following.

SUBTTL SPECIAL I/O ROUTINE

SPECIAL I/O ROUTINE appears under the title of each page. If you wish to change the subtitle, enter a new one as follows.

SUBTTL MACRO ADD1

If you wish to turn off SUBTTL, enter the following.

SUBTTL

Through this directive, the subtitle line on the listing file is left blank.

9.2 % OUT DIRECTIVE

The %OUT directive causes the assembler to list text on the terminal during assembly. %OUT is useful for displaying progress through a long assembly or for displaying the value of conditional assembly switches.

%OUT displays text on both passes of the assembler. If you only want to see text on one pass, use the IF1 or IF2 directives with the %OUT. The use of the %OUT accompanied by the IF1 directive causes the assembler to display or print out the text on the first pass. The use of the %OUT with the IF2 directive causes the assembler to display or to print out on the second pass. The syntax of the %OUT Directive is as follows.

%OUT text

In the following example, the message "Assembly half done" appears on both passes.

%OUT Assembly half done

In the next example, the message "Pass 1 is started" appears only on pass one.

```
IF1
%OUT Pass 1 is started
ENDIF
```

In the final example, the message "Pass 2 is started" appears only on pass two.

```
IF2
%OUT Pass 2 is started
ENDIF
```

9.3 TEXT SUPPRESSION DIRECTIVES

Several directives are designed to suppress text. Through these directives, you can control the number of lines in your listing file, the listing of false conditional expressions, the listing of the source and object code produced by a macro, and the creation of cross-references in portions of your cross-reference file.

9.3.1 The .LIST and .XLIST Directives

The .LIST and .XLIST directives allow you to control the number of lines that go into the listing file. For example, if you only want to see the listing for four lines, you can suppress the listing for all of the other lines in the file, saving both processing time and space. The .LIST directive causes the assembler to list all lines with their code. It is the default condition.

The .XLIST suppresses all listing. If you have created a list file and the assembler encounters the .XLIST directive, it stops listing the source and object code. .XLIST overrides all other listing directives. The assembler will not list, even if it encounters another type of listing directive, until it encounters a .LIST directive. When the processor encounters a .LIST directive, it begins to create a list program again. The syntax of .LIST and .XLIST is as follows.

```
.LIST
.XLIST
```


In the following example, the assembler suspends listing when it encounters the .XLIST directive and resumes listing when it encounters the .LIST directive.

```

      .
      .
      .
.XLIST
      .
      .
      .
.LIST

```

9.3.2 The .SFCOND, .LFCOND, and .TFCOND Directives

The .SFCOND, .LFCOND, and .TFCOND directives allow you to control the assembly of false conditionals. If you need to save space in memory, the suppression of false conditionals provides a method.

The .SFCOND directive suppresses the parts of the listing file that contain conditional expressions that evaluate to be false. The /X switch, a directive that you use when you call upon the assembler, also suppresses false conditionals. The .SFCOND directive overrides the /X switch. For more information on switches, see Chapter 1, Assembling the Program.

The .LFCOND directive causes the listing of conditional expressions that evaluate as false. This directive overrides the /X switch.

The first time the assembler encounters a .TFCOND directive, it suppresses false conditionals; the second time, it restores the listing of the false conditionals; the third time, it suppresses them, and so on. You can also use the .TFCOND directive in conjunction with the /X switch. If you use the /X switch when you answer the listing file prompt, the assembler suppresses the listing of false conditionals. If the /X switch is active when the assembler encounters a .TFCOND directive, the assembler reverses the normal outcome of a single .TFCOND directive. That is, normally the first .TFCOND directive causes the suppression of false conditionals. When an /X switch is active, however, the assembler restores the listing. When it encounters the second .TFCOND and the /X switch is active, it suppresses false conditionals. The /X switch toggles whatever condition was in effect at that time. The following table illustrates the use of .TFCOND.

Table 9-1. The Use of .TFCOND with /X

Number	.TFCOND	Effect of /X	Effect of No /X
First	.TFCOND	Restored	Suppressed
Second	.TFCOND	Suppressed	Restored
Third	.TFCOND	Restored	Suppressed
Fourth	.TFCOND	Suppressed	Restored

9.3.3 The .XALL, .LALL, and .SALL Directives

The .XALL, .LALL, and .SALL directives allow you to control the listing of the source and object code produced by a macro. These directives can help you save space in memory.

.XALL causes the assembler to list only the macro source code and object code that generate code. Source lines that do not generate code, such as comment lines for example, are not listed when .XALL is in effect. .XALL is the default condition.

.LALL causes the assembler to list the complete macro text for all expansions, including lines that do not generate code. However, comments preceded by two semicolons (;;) are suppressed even when you use the .LALL directive. To list comments inside macro blocks, use the COMMENT directive and .LALL.

.SALL causes the assembler to suppress the listing of all text and object code produced by macros.

9.3.4 The .CREF and .XCREF Directives

Through the .CREF and .XCREF directives, you can control the listing of cross-references in your file. These directives allow you to save space in memory. If you use the .XCREF directive without arguments, it suppresses the creation of a cross-reference file. .XCREF remains in effect until the assembler encounters .CREF to restart the creation of a cross-reference file. .CREF causes the assembler to create a cross-reference file and is the default condition. When the default .CREF is in effect, it is overridden by .XCREF just as .XCREF is overridden by .CREF.

When you use .XCREF followed by variables, the assembler does not place the variables in the cross-reference file. Only the variables are affected by the cross-referencing. The syntax of the .XCREF directive used with variables is as follows.

.XCREF variable list

In the following example, ARG1, ARG2, ARG3, and ARG4 will not appear in the cross-reference file.

.XCREF ARG1, ARG2, ARG3, ARG4

CHAPTER 10

THE LISTING FILE

10.1 THE LISTING FILE

The assembler can produce three separate files: the object file, the binary file that is sent on to the linker for further processing; the cross-reference file, which is the source file for CREF, the cross-reference program; and the listing file, which lists the code and offsets generated for each statement along with the errors in the program.

The listing file provides you with information that you can use to debug your program. It is composed of two parts. The first part of the listing file includes the following information.

- The source file
- The line number for each line of the source file
- The offset of each element in the source file
- The object code in either hexadecimal or octal notation (hexadecimal is the default; octal notation appears if you use the /O switch when you invoke the assembler)
- Whether a cross-reference file is being created
- The object and source code from a macro
- The object and source code from an INCLUDE statement

The second part of the listing file includes the following information:

- The name and length in bytes of all macros
- The name, width, and fields of all structures
- The name, size, align type, combine type, and class of all segments

- The name, type, value, and attributes of all symbols
- The number of warnings in the program
- The number of errors in the program

10.2 THE FIRST SECTION OF THE LISTING FILE

The following is an example of the first section of a listing file.

<u>Line</u>	<u>Offset</u>	<u>Code</u>	<u>Source Program</u>
1			
2			NAME PROG1
3			ASSUME DS:ABC, CS:ACD
4	0000		ABC SEGMENT
5	0000	0005	VAR1 DW 5
6	0002	0003	VAR2 DW 3
7	0004		ABC ENDS
8			
9	0000		ACD SEGMENT
10	0000	B8 ---- R	START: MOV AX, ABC
11	0003	8E D8	MOV, DS, AX
12	0005	A1 0000 R	MOV AX, VAR1
13	0008	8B 1E 0002 R	MOV BX, VAR2
14	000C	03 C3	ADD, AX, BX
15	000E		ACD ENDS
16			END START

In PROG1, offset addressing begins with 0000 twice: at the beginning of the ABC Segment and the beginning of the ACD Segment. The assembler counts the size of each element in bytes from the beginning of each segment. You can find the size of each element by subtracting its offset from the offset of the next line. MOV AX, ABC is three bytes long (0003-0000), for example, while MOV AX, VAR1 (0005-0003) is two bytes long.

The line of code is the hexadecimal translation of the binary instructions that the processor requires to run the program. These instructions provide the programmer with the following information.

- the op code for the instruction
- how long the operand is
- whether the operand is in a register or in memory
- the addressing mode used in this instruction
- in which registers the elements that add up to the offset address are located
- where the displacement, the 16 bit-value specified in the instruction, can be found
- which register is used in register-mode addressing
- whether there is a segment override in the instruction
- where the result should be stored in a double operand operation
- whether the instruction contains all the bits or whether it needs to be extended to form a 16 bit operand
- the data required for the instruction

Each digit of a hexadecimal number translates into a nibble; two digits together form a byte, while four digits form a word. Thus, the code on line 5, 0005, gives both the value and the size of VAR1. The four hexadecimal digits translate to a word in binary notation.

The processor must be able to locate information in order to process it. The offsets of VAR1 and VAR2 are stated in their codes. The offset of VAR1 is 0000, while the offset of VAR2 is 0002. In the instruction MOV AX, VAR1, the code is A1 0000 R while in the instruction MOV BX, VAR2, the code is 8B 1E 0002 R, naming the locations of VAR1 and VAR2 within the codes themselves.

A number of special symbols for the Linker occur throughout the code. They are as follows.

R	Indicates relocatable code. This code can be placed in memory at a location chosen by the Linker. It does not have to be placed at any fixed point.
E	Indicates an external symbol that the Linker must resolve.
----	Refers to a segment name, a group name, or a segment variable. It is used in moving the address of a segment or group to a register or in a JMP operation. The linker resolves the real address of the symbol.

There are no external symbols in PROG1. However, line 10 contains both the symbol for a segment name (----) and the R symbol indicating that the code is relocatable. Lines 12 and 13 also contain the R symbol, indicating that the two variables are relocatable and that the Linker must resolve their final addresses.

A number of other symbols can occur in a listing program. The following listing file of an incomplete program contains a number of symbols that did not occur in PROG1:

```

1      0000          STACK SEGMENT WORD STACK 'STACK'
2      = 0000      HEAPBEG      EQU      THIS BYTE
3      0000      14 [      DB      20 DUP(?)
4              ??
5      ]
6
7      = 0014      SKTOP      EQU      THIS BYTE
8      0014      STACK      ENDS
9
10     0000      MAINSTARTUP SEGMENT      'MEMORY'
11     DGROUP GROUP DATA, STACK,HEAP, CONST, MEMORY
12     ASSUME CS:MAINSTARTUP,
        DS:GROUP,
        ES:DGROUP, SS:DGROUP
13
14
15     PUBLIC      BEGXQQ
16
17     0000      BEGXQQ PROC      FAR
18     0000      B8 ---- R      MOV AX, DGROUP
19     0003      8E D8      MOV DS, AX
20     0005      8C 06 0022 R      MOV CESXQQ, ES
        .
        .
        .
25     000C      26: 8B 1E 0002      MOV BX, ES:2
26     0011      2B D8      SUB BX, AX
27     0013      81 FB 1000      CMP BX, 4096
28     0017      7E 03      JLE SMLSTK
29     0019      BB 1000      MOV BX, 4096
30     001C      SMLSTK:      REPT 4
31                        SHL BX, 1
32                        ENDM
33     001C      D1 E3      +      SHL BX,1
34     001E      D1 E3      +      SHL BX,1
35     0020      D1 E3      +      SHL BX,1
36     0022      D1 E3      +      SHL BX,1
37     0024      8B E3      MOV SP, BX
38
39
40
41     0069      EA 0000 ---- R      JMP FAR PTR BEG

```



```

42 006E                                BEGXQQ ENDP
43
44
45
46 007E                                MAINSTARTUP ENDS
47
48 0000                                ENTXCM SEGMENT WORD 'CODE'
49                                ASSUME CS: ENTXCM
50                                PUBLIC ENDXQQ, DOSXQQ
51
52 0000                                BEG PROC FAR
53 0000      9A 0000 ---- E            CALL ENTGQQ
54
55 0005                                ENDXQQ LABEL FAR
56 0005      9A 0000 ---- E            CALL ENDOQQ
57 000A      9A 0000 ---- E            CALL ENDYQQ
58 000F      9A 0000 ---- E            CALL ENDUQQ
59 0014      C7 06 0020 R 0000        MOV DOSOFF, 0
60 00      2E 0020 R                  JMP DWORD PTR DOSOFF
61 001E                                BEG ENDP
62
63
64
65 0037                                ENTXCM ENDS
66                                END BEGXQQ

```

There are a number of symbols in the above text that did not appear in PROG1. They are as follows.

Line(s)	Symbol	Meaning
2, 7	=	Indicates the presence of an = or EQU directive that equates two symbols.
3-5	[nn]	Indicates the presence of a value as given in the DUP directive.
25	nn:	Indicates the presence of a segment override.
33-36	+	Indicates that the code comes from a macro expansion.
56-58	E	Indicates that the symbol is external.

The assembler uses two more symbols that do not appear in either program. They are the following.

C	indicates that the line comes from an INCLUDE statement
nn/	indicates that the REP or LOCK prefix instruction is present

10.3 THE SYMBOL TABLE

The assembler composes the symbol table so that the assembler has easy access to a wide variety of information, such as where to find symbols; the size, align, combine and class type of segments and groups; the type value and attribute of symbols; the size, the number of fields in, and the offset of the fields of a structure; the size of, the number of fields in, the bit shift of the fields in a structure, the size of the mask of, and the initial value of the fields in a record; and the length of macros.

The symbol table is divided into four separate areas.

- macros
- structure and records
- segments and groups
- symbols

If your program contains no elements in any one category, that category does not appear in the symbol table.

In PROG1, the symbol table contains only two parts: 1) segments and groups and 2) symbols since the program itself contains only those symbols. The symbol table for PROG1 is as follows.

Segments and groups:

N a m e	Size	align	combine class
ABC.	0004	PARA	NONE
ACD.	000E	PARA	NONE

Symbols:

N a m e	Type	Value	Attr
START.	L NEAR	0000	ACD
VAR1	L BYTE	0000	ABC
VAR2	L BYTE	0002	ABC

Warning Severe
Errors Errors
0 0

10.3.1 The Listing of Macros

If your program contains macros, they are listed first in the symbol table. They appear as follows.

MACROS:

Name	Length
BIOSCALL	0002
DISPLAY	0005
DOSCALL	0002
KEYBOARD	0003

The names of all the macros that appear in your program are listed under Name. The size of the macros is listed under Length. The number refers to a 32-byte block unit. The macro BIOSCALL is 64 bytes long (2 x 32), while DISPLAY is 160 bytes long (5 x 32).

10.3.2 The Listing of Structures and Records

While structures are composed of various fields of bytes, records are composed of various fields of bits. Two subdivisions of the structures and records table apply exclusively to records: mask and initial. An example of a structure listing in the symbol table is the following.

Name	Width Shift	# fields Width Mask Initial
PARMLIST	001C	0004
BUFSIZE.	0000	
NAMESIZE	0001	
NAMETEXT	0002	
TERMINATOR	001B	

In the previous example, the unindented name, PARMLIST, is the name of the structure; the four indented names are the names of the fields in PARMLIST. The number that follows PARMLIST under the Width Shift category is the number of bytes in the whole structure. In this case, the structure has 001CH (28 bytes). Under the category # Fields/Width, the number following PARMLIST is the number of fields in the structure. PARMLIST, as the 0004 indicates, has four fields.

The numbers under the category of Width/Shift that follow the field names of PARMLIST indicate the offset of the field into the structure. Thus, 0000 indicates that BUFSIZE is the first field in the record PARMLIST. BUFSIZE is one byte long (subtract the offset of NAMESIZE from the offset of BUFSIZE to determine BUFSIZE's size), and begins with byte #0. The following is an example of a record listing of a symbol table.

Name	Width Shift	# fields Width	Mask	Initial
BAZ	0008	0003		
FLD1.	0006	0002	00CO	0040
FLD2.	0003	0003	0038	0000
FLD3.	0000	0003	0007	0003
BAZ1.	000B	0002		
BZ1	0003	0008	07F8	0400
BZ2	0000	0003	0007	0002

The names of the records are under BAZ and BAZ1. Following the record names are the sizes of the records in bits (eight in each case) and the number of fields in the record. There are three fields in BAZ and two fields in BAZ1.

Following the field names is the shift count. The shift count is the number of bits that the lowest bit in the field must move to the right to be positioned in the lowest bit of the record. For example, FLD1 has a shift count of six; that is, the lowest bit of FLD1 must move six places to the right to be in the 0 position. FLD2 has a shift count of three because the lowest bit in FLD2 has to move three bits to the right to be positioned in the lowest bit of the record. The lowest bit in FLD3 already occupies the lowest bit position in the record.

The next group of numbers under #Fields/Width tells how many bits are in the field. For example, FLD1 has 2 bits, while FLD2 has three bits, and FLD3 has three bits.

Next to # Fields/Width is Mask. The mask value tells the maximum amount the field is worth if all the bits are set (if they are all 1's instead of 0's). For example, BAZ is composed of three fields. The first field, FLD1, occupies the highest two bits whose combined worth is 192 ($2^{**7} + 2^{**6}$) if both bits are set. FLD2 is composed of three bits, occupying the positions 3, 4, and 5; their total worth is 56 ($2^{**5} + 2^{**4} + 2^{**3}$) if all the bits are set. FLD3 occupies the three lowest bits in positions 0, 1, and 2 whose worth is 7 ($2^{**2} + 2^{**1} + 2^{**0}$) if all the bits are set. This total worth is called the mask. The following diagram illustrates the shift count and mask of FLD1, FLD2, and FLD3 of BAZ.

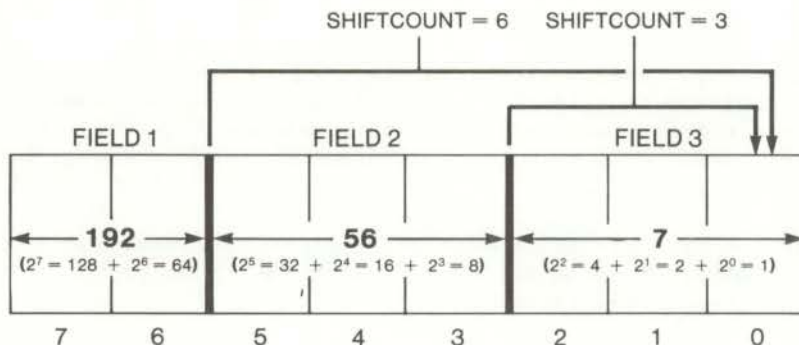


Figure 10-1. The Shift Count and Mask of a Record

The last column is Initial. Initial specifies the initial value that you gave to the field when you named it, if any. For example, FLD1 is initially worth 40 hexadecimal, while FLD2 is 0.

10.3.3 The Listing of Segments and Groups

The third category listed in the symbol table is segments and groups. Groups include multiple segments combined so that the assembler addresses all of the separate segments from one base address. Segments that are not part of a group have individual base addresses. The following list includes both segments and groups.

Name	Size	align	combine	class
AAAXQQ	0000	WORD	NONE	'CODE'
DGROUP	GROUP			
DATA	0024	WORD	PUBLIC	'DATA'
STACK.	0014	WORD	STACK	'STACK'
CONST.	0000	WORD	PUBLIC	'CONST'
HEAP	0000	WORD	PUBLIC	'MEMORY'
MEMORY	0000	WORD	PUBLIC	'MEMORY'
ENTXCM	0037	WORD	NONE	'CODE'
MAINSTARTUP.	007E	PARA	NONE	'MEMORY'

In the previous example, GROUP appears in the size column next to a group name. Segments that are part of a group are indented under the group name. Segments that are not part of a group are not indented. There are three segments: AAAXQQ, ENTXCM, and MAINSTARTUP. There is also one group, DGROUP, composed of five segments: DATA, STACK, CONST, HEAP, and MEMORY.

Next to the name of the segment or group is the Size column that indicates the number of bytes the segment occupies. For example, AAAXQQ does not occupy any bytes while ENTXCM occupies 37H bytes.

The column to the right of Size is align. This column indicates the type of boundary on which a segment begins. There are four types of boundaries, defined as follows.

PAGE	The address begins on a 256 byte boundary, the last two digits of the hexadecimal number being 00: XXX00.
PARA	The address begins on a boundary evenly divisible by 16, the last digit of the hexadecimal number being 0: XXXX0.
WORD	The address begins on an even numbered boundary, the last bit of the low byte being 0.
BYTE	The address can begin anywhere.

In the previous example, most of the segments begin on WORD boundaries. MAINSTARTUP, however, begins on a PARA boundary. If you want to know more about these attributes, refer to Section 6.2, Combining Segments.

There are five possible combine types you can declare in the source program: none, public, stack, common, and memory. They are defined as follows.

NONE	The processor loads the segments separately and they remain separate entities (each private segment has its own base address).
PUBLIC	The processor loads public segments of the same name and class next to each other, and they use one base address. The offset is from the first to the last segment loaded.
STACK	The processor loads stack segments of the same name and class next to each other, and they use one base address. The offset is from the first to the last segment loaded. The stack pointer is set to the first address of the first stack segment
MEMORY	The processor loads memory segments of the same and class next to each other, and they use one base address. The offset is from the first to the last segment loaded.
COMMON	The processor loads common segments of the same name and class so that they overlap one another. There is one base address for all common segments of the same name and the length of the common area is the length of the longest segment.

In the previous example, AAAXQQ, ENTXCM, and MAINSTARTUP are loaded separately, while DATA, STACK, CONST, HEAP, and MEMORY are loaded next to other segments of the same name and class.

The Linker loads segments by declared class name. It loads all the segments that belong to the first class it encounters together with the other segments of that class, and then all the segments that belong to the next class it encounters, and so on, until it loads all the segments. In the previous example, the Linker loads the two 'CODE' segments first; the 'STACK' segment second; the 'CONST' segment third; and the three 'MEMORY' segments last.

10.3.4 The Listing of Symbols

The last category listed in the symbol table is symbols. The table is divided into two parts: those symbols formed through the EQU or = directives and those that are not. The symbols formed through the EQU or = directive appear in the following example.

Symbols:

Name	Type	Value	Attr
VAR1	Number	0005	
VAR2	Text	1.234	
VAR3	Number	0008	
VAR4	Alias	VAR1	
VAR5	Text	5[PB] [DI]	
VAR6	Opcode		

All of the above symbols were defined through the use of either the EQU or the = directive. The entries under type, number, text, alias, and opcode express the value of the symbols. These expressions are defined as follows.

Number	A constant numeric value
Opcode	An instruction mnemonic, such as: VAR1 EQU ADD
Alias	The substitution of one symbolic name for another, such as: VAR1 EQU VAR2
Text	The "text" the symbol represents (any other operand to an EQU directive that does not fit into one of the other three categories), such as: VAR3 EQU 'WOW' VAR4 EQU DS:8[BX] VAR5 EQU 1.234

In the previous listing, VAR1 and VAR3 fall under the Number category and are worth 5 and 8, respectively. VAR2 and VAR5 with the values 1.234 and 5[BP] [DI], respectively, fall under the category of Text. VAR4 is an alias for another variable, VAR1, while VAR6 is in the opcode category.

The second part of the symbol table, presenting the symbols not defined by the EQU or = directives, follows.

Symbols:

Name	Type	Value	Attr
BEGHQQ . . .	L WORD	0012	DATA Global
BEGQQQ . . .	L FAR	0000	External
BEGXQQ . . .	F PROC	0000	MAIN Global Length=006E
CESXQQ . . .	L WORD	0022	DATA Global
CLNEQQ . . .	L WORD	0001E	DATA Global
DOSXQQ . . .	F PROC	001E	ENTXCM Global Length = 0019
ENDOQQ . . .	L FAR	0000	External
HEAPBEG . . .	BYTE	0000	STACK
HEAPLOW . . .	BYTE	0000	HEAP
RESEQQ . . .	L WORD	0010	DATA Global
SMLSTK . . .	L NEAR	001C	MAIN

Under the type category, six symbols can occur. They are the following.

L	A label
F	Far, accessed by segment and offset
N	Near, accessed by offset from same segment
PROC	A procedure
WORD	A word in length
BYTE	A byte in length

The type for BEGLHQQ, the first entry, is a label one word long. SMLSTK is a label that can be accessed by a NEAR JMP or CALL; and BEGOQQ is a label that must be accessed via a FAR JMP or CALL.

In the value category are the addresses that the symbol represents. Thus, BEGLHQQ is located at 12H, while ENDOQQ is at 0.

The ATTR column shows the segment of the symbol. If the segment is not known, the ATTR column is blank. In the above symbol table, BEGLHQQ belongs to the DATA segment and HEAPBEG belongs to the STACK segment, but the segment to which HEAPLOW belongs is not known.

Next to the segment name, the words Global or External may appear. The assembler distinguishes between the global and the external names. If the name is neither global nor external, this column is blank. BEHLHQQ is global while BEGOQQ is external, for example. HEAPBEG is neither global nor external.

The last entry in the symbol table belongs only to procedures. It takes the following syntax.

Length = nnn

The length tells the size of procedures in bytes. There are two such entries above. The length of BEGXQQ is 006EH, while the length of DOSYQQ is 19H.

CHAPTER 11

ERROR MESSAGES

The error messages help you debug your program. There are three types of error messages: those produced by the assembler, I/O handler errors, and runtime errors. The assembler picks up errors in referencing and definition during assembly. For example, an undefined variable, a variable defined as two different types, and the illegal use of registers are errors for which the assembler sends out messages.

The faulty use of input and output devices cause I/O handler errors. For example, trying to access a file that is nonexistent or already in use, or trying to print your file on a nonexistent device causes an I/O handler message. Runtime errors occur as your program is being executed. There are only two runtime errors: Out of Memory and Internal Error.

11.1 ASSEMBLER ERROR MESSAGES

When the assembler encounters an error, it sends out an error message to help you debug your program. The message occurs within the listing file. In addition, the assembler provides a summary of all errors at the end of the listing file.

1				NAME PROG 1
	E r r o r	---	1:Extra characters	on line
2				ASSUME DS:ABC, CS: ACD
3		0000		ABC SEGMENT
4		0000	05	VAR1 DB 5
5		0001	03	VAR2 DB 3
6		0002		ABC ENDS

```

7
8          0000          ACD SEGMENT
9          0000      B8 ---- R          START:      MOV AX, ABC
10         0003      8E D8          MOV DS, AX
11         0005      A1 0000 R          MOV AX, VAR1
   Error    ---      31:Operand types must match
12         0008      8B 1E 0001 R          MOV BX, VAR2
   Error    ---      31:Operand types must match
13         000C      03 C3          ADD AX, BX
14         000E          ACD ENDS
15          END START

```

In the program above, three errors occur, but two errors are of the same type, type 31. The name of the program, PROG 1, has an extra space in it. Therefore, the message "Extra characters on line" indicates that the space is superfluous. By deleting the space and writing PROGL, you eliminate the error.

The second error is caused because the operands VAR1 and VAR2 are defined as bytes in the ABC SEGMENT and are moved into word registers. You can correct this mistake by making the variable size consistent with the register size.

```

MOV AL, VAR1
MOV BL, VAR2
OR
VAR1 DW
VAR2 DW
OR
MOV AX, WORD PTR VAR1
MOV BX, WORD PTR VAR1

```

The following table gives the code number of the error message, the message in alphabetical order, and the cause of the error.

<u>Code</u>	<u>Message</u>	<u>Error</u>
23	Already defined locally	Attempt to define previously defined local symbol as EXTERNAL.
7	Already had ELSE clause	ELSE clause defined within an existing ELSE clause when nesting is not legal.
46	Already have base register	Address given double base.

<u>Code</u>	<u>Message</u>	<u>Error</u>
47	Already have index register	Address given double index.
0	Block nesting error	Nested procedures, segments, structures, macros, IRC, IRP, or REPT are not properly terminated; they need an end statement.
58	Byte register is illegal	Illegal use of one of byte registers.
67	Can't override ES segment	Illegal attempt to override ES segment.
68	Can't reach with segment reg	Variable out of range due to lack of ASSUME statement.
70	Can't use EVEN on BYTE segment	Illegal use of EVEN directive with segment declared to be byte boundary.
83	Circular chain of EQU aliases	An alias EQU eventually points to itself.
42	Constant was expected	Assembler expects constant and finds something else.
59	CS register illegal usage	Attempt to use CS register illegally.
78	Directive illegal in STRUC	Illegal statement within STRUC block when only Define directives or comments preceded by semicolon are permitted.
29	Division by 0 or overflow	Expression is given that results in a divide by 0.
74	DUP is too large for linker	Nesting of DUP's is such that too large a record is created for the Linker.

<u>Code</u>	<u>Message</u>	<u>Error</u>
1	Extra characters on line	Characters other than those needed to define the instruction are received.
80	Field cannot be overridden	Attempt to give a value to a field that cannot be overridden in a STRUC initialization statement.
71	Forward needs override	Message is not currently used.
17	Forward reference is illegal	Attempt to forward reference something that must be defined in pass 1.
55	Illegal register value	Register value specified does not fit into 3 bit "reg" field; it is greater than 7.
57	Illegal size for item	Illegal size of referenced item.
32	Illegal use of external	Attempt to use external directive in an illegal manner.
49	Illegal use of register	Attempt to use register with an instruction where no such instruction is possible.
72	Illegal value for DUP count	Attempt to use DUP when it is not a positive, nonzero constant.
52	Improper operand type	Attempt to use operand illegally results in no generation of op code.
61	Improper use of segment reg	Illegal specification of segment register.
54	Index displ. must be constant	Attempt to use variable as displacement value.
65	Label can't have segment override	Attempt to use segment override illegally.
38	Left operand must have segment	Attempt to use something in right operand that requires a segment in the left operand which is absent.
76	More values than defined	Allocation of too many fields in REC or STRUC.
45	Must be associated with code	Attempt to use data related item where code item is expected.

<u>Code</u>	<u>Message</u>	<u>Error</u>
44	Must be associated with data	Attempt to use code related item where data related item is expected.
60	Must be AX or AL	Attempt to use register other than AX or AL when only these are legal.
48	Must be index or base register	Attempt to put other than index or base register in square brackets [] when only index or base are legal.
13	Must be declared in pass 1	Assembler expecting constant value, but got another value.
69	Must be in segment block	Attempt to generate code outside of a segment.
33	Must be record field name	Assembler expecting record field name, but another value is specified.
34	Must be record or field name	Assembler expecting a record or field name, but another value is specified.
18	Must be register	Assembler expecting register as an operand, but another value is specified.
20	Must be segment or group	Assembler expecting segment or group, but another value is specified.
37	Must be structure field name	Assembler expecting structure field name, but another value is specified.
22	Must be symbol type	Assembler expecting WORD, DW, QW, BYTE or TB but another value is specified.
36	Must be var, label or constant	Assembler expecting a variable, label, or constant, but another value is specified.
66	Must have opcode after prefix	Attempt to use one of prefix instructions without specifying any op code after it.
64	Near JMP/CALL to different CS	Attempt to do NEAR jump or call to location in different CS ASSUME.
56	No immediate mode	Attempt to use immediate mode with op code that cannot accept immediate mode.

<u>Code</u>	<u>Message</u>	<u>Error</u>
62	No or unreachable CS	Attempt to jump to label that is out of reach.
41	Normal type operand expected	Assembler expecting a variable label, but either STRUCT, FIELDS, NAMES, BYTE, WORD, or DW is specified.
8	Not in conditional block	Specification of an ENDIF or ELSE without an active conditional assembly directive.
25	Not proper align/combine type	SEGMENT parameters are incorrect.
39	One operand must be const	Illegal use of addition operator.
77	Only initialize list legal	Attempt to use STRUC name without angle brackets.
63	Operand combination illegal	Specification of two-operand instruction where combination specified is illegal.
40	Operands must be same or 1 abs	Illegal use of subtraction operator.
43	Operand must have segment	Illegal use of SEG directive.
35	Operand must have size	Assembler expecting operand to have a size, but it is absent.
51	Operand not in IP segment	Assembler cannot access operand because it is not in the current CS segment.
31	Operand types must match	Assembler needs matching arguments, but different kinds or sizes are specified.
27	Operand was expected	Assembler expecting operand, but it is missing.
28	Operator was expected	Assembler expecting operator, but it is missing.

<u>Code</u>	<u>Message</u>	<u>Error</u>
81	Override is of wrong type	Assembler expecting certain type of override in STRUC initialization statement, but an other type is specified.
79	Override with DUP is illegal	Illegal use of DUP in override in a STRUC initialization statement.
6	Phase error between passes	An ambiguous instruction directive causes location of label to change value between pass 1 and pass 2 of assembler.
4	Redefinition of symbol	The symbol redefined so that its values on pass one and pass two are inconsistent.
26	Reference to mult defined	Instruction references something that has been defined more than once.
2	Register already defined	Assembler has internal logic errors.
82	Register can't be forward ref	You cannot use a forward reference with a register name.
53	Relative jump out of range	The jump is not within 127 bytes of current instruction.
24	Segment parameters are changed	Arguments to SEGMENT not identical with those stated the first time this segment is used.
30	Shift count is negative	Shift expression generates negative shift count.
12	Should have been group name	Assembler expecting group name, but another value is specified.
15	Symbol already different kind	Attempt to define symbol differently from previous definition.
73	Symbol already external	Attempt to define symbol as local that already is defined as external.
21	Symbol has no segment	Attempt to use variable with SEG with no defined segment.
5	Symbol is multi-defined	Symbol was already defined earlier in same program.

<u>Code</u>	<u>Message</u>	<u>Error</u>
16	Symbol is reserved word	Illegal attempt to use assembler reserved word.
9	Symbol not defined	Attempt to use symbol with no definition.
14	Symbol type usage illegal	Illegal use of PUBLIC symbol.
10	Syntax error	Syntax of statement does not match any recognizable syntax.
11	Type illegal in context	Attempt to use illegal type.
3	Unknown symbol type	Attempt to use symbol that has something unrecognizable in type field.
75	Usage of ? (indeterminate) bad	Improper use of "?".
50	Value is out of range	Value is too large for expected use.
19	Wrong type of register	Assembler expects one type of directive or instruction, but another one is specified.

At the end of the symbol table in the listing file, a summary of all the errors in the program appears. The error summary for PROG1 is as follows.

Warning	Severe
Errors	Errors
1	2

11.2 I/O HANDLER ERRORS

The second type of error message that appears in the list table is the I/O handler message. The I/O handlers, also known as device drivers, are small programs that communicate with the input and output devices. The error messages generated by the I/O handlers appear in a different format from those generated by the assembler. The format is as follows.

```
MASM Error -- error message
in: filename
```

The second line of the message in filename states the name of the file that is being handled when the error occurs. The following is a list of the error messages.

<u>Code</u>	<u>Message</u>
114	Data Format
108	Device full
102	Device name
105	Device offline
112	File in use
107	File name
110	File not found
113	File not open
104	File system
101	Hard data
115	Line too long
106	Lost file
103	Operation
111	Protected file
109	Unknown device

11.3 RUNTIME ERRORS

The processor puts out runtime errors when the program is being executed. There are only two runtime errors, and neither one uses a numerical code.

An Internal error is usually caused by an arithmetic check. When this code occurs, call the Wang Professional Computer Assistance Center at 1-800-343-1098. The Out of Memory error is caused when the source is too big or too many labels are in the symbol table.



CHAPTER 12

THE CROSS-REFERENCE FILE

The assembler creates the cross-reference file, the file that the Wang PC CREF facility uses to create the cross-reference listing file. The assembler's product, the cross-reference file, is input into CREF. CREF creates the final product, a file that lists in alphabetical order the line numbers of all of the symbols used in your program as well as the line numbers on which the symbol is defined.

CREF is a valuable tool in debugging your program. It can help you locate quickly all occurrences of any symbol by line number. In a long, multi-modular program, CREF provides you with a quick and easy way of checking symbols.

12.1 CREATING A CROSS-REFERENCE FILE

To create a cross-reference file, command the assembler to create a cross-reference file during assembly. You do this either by writing the name of your program into the fourth position in the series of file commands or by writing the name of your file after the fourth prompt. Both methods are demonstrated below.

```
MASM PROG1, PROG1, PROG1, PROG1
```

```
Cross reference [NUL.CRF]:
```

In the first example, the last PROG1 in the listing after MASM creates the cross-reference file, PROG1.CRF. In the second example, you create a cross reference file by entering PROG1 after the prompt as follows.

```
Cross reference [NUL.CRF]: PROG1
```

The default of the cross-reference prompt is a null file. Therefore, you must answer the prompt if you wish to create a cross-reference file. .CRF is the default extension of the cross-reference file. It is legal to use another extension if you wish, however.

12.2 INVOKING CREF

Once the assembler has created a cross-reference file, you must convert it to the cross-reference listing file with the default extension .REF. The CREF program accomplishes this.

To invoke CREF, you can either list the names of the cross-reference file and the cross-reference listing file next to the command CREF or you can respond to individual prompts.

12.2.1 Invoking CREF: Method One

When you use method 1 to invoke the assembler, you list the cross-reference file and the cross-reference listing file next to the command CREF. The syntax of this method is as follows.

CREF cross-reference file, cross-reference listing

For example, the following command causes CREF to be loaded into memory, where it creates a cross-reference listing file.

CREF PROG1, PROG1

To select the default file name and the default extension for the cross-reference listing file, you can enter a semicolon after the cross-reference file name. Through the following command, CREF produces a cross-reference listing file, called PROG1.REF, from the cross-reference file called PROG1.CRF.

CREF PROG1;

To give the listing file a different name, extension, or destination, simply specify these differences when entering the command line. The following command causes CREF to produce a cross-reference listing file named WORK.ARG, from the cross-reference file, named PROG1, and to place it on the diskette in drive B.

CREF PROG1, B:WORK.ARG

12.2.2 Invoking CREF: Method Two

When you use Method Two to invoke the assembler, you respond to individual prompts. Begin by entering the command CREF from the DOS COMMAND PROCESSOR, choose OTHER from the System Menu, or pick CREF from the the Program Development Menu choice CREF.

CREF

CREF is loaded into memory and returns with the following two prompts one at a time.

Cross reference [.CRF]:

Listing [crffile.REF]:

To the first prompt, enter the name of the cross-reference file produced by the assembler. If your file name has a different extension from the default, .CRF, enter it. Otherwise, the name of your file is sufficient.

Cross reference [.CRF]: PROG1

To the second prompt, enter the name that you want the cross-reference listing file to have. If you want to use the default name, the name of your cross-reference file with the default .REF, press RETURN. In the first example below, you give the cross-reference listing file the name PROG2.ART; in the second example, the processor gives it the default name, PROG1.REF.

Cross reference [.CRF]: PROG2.ART

Cross reference [.CRF]: RETURN

If you want the listing file placed on a drive or device other than the default drive, specify the drive or device when entering your response to the prompt as follows.

Cross reference [.CRF]: B:PROG2.ART

12.3 COMMAND CHARACTERS

There are two command characters in CREF: the semicolon and the Control-C. Their functions can be summarized as follows.

The semicolon (;)	To select the default for the cross-reference listing file prompt, use the semicolon.
CONTROL-C	To exit from CREF and return to the system prompt, enter Control-C.
SHIFT-CANCEL	To exit from CREF and return to the system prompt, enter SHIFT and CANCEL.
2ND -COMMAND-CANCEL	To exit from CREF and reboot the system, enter 2ND and COMMAND and CANCEL.

When you enter a semicolon rather than a file name after the cross-reference file prompt, you choose the default. In the following example, CREF creates the file PROG1.REF.

Cross reference [.CRF]: PROG1
 Listing [crffile.REF]: ;

If you make a mistake or otherwise wish to return to control level, press the CONTROL key at the same time as the C key. The following system prompt appears.

CONTROL-C
 B:

12.4 THE FORMAT OF THE CROSS-REFERENCE LISTING FILE

The cross-reference listing file is an alphabetical list of all the symbols in your program. Next to the symbols is a list of all the line numbers in which the symbol occurs. A pound sign indicates on which line number the symbol is first defined. The following is an example of a cross-reference listing file.

Symbol Cross Reference	(# is definition)			Cref-1	
AAAXQQ	37#	38			
BEGHQQ	83	84#	154	176	
BEGOQQ	33	162			
BEHXQQ	113	126#	164	223	
CESXQQ	97	99#	129		
CLNEQQ	67	68#			
CODE	37	182			
CONST.	104	104	104	110	
CRCXQQ	93	94#	210	215	
CRDXQQ	95	96#	216		
CSXEQQ	65	66#	149		
CURHQQ	85	86#	155		
DATA	64#	64	155		
DGROUP	110#	110	111	111	127
DOSOFF	98#	198	199		
DOXYQQ	184	204#	219		

12.5 ERROR MESSAGES

When an error occurs, CREF aborts and control returns to the operating system. The following is the format of all error messages.

Fatal I/O Error error number
in File: filename

The filename is the name of your file.

The error number is the code number for the particular error.

The following is a list of all errors, their code numbers and their cause, or what to do if the particular error occurs.

<u>Code</u>	<u>Message</u>	<u>Cause or what to do</u>
101	Hard data error	Unrecoverable input or output due to disk damage.
101	Device name error	Illegal device specification.
103	Internal error	Report to the Wang Professional Computer Assistance Center at 1-800-343-1098.
104	Internal error	Report to Wang Professional Computer Assistance Center at 1-800-343-1098.
105	Device offline	Device not ready to accept data because device is inaccessible. For example, disk drive door could be open or no printer is attached.
106	Internal Error	Report to Wang Professional Computer Assistance Center at 1-800-343-1098.
108	Disk full	The disk is full. To run program, delete some files.

<u>Code</u>	<u>Message</u>	<u>Cause or what to do</u>
110	File not found	Either the name submitted is incorrect or the file is not on the disk.
111	Disk is write protected	Protection bit set so that file cannot be written.
112	Internal error	Report to Wang Professional Computer Assistance Center at 1-800-343-1098.
113	Internal error	Report to Wang Professional Computer Assistance Center at 1-800-343-1098.
114	Internal error	Report to Wang Professional Computer Assistance Center at 1-800-343-1098.
115	Internal error	Report to Wang Professional Computer Assistance Center at 1-800-343-1098.

12.6 THE FORMAT OF CREF

The cross-reference file is made up of consecutive bytes in memory which store the name of each symbol, the line numbers on which references to the symbol occur, and the line number on which a definition of the symbol occurs. In addition, the cross-reference file contains some formatting information: the title, the page length and the line length.

To create the cross-reference file, the assembler counts the lines of the program. It must note each reference to a symbol and each definition that occurs on an individual line.

To create the cross-reference file, the assembler uses a series of numbers, each of which gives CREF some needed information to create the cross-reference listing file. The numbers and the information that they relate to CREF are as follows.

<u>Number</u>	<u>Symbolic Value</u>	<u>Meaning of Message and Rules</u>
01	Reference symbol	On the following line, there is a reference to a symbol whose name follows in ASCII code. The name may be from 1 to 80 ASCII characters long. Additional characters are truncated.
02	Define symbol	On the following line, there is a definition of a symbol whose name follows in ASCII code. The name may be from 1 to 80 ASCII characters long. Additional characters are truncated.
04	End of line	Add one more line to the line count.
05	End of file	No more processing is necessary. All symbols have been referenced. This symbol is followed by 1AH CONTROL-Z) and indicates the end of the file.
06	Title defined	The preceding ASCII characters compose the title. The title may be from 1 to 80 characters long. Additional characters are truncated. The last title definition record encountered is used for the title; if no title definition record is encountered, the title line is left blank.
07	page length/line length	The first of the preceding two numbers represents page length; the second represents line length. These two values are the first two bytes of the file. The page length can be from 1 to 155 lines; the page width can be from 1 to 132 characters. The default page length is 58; the default page width is 80.

The following lines of numbers are part of a memory dump of the cross-reference file of PROG4, a program whose listing file appears in Section 10.2. In other words, if you could see into the memory of the computer, the first 9 lines of the cross-reference file of PROG4 would appear as a sequence of bytes containing the following numbers.

```
3A 50 07 04 04 02 48 45 41 50 42 45 47 04 04 04
04 04 02 53 4B 54 4F 50 04 01 53 54 41 43 4B 04
```

The first number, 3A, tells the number of lines to be printed per page. 3A in hexadecimal is equivalent to 58 in decimal notation. The page length for this document is therefore 58 lines.

The second number, 50, refers to the number of characters per line. 50 is the hexadecimal equivalent for 80 decimal. Therefore, the page width for the document is 80 characters.

The third number, 07, is a kind of delimiter that tells CREF that the two preceding bytes define the page size.

The fourth number, 04, is the symbol for the end of the line. CREF counts each 04 as a line counter. The first 04 is followed by a second one, indicating that CREF is to add two lines to the count.

The next number, 02, informs CREF that the following symbol is defined on line three, the line that CREF is presently on.

The numbers, 48, 45, 41, 50, 42, 45, and 47, are the ASCII equivalents of H, E, A, P, B, E, G. Together they spell the name of the symbol: HEAPBEG.

The name of the symbol is followed by five 04's, indicating that the assembler has found no symbolic names on the following five lines, and should add five to the counter. The 04's are followed by an 02, indicating that on the eighth line, the assembler finds another symbol and its definition. The ASCII equivalent of SKTOP follows in the numbers 53, 4B, 54, 4F, and 50. An 04 indicates the completion of that line.

The next number, 01, indicates that the following code refers to a symbol. The numbers 53, 54, 41, 43, and 4B are the ASCII equivalent of that symbol: STACK. The last number on the line, 04, indicates the end of that line. Using this code, CREF creates a file: PROG4.REF:. PROG4.REF lists each symbol, the line numbers on which it falls, and the line number on which it is defined.

APPENDIX A
ASCII CHARACTER CODES

<u>Dec</u>	<u>Hex</u>	<u>CHR</u>	<u>Dec</u>	<u>Hex</u>	<u>CHR</u>	<u>Dec</u>	<u>Hex</u>	<u>CHR</u>
000	00H	NUL	043	2BH	+	086	56H	V
001	01H	SOH	044	2CH	,	087	57H	W
002	02H	STX	045	2DH	-	088	58H	X
003	03H	ETX	046	2EH	.	089	59H	Y
004	04H	EOT	047	2FH	/	090	5AH	Z
005	05H	ENQ	048	30H	0	091	5BH	[
006	06H	ACK	049	31H	1	092	5CH	\
007	07H	BEL	050	32H	2	093	5DH	]
008	08H	BS	051	33H	3	094	5EH	^
009	09H	HT	052	34H	4	095	5FH	_
010	0AH	LF	053	35H	5	096	60H	`
011	0BH	VT	054	36H	6	097	61H	a
012	0CH	FF	055	37H	7	098	62H	b
013	0DH	CR	056	38H	8	099	63H	c
014	0EH	SO	057	39H	9	100	64H	d
015	0FH	SI	058	3AH	:	101	65H	e
016	10H	DLE	059	3BH	;	102	66H	f
017	11H	DC1	060	3CH	<	103	67H	g
018	12H	DC2	061	3DH	=	104	68H	h
019	13H	DC3	062	3EH	>	105	69H	i
020	14H	DC4	063	3FH	?	106	6AH	j
021	15H	NAK	064	40H	@	107	6BH	k
022	16H	SYN	065	41H	A	108	6CH	l
023	17H	ETB	066	42H	B	109	6DH	m
024	18H	CAN	067	43H	C	110	6EH	n
025	19H	EM	068	44H	D	111	6FH	o
026	1AH	SUB	069	45H	E	112	70H	p
027	1BH	ESCAPE	070	46H	F	113	71H	q
028	1CH	FS	071	47H	G	114	72H	r
029	1DH	GS	072	48H	H	115	73H	s
030	1EH	RS	073	49H	I	116	74H	t
031	1FH	US	074	4AH	J	117	75H	u
032	20H	SPACE	075	4BH	K	118	76H	v
033	21H	!	076	4CH	L	119	77H	w
034	22H	"	077	4DH	M	120	78H	x
035	23H	#	078	4EH	N	121	79H	y
036	24H	\$	079	4FH	O	122	7AH	z
037	25H	%	080	50H	P	123	7BH	{

<u>Dec</u>	<u>Hex</u>	<u>CHR</u>	<u>Dec</u>	<u>Hex</u>	<u>CHR</u>	<u>Dec</u>	<u>Hex</u>	<u>CHR</u>
038	26H	&	081	51H	Q	124	7CH	
039	27H	'	082	52H	R	125	7DH	}
040	28H	(	083	53H	S	126	7EH	~
041	29H	)	084	54H	T	127	7FH	DEL
042	2AH	*	085	55H	U			

Dec=Decimal, Hex=Hexadecimal (H), CHR=Character.

LF=Line Feed, FF=Form Feed, CR=Carriage Return, DEL=Rubout

APPENDIX B
TABLE OF ASSEMBLER DIRECTIVES AND OPERATORS

B.1 MEMORY DIRECTIVES

The ASSUME statement

ASSUME segment register:segment name [,segment register:segment name...]
ASSUME NOTHING

The data allocation statements

name DB expression
name DD expression
name DQ expression
name DT expression
name DW expression

Symbols and their values

name EQU expression
name = expression

Symbols and their types

EXTRN name:type [,name:type...]
PUBLIC name [, name...]
name LABEL type
name PROC [NEAR]
name PROC [FAR]

Numbers and their notation

.RADIX expression

Addressing directives

EVEN
ORG expression

End statements

SEGMENT

.

ENDS

STRUCT

.

ENDS

program

.

END

Other directives

COMMENT delimiter text delimiter

NAME module name

B.2 MACRO DIRECTIVES

ENDM

EXITM

IRP dummy, parameters in angle brackets

IRPC dummy,string

LOCAL parameter [,parameter...]

name MACRO parameter [,parameter...]

PURGE macro-name [...]

REPT expression

Special Macro Operators

& (ampersand) - concatenation

<text> (angle brackets - single literal)

;; (double semicolons) - suppress comment

! (exclamation point) - next character literal

% (percent sign) - convert expression to number

B.3 CONDITIONAL DIRECTIVES

```

ELSE
ENDIF
IF expression
IFB argument
IFDEF symbol
IFDIF argument1,argument2
IFE expression
IFIDN argument1,argument2
IFNB argument
IFDEF symbol
IF1
IF2

```

B.4 LISTING DIRECTIVES

```

.CREF
.LALL
.LFCOND
.LIST
%OUT text
PAGE expression
.SALL
.SFCOND
SUBTTL text
.TFCOND
TITLE text
.XALL
.XCREF
.XLIST

```

B.5 ATTRIBUTE OPERATORSOverride Operators

```

Pointer (PTR)
    attribute PTR expression
Segment Override (:) (colon)
    segment-register:address-expression
    segment-name:address-expression
    group-name:address-expression
SHORT
    SHORT label
THIS
    THIS distance
    THIS type

```

Value Returning Operators

```

SEG
    SEG label
    SEG variable
OFFSET
    OFFSET label
    OFFSET variable
TYPE
    TYPE label
    TYPE variable
.TYPE
    .TYPE variable
LENGTH
    LENGTH variable
SIZE
    SIZE variable

```

Record Specific Operators

```

Shift-count - (Record fieldname)
    record-fieldname
MASK
    MASK record-fieldname
WIDTH
    WIDTH record-fieldname
    WIDTH record

```

B.6 PRECEDENCE OF OPERATORS

All operators in a single item have the same precedence, regardless of the order listed within the item. Spacing and line breaks are used for visual clarity, not to indicate functional relations.

1. LENGTH, SIZE, WIDTH, MASK
 Entries inside: parenthesis ()
 angle brackets < >
 square brackets []
 Structure variable operand: <variable>.<field>
2. Segment override operator: colon (:)
3. PTR, OFFSET, SEG, TYPE, THIS
4. HIGH, LOW
5. *, /, MOD, SHL, SHR
6. +, - (both unary and binary)

7. EQ, NE, LT, LE, GT, GE
8. Logical NOT
9. Logical AND
10. Logical OR, XOR
11. SHORT, .TYPE

APPENDIX C
ERROR MESSAGES

Numerical Order List of Error Messages

<u>Code</u>	<u>Message</u>
0	Block nesting error
1	Extra characters on line
2	Register already defined
3	Unknown symbol type
4	Redefinition of symbol
5	Symbol is multi-defined
6	Phase error between passes
7	Already had ELSE clause
8	Not in conditional block
9	Symbol not defined
10	Syntax error
11	Type illegal in context
12	Should have been group name
13	Must be declared in pass 1
14	Symbol type usage illegal
15	Symbol already different kind
16	Symbol is reserved word
17	Forward reference is illegal
18	Must be register
19	Wrong type of register
20	Must be segment or group
21	Symbol has no segment
22	Must be symbol type
23	Already defined locally
24	Segment parameters are changed
25	Not proper align/combine type
26	Reference to mult defined
27	Operand was expected
28	Operator was expected
29	Division by 0 or overflow
30	Shift count is negative
31	Operand types must match
32	Illegal use of external
33	Must be record field name
34	Must be record or field name
35	Operand must have size
36	Must be var, label or constant

<u>Code</u>	<u>Message</u>
37	Must be structure field name
38	Left operand must have segment
39	One operand must be const
40	Operands must be same or 1 abs
41	Normal type operand expected
42	Constant was expected
43	Operand must have segment
44	Must be associated with data
45	Must be associated with code
46	Already have base register
47	Already have index register
48	Must be index or base register
49	Illegal use of register
50	Value is out of range
51	Operand not in IP segment
52	Improper operand type
53	Relative jump out of range
54	Index displ. must be constant
55	Illegal register value
56	No immediate mode
57	Illegal size for item
58	Byte register is illegal
59	CS register illegal usage
60	Must be AX or AL
61	Improper use of segment reg
62	No or unreachable CS
63	Operand combination illegal
64	Near JMP/CALL to different CS
65	Label can't have seg. override
66	Must have opcode after prefix
67	Can't override ES segment
68	Can't reach with segment reg
69	Must be in segment block
70	Can't use EVEN on BYTE segment
71	Forward needs override
72	Illegal value for DUP count
73	Symbol already external
74	DUP is too large for linker
75	Usage of ? inate) bad (Code 75)
76	More values than defined with
77	Only initialize list legal
78	Directive illegal in STRUC
79	Override with DUP is illegal
80	Field cannot be overridden
81	Override is of wrong type
82	Register can't be forward ref
83	Circular chain of EQU aliases

<u>Code</u>	<u>Message</u>
101	Hard data
102	Device name
103	Operation
104	File system
105	Device offline
106	Lost file
107	File name
108	Device full
109	Unknown device
110	File not found
111	Protected file
112	File in use
113	File not open
114	Data format
115	Line too long

APPENDIX D
THE 8086 INSTRUCTION SET

AAA	AAA (no operands)	ASCII Adjust for Addition	
		AAA performs a correction of the result in AL of adding two unpacked decimal operands yielding an unpacked decimal sum.	
Operands	Bytes	Example	Flags
none	1	AAA	AF=A PF=U CF=A SF=U OF=U ZF=U

AAD	AAD (no operands)	ASCII Adjust for Division	
		AAD performs an adjustment of the dividend in AL before a subsequent instruction divides two unpacked decimal operands, so that the result of the division will be an unpacked decimal quotient.	
Operands	Bytes	Example	Flags
none	2	AAD	AF=U SF=A CF=U ZF=A OF=U PF=A

AAM	AAM (no operands)	ASCII Adjust for Multiply
-----	-------------------	---------------------------

AAM performs a correction of the result in AX of multiplying two unpacked decimal operands yielding an unpacked decimal product.

Operands	Bytes	Example	Flags
none	1	AAM	AF=U PF=A CF=U SF=A OF=U ZF=A

AAS	AAS (no operands)	ASCII Adjust for Subtraction.
-----	-------------------	-------------------------------

AAS performs a correction of the result in the AL register of subtracting two unpacked decimal operands, yielding an unpacked decimal difference.

Operands	Bytes	Example	Flags
none	1	AAS	AF=A PF=U CF=A SF=U OF=U ZF=U

ADC	ADC <u>destination</u> , <u>source</u>	Add with Carry
-----	---	----------------

ADC performs an addition of the two operands, adds one if the CF flag is set, and returns the result to the destination (left-most operands).

Operands	Bytes	Example	Flags
register, register	2	ADC AX, SI	AF=A PF=A
register, memory	2-4	ADC DX, BETA [SI]	CF=A SF=A
memory, register	2-4	ADC ALPHA [BX] [SI], DI	OF=A ZF=A
register, immediate	3-4	ADC BX, 256	
memory, immediate	3-6	ADC GAMMA, 30H	
accumulator, immediate	2-3	ADC AL, 5	

ADD ADD destination,
 source

Addition

ADD performs an addition of the two source operands and returns the result to the destination operands.

Operands	Bytes	Example	Flags
register, register	2	ADD CX, DX	AF=A PF=A
register, memory	2-4	ADD DI, [BX]. ALPHA	CF=A SF=A
memory, register	2-4	ADD TEMP, CL	OF=A ZF=A
register, immediate	3-4	ADD CL, 2	
memory, immediate	3-6	ADD ALPHA, 2	
accumulator, immediate	2-3	ADD AX, 200	

AND AND destination,
 source

Logical AND

AND performs the bitwise logical conjunction of the two source operands and returns the result to one of the operands.

Operands	Bytes	Example	Flags
register, register	2	AND AL, BL	AF=U PF=A
register, memory	2-4	AND CX, FLAG_WORD	CF=0 SF=A
memory, register	2-4	AND ASCII [DI],AL	OF=0 ZF=A
register, immediate	3-4	AND CX, OFOH	
memory, immediate	3-6	AND BETA, 01H	
accumulator, immediate	2-3	AND AX, 01010000B	

CALL CALL target

Call a Procedure

CALL pushes the offset address of the next instruction onto the stack (in the case of an inter-segment call the CS segment register is pushed first) and then transfers control to the target operand.

Operands	Bytes	Example	Flags
near-proc	3	CALL NEAR_PROC	none
far-proc	5	CALL FAR_PROC	
memptr 16	2-4	CALL PROC_TABLE [SI]	
regptr 16	2	CALL AX	
memptr 32	2-4	CALL [BX].TASK [SI]	

CBW	(no operands)	Convert Byte to Word
		CBW performs a sign extension of the AL register into the AH register.

Operands	Bytes	Example	Flags
none	1	CBW	none

CLC	CLC (no operands)	Clear Carry Flag
		CLC clears the CF flag.

Operands	Bytes	Example	Flags
none	1	CLC	CF=0

CLD	CLD (no operands)	Clear Direction Flag
		CLD clears the DF flag, causing the string operations to auto-increment the operand pointers.

Operands	Bytes	Example	Flags
none	1	CLD	DF=0

CLI	CLI (no operands)	Clear Interrupt Flag (Disable)
		CLI clears the IF flag, disabling maskable external interrupts, which appear on the INTR line of the processor. (Non-maskable interrupts, which appear on the NMI line, are not disabled.)

Operands	Bytes	Example	Flags
none	1	CLI	DF=0

CMC	CMC (no operands)	Complement Carry Flag
		CMC complements the CF flag.

Operands	Bytes	Example	Flags
none	1	CMC	CF=A

CMP CMP destination,
 source

Compare Two Operands

CMP performs a subtraction of the two operands causing the flags to be affected but does not return the result.

The source (right-most) operand must be of the same type (byte or word) as the destination operand. The only exception for CMP is comparing an immediate-data byte with a memory word.

Operands	Bytes	Example	Flags
register, register	2	CMP BX CX	AF=A PF=A
register, memory	2-4	CMP DH, ALPHA	CF=A SF=A
memory, register	2-4	CMP [BP +2], SI	OF=A ZF=A
register, immediate	3-4	CMP BL, 02H	
memory, immediate	3-6	CMP [BX].RADAR [DI], 3420H	
accumulator, immediate	2-3	CMP AL, 00010000B	

CMPS CMPS dest-string,
CMPSB source-string
CMPSW or
 CMPSB (no operands)
 or
 CMPSW (no operands)

Compare Byte or Word String

CMPS subtracts the type (or word) operand addressed by DI from the operand addressed by SI and affects the flags but does not return the result. As a repeated operation, this provides for comparing two strings. With the appropriate repeat prefix, it is possible to determine after which string element the two strings become unequal, thereby establishing an ordering between the strings.

Note that the operand indexed by DI is the right-most operand in this instruction, and that this operand is addressed using the ES register only. This default CANNOT be overridden.

Operands	Bytes	Example	Flags
dest-string, source-string	1	CMPS BUFF1, BUFF2	AF=A PF=A CF=A SF=A
(repeat) dest-string, source string	1	REPE CMPS ID, KEY	OF=A ZF=A

CWD	CWD (no operands)	Convert Word to Doubleword
		CWD performs a sign extension of the AX register into the DX register. See the DIV instruction in this chapter.

Operands	Bytes	Example	Flags
none	1	CWD	none

DAA	DAA (no operands)	Decimal Adjust for Addition
		DAA performs a correction of the result in AL of adding two packed decimal operands, yielding a packed decimal sum.

Operands	Bytes	Example	Flags
none	1	DAA	AF=A PF=A CF=A SF=A OF=A ZF=A

DAS	DAS (no operands)	Decimal Adjust for Subtraction
		DAS performs a correction of the result in the AL register of subtracting two packed decimal operands, yielding a packed decimal difference.

Operands	Bytes	Example	Flags
none	1	DAS	AF=A PF=A CF=A SF=A OF=U ZF=A

DEC	DEC <u>destination</u>	Decrement Destination by One
		DEC performs a subtraction of 1 from the operand and returns the result to that operand.

Operands	Bytes	Example	Flags
reg16	1	DEC AX	AF=A SF=A
reg8	2	DEC AL	OF=A ZF=A
memory	2-4	DEC ARRAY [SI]	PF=A

DIV

DIV source

Division, Unsigned

DIV performs an unsigned division of the double-length NUMR operand, contained in the accumulator and its extension (AL and AH for 8-bit operation, or AX and DX for 16-bit operation) by the DIVR operand, contained in the specified source operand. It returns the single-length quotient (QUO Operand) to the accumulator (AL or AX), and returns the single-length remainder (the REM operand) to the accumulator extension (AH for 8-bit operation or DC for 16-bit operation). If the quotient is greater than MAX (as when division by zero is attempted), QUO and REM are undefined, and a type 0 interrupt is generated. Flags are undefined in any DIV operation. Non-integral quotients are truncated to integers.

Operands	Bytes	Example	Flags
reg8	2	DIV CL	AF=U PF=U
reg16	2	DIV BX	CF=U SF=U
mem8	2-4	DIV ALPHA	OF=U ZF=U
mem16	2-4	DIV TABLE [SI]	

ESC

ESC external-opcode,
source

Escape

The ESC instruction provides a mechanism by which other processors may receive their instructions from the instruction stream and make use of the addressing modes. The processor does no operation for the ESC instruction other than to access a memory operand and place it on the bus.

Operands	Bytes	Example	Flags
immediate, memory	2-4	ESC 6, ARRAY [SI]	none
immediate, register	2	ESC 20, AL	

HLT HLT (no operands) Halt

The HLT instruction causes the processor to enter its halt state.

Operands	Bytes	Example	Flags
none	1	HLT	none

IDIV IDIV source Integer Division, Signed

IDIV performs a signed division of the double-length NUMR operand, contained in the accumulator and its extension (AL and AH for 8-bit operation, or AX and DX for 16-bit operation) by the DIVR operand, contained in the specified source operand. It returns the single-length quotient (QUO operand) to the accumulator (AL or AX), and returns the single-length remainder (the REM operand) to the accumulator extension (AH for 8-bit operation or DX for 16-bit operation). If the quotient is positive and greater than MAX or if the quotient is negative and less than 0-MAX-1, (as when division by zero is attempted) then QUO and REM are undefined, and a type 0 interrupt is generated. Flags are undefined in any zero operation. IDIV truncates non-integral quotients and returns a remainder with the same sign as the numerator.

Operands	Bytes	Example	Flags
reg8	2	IDIV BL	AF=U PF=U
reg16	2	IDIV CX	CF=U SF=U
mem8	2-4	IDIV DIVISOR_BYTE [SI]	OF=U ZF=U
mem16	2-4	IDIV [BX].DIVISOR_WORD	

IMUL IMUL source

Integer Multiply

IMUL performs a signed multiplication of the accumulator (AL or AX) and the source operand, returning a double-length result to the accumulator and its extension (AL and AH for 8-bit operation, or AX and DX for 16-bit operation). CF and OF are set if the top half of the result is not the sign-extension of the low half of the result.

Operands	Bytes	Example	Flags
reg8	2	IMUL CL	AF=U PF=U
reg16	2	IMUL BX	CF=A SF=U
mem8	2-4	IMUL RATE_BYTE	OF=A ZF=U
mem16	2-4	IMUL RATE_WORD[BP][DI]	

IN IN accumulator, source

Input Byte or Word

IN transfers a byte/word from an input port to the AL register (or AX register). The port is specified either with an inline data byte, allowing fixed access to ports 0 through 255, or with a port number in the DX register, allowing variable access to 64K input ports.

Operands	Bytes	Example	Flags
accumulator, immed8	2	IN AL, OFFEAH	none
accumulator, DX	1	IN AX,DX	

INC INC destination

Increment Destination by One

INC performs an addition of the operand and one, and returns the result to the operand.

Operands	Bytes	Example	Flags
reg16	1	INC CX	AF=A SF=A
reg8	2	INC BL	OF=A ZF=A
memory	2-4	INC ALPHA [DI][BX]	PF=A

INT INT interrupt-type Interrupt

INT pushes the flag registers (as in PUSHF), clears the TF and IF flags, and transfers control with an indirect call through any one of the 256 vector elements. The one-byte form of this instruction generates a type 3 interrupt.

Operands	Bytes	Example	Flags
immed8(type=3)	1	INT 3	IF=0
immed8(type≠3)	2	INT 67	TF=0

INTO INTO (no operands) Interrupt If Overflow

INTO pushes the flag registers (as in PUSHF), clears the TF and IF flags, and transfers control with an indirect call through vector element 4 (location 10H) if the OF flag is set (trap on overflow). If the OF flag is clear, no operation takes place.

Operands	Bytes	Example	Flags
none	1	INTO	IF=0 TF=0

IRET IRET (no operands) Interrupt Return

IRET transfers control the the return address saved by a previous interrupt operation and restores the saved flag registers (as in POPF).

Operands	Bytes	Example	Flags
none	1	IRET	SF=R PF=R CF=R SF=R DF=R TF=R IF=R ZF=R

JA	JA <u>short-label</u>	Jump If Above/
JNBE	or	If Not Below or Equal
	JNBE <u>short-label</u>	

JA or JNBE transfers control to the target operand on not below or equal (or above).

Operands	Bytes	Example	Flags
short-label	2	JA ABOVE	none

JAE	JAE <u>short-label</u>	Jump If Above or Equal/
JNB	or	If Not Below
	JNB <u>short-label</u>	

JNB or JAE transfers control to the target operand on not below (or above or equal).

Operands	Bytes	Example	Flags
short-label	2	JAE ABOVE_EQUAL	none

JB	JB <u>short-label</u>	Jump If Below/
JNAE	or	If Not Above nor Equal/
JC	JNAE <u>short-label</u>	If Carry
	or	
	JC <u>short-label</u>	

JB, JNAE or JC transfers control to the target operand on below (or not above or equal).

JB/JNAE short-label - Jump if below/Jump if not above nor equal

Operands	Bytes	Example	Flags
short-label	2	JB BELOW	none

JC short-label - Jump if carry

Operands	Bytes	Example	Flags
short-label	2	JC CARRY_SET	none

JBE JBE short-label
 JNA or
 JNA short-label

Jump If Below or Equal/
 If Not Above

JBE or JNA transfers control to the target operand on below or equal (or not above).

Operands	Bytes	Example	Flags
short-label	2	JNA NOT_ABOVE	none

JCXZ JCXZ short-label

Jump If CS Is Zero

JCXZ transfers control to the target operand if the CX register is 0.

Operands	Bytes	Example	Flags
short-label	2	JCXZ COUNT_DONE	none

JE JE short-label
 JZ or
 JZ short-label

Jump If Equal/
 If Zero

JE or JZ transfers control to the target operand on equal (or zero).

Operands	Bytes	Example	Flags
short-label	2	JZ ZERO	none

JG JG short-label
 JNLE or
 JNLE short-label

Jump If Greater/
 If Not Less nor Equal

JG or JNLE transfers control to the target operand on not less or equal (or greater).

Operands	Bytes	Example	Flags
short-label	2	JG GREATER	none

JGE	JGE <u>short-label</u>	Jump If Greater or Equal/
JNL	or	If Not Less
	JNL	

JGE or JNL transfers control to the target operand on not less (or greater or equal).

Operands	Bytes	Example	Flags
short-label	2	JGE GREATER_EQUAL	none

JL	JL <u>short-label</u>	Jump If Less/
JNGE	or	If Not Greater nor Equal
	JNGE	

JL or JNGE transfers control to the target operand on less (or not greater or equal).

Operands	Bytes	Example	Flags
short-label	2	JL LESS	none

JLE	JLE <u>short-label</u>	Jump If Less or Equal/
JNG	or	If Not Greater
	JNG <u>short-label</u>	

JLE or JNG transfers control to the target operand on less or equal (or not greater).

Operands	Bytes	Example	Flags
short-label	2	JNG NOT_GREATER	none

JMP	JMP <u>target</u>	Jump
		JMP transfers control to the target operand.

Operands	Bytes	Example	Flags
short-label	2	JMP SHORT	none
near-label	3	JMP WITHIN_SEGMENT	
far-label	5	JMP FAR_LABEL	
memptr16	2-4	JMP [BX].TARGET	
regptr16	2	JMP CX	
memptr32	2-4	JMP OTHER.SEG [SI]	

JNC JC short-label Jump if No Carry

JC transfers control to the target operand on below or equal (or not above).

Operands	Bytes	Example	Flags
short-label	2	JNC NOT_CARRY	none

JNE JNE short-label
JNZ or
JNZ short-label

Jump If Not Equal/
If Not Zero

JNE (or JNZ) transfers control to the target operand on not equal (or not zero).

Operands	Bytes	Example	Flags
short-label	2	JNE NOT_EQUAL	none

JNO JNO short-label

Jump If No Overflow

JNO transfers control to the target operand on no overflow.

Operands	Bytes	Example	Flags
short-label	2	JNO NO_OVERFLOW	none

JNP JNP short-label
JPO or
JPO short-label

If No Parity/
If Parity Odd

JNP or JPO transfers control to the target operand on not parity (or parity odd).

Operands	Bytes	Example	Flags
short-label	2	JPO ODD_PARITY	none

JNS JNS short-label

Jump If No Sign/
If Positive

JNS transfers control to the target operand on not sign.

Operands	Bytes	Example	Flags
short-label	2	JNS POSITIVE	none

JO	JO <u>short-label</u>	Jump On Overflow		
JO transfers control to the target operand on overflow.				
Operands		Bytes	Example	Flags
short-label		2	JO SIGNED_OVRFLW	none

JP JPE	JP <u>short-label</u> or JPE	Jump on Parity/ If Parity Even		
JP or JPE transfers control to the target operand on parity (or parity even).				
Operands		Bytes	Example	Flags
short-label		2	JPE EVEN_PARITY	none

JS	JS <u>short-label</u>	Jump On Sign		
JS transfers control to the target operand on sign.				
Operands		Bytes	Example	Flags
short-label		2	JS NEGATIVE	none

LAHF	LAHF (no operands)	Load AH from Flags		
LAHF transfers the flag registers SF, ZF, AF, PF, and CF into specific bits of the AH register.				
Operands		Bytes	Example	Flags
none		1	LAHF	none

LDS

LDS destination,
source

Load Data Segment Register

LDS transfers a "pointer-object" (a 32-bit object containing an offset address and a segment address) from the source operand (which must be a doubleword memory operand) to a pair of destination registers. The segment address is transferred to the DS segment register. The offset address may be transferred to any 16-bit general, pointer, or index register you specify (not a segment register).

Operands

Bytes

Example

Flags

reg16, mem32

2-4

LDS SI, DATA.SEG[DI]

none

LEA

LEA destination,
source

Load Effective Address

LEA transfers the offset address of the source operand to the destination operand. The source operand must be a memory operand and the destination operand can be any 16-bit general, pointer, or index register. LEA allows the source to be subscripted. This is not allowed using the MOV instruction with the OFFSET operator. Also, the later operation invariably uses the offset of the variable in the segment where it was defined. LEA, however, will take into account a group offset if the group is the only possible access route via the latest ASSUME directive.

Operands

Bytes

Example

Flags

reg16, mem16

2-4

LEA BX,[BP][DI]

none

LES	LES <u>destination</u> , <u>source</u>	Load Extra Segment Register
-----	---	-----------------------------

LES transfers a "pointer object" (a 32-bit object containing an offset address and a segment address) from the source operand (which must be a doubleword memory operand) to a pair of destination registers. The segment address is transferred to the ES segment register. The offset address may be transferred to a 16-bit general, pointer, or index register (not a segment register).

Operands	Bytes	Example	Flags
reg16, mem32	2-4	LES DI,[BX].TEXT_BUFF	none

LOCK	LOCK (no operands)	Lock Bus
------	--------------------	----------

A special one-byte lock prefix may precede any instruction. It causes the processor to assert its bus-lock signal for the duration of the operation caused by the instruction. In multiple processor systems with shared resources it is necessary to provide mechanisms to enforce controlled access to those resources. Such mechanisms, while generally provided through software operating systems, require hardware assistance. A sufficient mechanism for accomplishing this is a locked exchange (also referred to as test-and-set-lock).

Operands	Bytes	Example	Flags
none	1	LOCK XCHG FLAG,AL	none

LODS	LODS source-string	Load Byte or Word String
LODSB	or	
LODSW	LODSB (no operands)	LODS transfers a byte (or word)
	or	operand from the source operand
	LODSW (no operands)	addressed by SI to accumulator AL
		(or AX) and adjusts the SI
		register by DELTA. This operation
		ordinarily would not be repeated.

Operands	Bytes	Example	Flags
source-string	1	LODS CUSTOMER_NAME	none
(repeat) source-string	1	REP LODS NAME	

LOOP	LOOP <u>short-label</u>	Loop until Count Complete
		LOOP decrements the CX (count)
		register by 1 and transfers
		control to the target operand
		(short label) if CX is not zero.

Operands	Bytes	Example	Flags
short-label	2	LOOP AGAIN	none

LOOPE	LOOPE <u>short-label</u>	Loop If Equal/
LOOPZ	or	If Zero
	LOOPZ <u>short-label</u>	
		LOOPE or LOOPZ decrements the CX
		register by one and transfers if
		CX is not zero and if the ZF flag
		is set to 1.

Operands	Bytes	Example	Flags
short-label	2	LOOPE AGAIN	none

LOOPNE	LOOPNZ <u>short-label</u>	Loop If Not Equal/
LOOPNZ	or	If Not Zero
	LOOPNE <u>short-label</u>	
		LOOPNZ or LOOPNE decrements the CX
		register by one and transfers if
		CX is not zero and the ZF flag is
		cleared.

Operands	Bytes	Example	Flags
short-label	2	LOOPNE AGAIN	none

MOV

MOV destination,
source

Move

The seven separate types of move instructions are described below.

Each type has multiple uses and encodings depending on the type of data being moved and the location of that data. The assembler generates the correct encoding based on these two factors.

Operands	Bytes	Example	Flags
memory, accumulator	3	MOV ARRAY [SI], AL	none
accumulator, memory	3	MOV AX, TEMP_RESULT	
register, register	2	MOV AX, CX	
register, memory	2-4	MOV BP, STACK_TOP	
memory, register	2-4	MOV COUNT [DI], CX	
register, immediate	2-3	MOV CL, 2	
memory, immediate	3-6	MOV MASK [BX][SI], 2CH	
seg-reg, reg16	2	MOV ES, CX	
seg-reg, mem16	2-4	MOV DS, SEGMENT_BASE	
reg16, seg-reg	2	MOV BP, SS	
memory, seg-reg	2-4	MOV [BX].SEG_SAVE, CS	

MOVS	MOVS <u>dest-string</u> ,	Move Byte or Word String
MOVSB	<u>source-string</u>	
MOVSW	or	MOVS transfers a byte (or word)
	MOVSB (no operands)	operand from the <u>source</u> operand
	or	addressed by SI to the <u>destination</u>
	MOVSW (no operands)	operand addressed by DI, and
		adjusts the SI and DI registers by
		DELTA (number of bytes specified
		by the operand TYPE). As a
		repeated operation this provides
		for moving a string from one
		location in memory to another.

MOVS dest-string, source-string - MOVE string

Operands	Bytes	Example	Flags
dest-string, source-string	1	MOVS LINE EDIT_DATA	none
(repeat) dest-string, source-string	1	REP MOVS SCREEN, BUFFER	

MOVSB/MOVSW no operands - Move string (byte/word)

Operands	Bytes	Example	Flags
none	1	MOVSB	none
(repeat) none	1	REP MOVSW	

MUL MUL source Multiply, Unsigned

MUL performs an unsigned multiplication of the accumulator (AL or AX) and the source operand, returning a double-length result to the accumulator and its extension (AL and AH for 8-bit operation, or AX and DX for 16-bit operation). CF and OF are set if the top half of the result is non-zero.

Operands	Bytes	Example	Flags
reg8	2	MUL BL	AF=U PF=U
reg16	2	MUL CX	CF=A SF=U
mem8	2-4	MUL MONTH [SI]	OF=A ZF=U
mem16	2-4	MUL BAUD_RATE	

NEG NEG destination Negate/
Form 2's Complement

NEG performs a subtraction of the operand from zero, adds 1, and returns the result to the operand. This forms the 2's complement of the specified operand.

Operands	Bytes	Example	Flags
register	2	NEG AL	AF=A PF=A
memory	2-4	NEG MULTIPLIER	CF=1* SF=A
			OF=A ZF=A

*0 if destination = 0

NOP NOP (no operands) No Operation

NOP causes no operation.

Operands	Bytes	Example	Flags
none	1	NOP	none

NOT NOT destination NOT forms the ones complement of the operand and returns the result to the operand. Flags are not affected.

Operands	Bytes	Example	Flags
register	2	NOT AX	none
memory	2-4	NOT CHARACTER	

OR OR destination,
source Logical Inclusive OR

OR performs the bit logical inclusive disjunction of the two operands and returns the result to one of the operands.

Operands	Bytes	Example	Flags
register, register	2	OR AL, BL	AF=U PF=A
register, memory	2-4	OR DX, PORT ID [DI]	CF=0 SF=A
memory, register	2-4	OR FLAG BYTE, CL	OF=0 ZF=A
accumulator, immediate	2-3	OR CS, 01FH	
register, immediate	3-4	OR CX, 01FH	
memory, immediate	3-6	OR[BS] SMD WORD,OCFH	

OUT	OUT <u>port</u> , <u>accumulator</u>	Output Byte or Word
-----	---	---------------------

OUT transfers a byte (or word) from the AL register (or AX register) to an output port. The port is specified either with an inline data byte, allowing fixed access to ports 0 through 255, or with a port number in the DX register, allowing variable access to 64K output ports.

Operands	Bytes	Example	Flags
immed8, accumulator	2	OUT 44, AX	none
DX, accumulator	1	OUT DX, AL	

POP	POP <u>destination</u>	POP Word Off Stack to Destination
-----	------------------------	-----------------------------------

POP transfers a word operand from the stack element addressed by the SP register to the destination operand and then increments SP by 2. There are three separate types of POP instructions, for different destinations.

Operands	Bytes	Example	Flags
register	1	POP DX	none
seg-reg (CS illegal)	1	POP DS	
memory	2-4	POP PARAMETER	

POPF	POPF (no operands)	POPF transfers specific bits of the stack element addressed by the SP register to the flag registers and then increments SP by 2.
------	--------------------	---

Operands	Bytes	Example	Flags
none	1	POPF	AF=R PF=R CF=R SF=R DF=R TF=R IF=R ZF=R

PUSH

PUSH source

Push Word onto Stack

PUSH decrements the stack pointer SP by 2 and then transfers a word from the service operand to the stack element currently addressed by SP. There are three separate types of PUSH instructions depending on the kind of operand supplied.

Operands	Bytes	Example	Flags
register	1	PUSH SI	none
seg-reg (CS legal)	1	PUSH ES	
memory	2-4	PUSH RETURN_CODE [SI]	

PUSHF

PUSHF (no operands)

PUSHF decrements the SP register by 2 and transfers all of the flag registers into specific bits of the word operand (stack element) addressed by SP.

Operands	Bytes	Example	Flags
none	1	PUSHF	none

RCL

RCL destination,
source

Rotate Left through Carry

RCL rotates the operand left through the CF flag register by COUNT bits.

Operands	Bytes	Example	Flags
register, 1	2	RCL CX, 1	CF=A
register, CL	2	RCL AL, CL	OF=A
memory, 1	2-4	RCL ALPHA, 1	
memory, CL	2-4	RCL [BP].PARM, CL	

RCR	<u>destination</u> , <u>count</u>	Rotate Right through Carry
		RCR rotates in EA operand right through the CF flag register by <u>count</u> bits.

Operands	Bytes	Example	Flags
register, 1	2	RCR BX, 1	CF=A
register, CL	2	RCR BL, CL	OF=A
memory, 1	2-4	RCR [BX]. STATUS, 1	
memory, CL	2-4	RCR ARRAY [DI], CL	

REP	REP (no operands)	Repeat String Operation
REPZ	or	
REPE	REPZ (no operands)	REP causes the primitive string operation that follows to be performed repeatedly while (CX) is not zero. In the case of CMPS and SCAS, if after any repetition of the primitive operation the ZF flag differs from the "z" bit of the repeat prefix, the repetition is terminated. This prefix may be combined with the segment override and/or LOCK prefixes, although with multiple prefixes, interrupts must be disabled, because the return from an interrupt returns control to the interrupted instruction or to at most one prefix byte before that instruction.
REPNE	or	
REPZ	REPE (no operands)	
	or	
	REPNE (no operands)	
	or	
	REPZ (no operands)	

REP (no operands) - Repeat string operation

Operands	Bytes	Example	Flags
none	1	REP MOVS DEST,SRCE	none

REPE/REPZ (no operands) - Repeat string operation while equal/while zero

Operands	Bytes	Example	Flags
none	1	REPE CMPS DATA,KEY	none

REPNE/REPZ (no operands) - Repeat string operation while not equal/not zero

Operands	Bytes	Example	Flags
none	1	REPNE SCAS INPUT_LINE	none

RET

RET [pop-value]

Return from Procedure

RET transfers control to the return address pushed by a previous CALL operation and optionally adds an immediate constant, pop-value, to the SP register to discard stack parameters. If this is an inter-segment RET, it was assembled under a procedure labeled FAR, it will replace the IP and the CS using the two words at the top of the stack. Otherwise, only the IP is replaced, using only one word from the top of the stack.

When using indirect CALLs, you must carefully ensure that the type of CALL matches the type of return in the procedure.

CALL WORD PTR [BX]

must not invoke a FAR procedure.
And,

CALL DWORD PTR [BX]

must not invoke a NEAR procedure.

Operands	Bytes	Example	Flags
(intra-segment, no pop)	1	RET	none
(intra-segment, pop)	3	RET 4	
(inter-segment, no pop)	1	RET	
(inter-segment, pop)	3	RET 2	

ROL

ROL destination,
count

Rotate Left

ROL rotates the operand left by count bits.

Operands	Bytes	Example	Flags
register, 1	2	ROL BX, 1	CF=A
register, CL	2	ROL DI, CL	OF=A
memory, 1	2-4	ROL FLAG_BYTE[DI], 1	
memory, CL	2-4	ROL ALPHA, CL	

ROR	ROR <u>destination</u> ,	Rotate Right
	<u>count</u>	
		ROR rotates the operand right by
		count bits.

Operands	Bytes	Example	Flags
register, 1	2	ROR AL, 1	CF=A
register, CL	2	ROR BX, CL	OF=A
memory, 1	2-4	ROR PORT_STATUS, 1	
memory, CL	2-4	ROR CMD_WORD, CL	

SAHF	SAHF (no operands)	SAHF transfers specific bits of the AH register to the flag registers SF, ZF, AF, PF, and CF. The bits of AH indicated by "X" in the operation are ignored.
------	--------------------	---

Operands	Bytes	Example	Flags
none	1	SAHF	AF=R CF=R ZF=R PF=R

SAL	<u>destination</u> ,	Shift Arithmetic Left/
SHL	<u>count</u>	Shift Logical Left
	or	
	SHL <u>destination</u> ,	SAL and SHL shift the operand left
	<u>count</u>	by <u>count</u> bits, shifting in low-
		order zero bits.

Operands	Bytes	Example	Flags
register, 1	2	SAL AL, 1	CF=A
register, CL	2	SHL DI, CL	OF=A
memory, 1	2-4	SHL [BX], OVERDRAW, 1	
memory, CL	2-4	SAL STORE COUNT, CL	

SAR	SAR <u>destination</u> , <u>count</u>	Shift Arithmetic Right
-----	--	------------------------

SAR (shift arithmetic right) shifts the destination operand right by count bits, shifting in high-order bits equal to the original high-order bit of the operand (sign extension).

Operands	Bytes	Example	Flags
register, 1	2	SAR DX, 1	AF=A PF=A
register, CL	2	SAR DI, CL	CF=A SF=A
memory, 1	2-4	SAR N_BLOCKS, 1	OF=A ZF=A
memory, CL	2-4	SAR N_BLOCKS, CL	

SBB	SBB <u>destination</u> , <u>source</u>	Subtract with Borrow
-----	---	----------------------

SBB performs a subtraction of the two operands, subtracts one if the CG flag is set, and returns the result to one of the operands.

Operands	Bytes	Example	Flags
register, register	2	SBB BX, CX	AF=A PF=A
register, memory	2-4	SBB DI, [BX].PAYMENT	CF=A SF=A
memory, register	2-4	SBB BALANCE, AX	OF=A ZF=A
accumulator, immediate	2-3	SBB AX, 2	
register, immediate	3-4	SBB CL, 1	
memory, immediate	3-6	SBB COUNT[SI], 10	

SCAS SCASB SCASW	SCAS <u>dest-string</u> or SCASB (no operands) or SCASW (no operands)	Scan Byte or Word String
------------------------	---	--------------------------

SCAS subtracts the destination byte (or word) operand addressed by DI from AL (or AX) and affects the flags but does not return the result. As a repeated operation this provides for scanning for the occurrence or, or departure from, a given value in a string. See REP instruction in this chapter.

Operands	Bytes	Example	Flags
dest-string	1	SCAS INPUT_LINE	AF=A PF=A
(repeat) dest-string	1	REPNE SCAS BUFFER	CF=A SF=A
			OF=A ZF=A

SHR	SHR <u>destination</u> , <u>count</u>	Shift Logical Right	
		SHR shifts the operand right shifting in high-order zero bits.	
Operands	Bytes	Example	Flags
register, 1	2	SHR SI, 1	CF=A
register, CL	2	SHR SI, CL	OF=A
memory, 1	2-4	SHR ID_BYTE [SI][BX], 1	
memory, CL	2-4	SHR INPUT_WORD, CL	
STC	STC (no operands)	Set Carry Flag	
		STC sets the CG flag.	
Operands	Bytes	Example	Flags
none	1	STC	CF=1
STD	STD (no operands)	Set Direction Flag	
		STD sets the DF flag causing the string operations to auto-decrement the operand index(es).	
Operands	Bytes	Example	Flags
none	1	STD	DF=1
STI	STI (no operands)	Set Interrupt Flag (Enable)	
		STI sets the IF flag, enabling maskable external interrupts after the execution of the next instruction.	
Operands	Bytes	Example	Flags
none	1	STI	IF=1

STOS	STOS <u>destination-</u> <u>string</u>	Store Byte or Word String
	or	STOS transfers a byte (or word)
	STOSB (no operands)	operand from AL (or AX) to the
	or	<u>destination</u> operand addressed by
	STOSW (no operands)	DI and adjusts the DI register by
		DELTA. As a repeated operation
		(see REP) this provides for
		filling a string with a given
		value. The operand named in the
		STOS instruction is used only by
		the assembler to verify type and
		accessibility using current
		segment register contents. The
		actual operation of the
		instruction uses only DI to point
		to the location being stored into.

Operands	Bytes	Example	Flags
dest-string	1	STOS PRINT_LINE	none
(repeat) dest-string	1	REP STOS DISPLAY	

SUB	SUB <u>destination,</u> <u>source</u>	Subtract
		SUB performs a subtraction of the
		<u>source</u> from the <u>destination</u> . The
		result goes to the <u>destination</u>
		operand.

Operands	Bytes	Example	Flags
register, register	2	SUB CX, BX	AF=A PF=A
register, memory	2-4	SUB DX, MATH_TOTAL[SI]	CF=A SF=A
memory, register	2-4	SUB [BP+2], CL	OF=A ZF=A
accumulator, immediate	2-3	SUB AL, 10	
register, immediate	3-4	SUB SI, 5280	
memory, immediate	3-6	SUB [BP].BALANCE, 1000	

TEST TEST destination,
 source

Test (logical Compare)

TEST performs the bit-to-bit logical conjunction of the two operands, causing the flags to be affected, but does not return the result.

The source (right-most) operand must be of the same type (byte or word) as the destination operand. The only exception for TEST is testing an immediate-data byte with a memory word.

Operands	Bytes	Example	Flags
register, register	2	TEST SI, DI	AF=U PF=A
register, memory	2-4	TEST SI, END_COUNT	CF=0 SF=A
accumulator, immediate	2-3	TEST AL, 00100000B	OF=0 ZF=A
register, immediate	3-4	TEST BX, 0CC4H	
memory, immediate	3-6	TEST RETURN_CODE, 01H	

WAIT WAIT (no operands)

Wait

This instruction allows the processor to synchronize itself with external hardware by placing the processor in a wait state until an external interrupt occurs.

Operands	Bytes	Example	Flags
none	1	WAIT	none

XCHG XCHG destination,
 source

Exchange

XCHG exchanges the byte or word source operand with the destination operand.

Operands	Bytes	Example	Flags
accumulator, reg16	1	XCHG AX, BX	none
memory, register	2-4	XCHG SEMAPHORE, AX	
register, register	2	XCHG AL, BL	

XLAT

XLAT source-table

Translate

XLAT performs a table lookup byte translation. The AL register is used as an index into a table (256-bytes at most) addressed by the BX register. The byte operand so addressed is transferred to AL.

Operands	Bytes	Example	Flags
source-table	1	XLAT ASCII_TAB	none

XOR

XOR destination,
source

Exclusive OR

XOR performs the bit-to-bit logical exclusive disjunction of the operands and returns the result to the destination operand.

Operands	Bytes	Example	Flags
register, register	2	XOR CX, BX	AF=U PF=A
register, memory	2-4	XOR CL, MASK_BYTE	CF=0 SF=A
memory, register	2-4	XOR ALPHA [SI], DX	OF=0 ZF=A
accumulator, immediate	2-3	XOR AL, 010000010B	
register, immediate	3-4	XOR SI, 00C2H	
memory, immediate	3-6	XOR RETURN_CODE 0D2H	

INDEX

A

Address, 1-2, 2-1, 2-3, 2-4, 3-1,
3-2, 3-4, 3-6, 3-10, 3-11,
4-1, 4-4, 4-5, 4-8 to 4-11,
5-2, 5-3, 5-5, 5-6, 5-8,
6-1, 6-3 to 6-7, 7-3, 10-3,
10-9 to 10-11, 11-2, 11-3,
B-3, D-3, D-10, D-16, D-17,
D-25
Addressing modes, 2-3, 2-5, D-7
Addressing through Base and
Index Registers Plus an
Offset, 2-4
Align-type of segments, 6-4,
10-10
Ampersand, 7-7, B-2
AND, 4-3, 4-4
Angle Brackets, 4-14, 5-11, 7-2,
7-6 to 7-8, 8-3, 8-4, B-2,
B-4
Arithmetic operators, 4-1
ASCII Character Codes, A-1
Assembler
errors, 1-1, 1-2, 1-10, 2-5,
6-4, 7-8, 8-1, 10-1, 10-2,
10-6, 11-1, 11-2,
11-7 to 11-9, 12-5
loading, 1-3 to 1-9
starting, 1-3 to 1-9
Assembler, invoking of, 1-3 to
1-5
Assembler, operation of, 1-3
ASSUME, 3-4, 3-6, 4-5, 4-9,
4-10, 5-1, 5-2, 5-8, 6-3,
6-6, 6-7, 7-4, 10-2, 10-4,
10-5, 11-1, 11-3, 11-5,
B-1, D-16
ASSUME NOTHING, 5-2, B-1
At Combine Type, 6-4, 10-2, 10-3,
10-13
Attribute, 10-12
Attributal operators, 4-4
Auxiliary carry flag, 2-5

B

Base addresses, 2-1, 6-4, 10-9
Base registers, 2-4
Block nesting error, 11-3, C-1
Byte Align Type, 6-4, 10-10

C

Carry flag, 2-5
Character string, 3-8
Class Name, 6-6
Clear direction flag, 2-5, 2-6
Colon - segment override
operator, 4-9 to 4-14
Combine type, 3-3, 6-3 to 6-5,
10-1, C-1
Command characters, 12-3, 1-9
Command statement, the elements
of, 3-6 to 3-9
Comments, 3-6, 3-8, 7-8, 9-6,
11-3
COMMENT directive, 3-8, 9-6
COMMON combine type, 6-4, 10-11
Conditional directives, 1-1,
4-2, 8-1 to 8-3, B-3
Constants, 2-2, 3-2, 3-6,
3-8 to 3-10, 4-1, 5-1,
5-3, 5-4, 5-9, 7-6
Control Characters
carriage return, 1-9
comma, 1-9
CONTROL-C, 12-3, 12-4
2ND-COMMAND-CANCEL, 12-3
semicolon, 1-8, 1-9, 3-2, 3-8,
7-7, 7-8, 11-3, 12-2, 12-3
SHIFT-CANCEL, 1-9
Control characters for assembly,
1-8
Count, shift, 4-11, 4-12, 5-9,
10-8, 10-9, 11-7, B-4, C-1
B-3
CREF, 9-6, 10-1, 12-1 to 12-6,
12-8, B-3
.CREF, 9-6, 12-1 to 12-5, 12-8,
B-3

INDEX (continued)

Cross-reference file, 1-2, 1-4,
1-6, 1-9, 9-4, 9-6, 9-7,
10-1, 12-1 to 12-8
command characters, 12-3
error messages, 1-10, 11-1,
11-8, 11-9, 12-5, C-1
format, 12-6
invoking, 12-2

D

Data allocation statement, 5-1,
5-2, 5-10, 5-11, 5-13
Data items, 3-9
constants, 2-2, 3-2, 3-6,
3-8 to 3-10, 4-1, 5-1, 5-3,
5-4, 5-9, 7-6
labels, 1-2, 3-6, 3-7, 3-9,
3-10, 4-8 to 4-10, 5-1,
5-8, 6-2, 6-4, 7-3, 11-9
variables, 1-2, 2-2, 3-4,
3-6 to 3-10, 4-1, 4-2, 4-5,
4-8, 4-9, 5-9, 5-12, 6-4,
6-5, 7-3, 9-7, 10-4
Default extension names, 1-4
DB, 3-5 to 3-7, 5-1 to 5-3, 5-9,
5-12, 5-13, 6-5,
7-4 to 7-7, 7-9, 10-4,
11-1, B-1
DD, 3-6, 4-5, 4-5, 5-1 to 5-3,
5-9, 5-12, 5-1 to 5-3, 5-9,
5-12, B-1, B-1
DQ, 3-6, 5-1 to 5-3, 5-9, 5-12,
B-1
DT, 3-6, 5-1 to 5-3, 5-9, 5-12,
B-1
DW, 3-4 to 3-7, 4-5 to 4-7, 4-9,
5-1 to 5-3, 5-9, 5-12, 6-2,
6-6, 7-4, 10-2, 11-2, 11-5,
11-6, B-1
Direction flag, 2-6
Direct offset addressing, 2-3
Directives, 1-1, 1-5, 3-1, 3-5,
3-9, 4-2, 4-11, 5-1, 5-7,
5-12, 6-1, 6-3, 8-1 to 8-3,
9-1, 9-3 to 9-6, 10-11,
10-12, 11-3, B-1 to B-3
ASSUME, 3-4, 3-6, 4-5, 4-9,
4-10, 5-1, 5-2, 5-8, 6-3,
6-6, 6-7, 7-4, 10-2, 10-4,
10-5, 11-1, 11-3, 11-5, B-1

ASSUME NOTHING, 5-2, B-1
COMMENT, 3-5, 3-6, 3-8, 7-8,
9-6, B-2
.CREF, 9-6, 12-1 to 12-5,
12-8, B-3
DB, 3-5 to 3-7, 5-1 to 5-3,
5-9, 5-12, 5-13, 6-5,
7-4 to 7-7, 7-9, 10-4,
11-1, B-1
DD, 3-6, 4-5, 4-5, 5-1 to 5-3,
5-9, 5-12, 5-1 to 5-3, 5-9,
5-12, B-1, B-1
DQ, 3-6, 5-1 to 5-3, 5-9,
5-12, B-1
DT, 3-6, 5-1 to 5-3, 5-9,
5-12, B-1
DW, 3-4 to 3-7, 4-5 to 4-7,
4-9, 5-1 to 5-3, 5-9,
5-12, 6-2, 6-6, 7-4, 10-2,
11-2, 11-5, 11-6, B-1
ELSE, 8-1, 11-2, 11-3, 11-6,
B-3
END, 3-2, 3-3, B-2
ENDS, 3-2, 3-3, B-2
ENDC, 8-1 to 8-4
ENDIF, 8-1, 9-4, 11-6, B-3
ENDM, 7-1, 7-4 to 7-7, 7-9,
10-4, B-2
ENDP, 5-1, 5-7, 5-8, 7-4, 10-5
EQU, 3-5, 3-9, 3-10, 4-9, 5-1,
5-4, 5-7, 6-2, 7-9, 10-4,
10-5, 10-11, 10-12, 11-3,
B-1
Equal sign, 5-4
EVEN, 5-5, B-1
EXITM, B-2, 6-2, 6-3
EXTRN, 5-6, 6-1 to 6-3, B-1
GROUP, 6-6, 6-7, 10-3, 10-4,
10-8 to 10-10, 11-5, 11-7,
B-3
IF, 1-1, 1-3 to 1-5, 8-1 to 8-4
IF1, 8-1, 8-2, 9-3, 9-4, B-3
IF2, 8-1, 8-3, 9-3, 9-4, B-3
IFB, 8-1, 8-3, B-3
IFDEF, 8-1 to 8-3, B-3
IFDIF, 8-1, 8-3, B-3
IFE, 8-1, 8-2, B-3
IFIDN, 8-1, 8-3, B-3
IFNB, 8-1, 8-4, B-3
IFNDEF, 8-1 to 8-3, B-3

INDEX (continued)

INCLUDE, 5-1, 5-3, 5-5, 5-12,
 7-3, 9-2, 10-1, 10-6, 10-9
 IRP, 3-1, 7-5 to 7-7, 11-3, B-2
 IRPC, 7-6, B-2
 LABEL, 5-6
 .LALL, 7-8, 9-6, B-3
 .LFCOND, 1-5, 9-5, B-3
 .LIST, 1-4, 1-10, 9-4, 9-5,
 B-3
 LOCAL, 4-8, 6-1, 7-3, 7-4,
 11-2, 11-7, B-2
 MACRO, 1-1, 1-2, 3-1, 4-8, 7-1
 to 7-9, 8-2 to 8-4, 9-3,
 9-4, 9-6, 10-1, 10-5,
 10-7, B-2
 NAME, 5-6
 ORG, 5-1, 5-7, B-1
 %OUT, 9-3, 9-4, 11-1, B-3, D-22
 PAGE, 9-1 to 9-3
 PROC, 5-1, 5-7, 5-8, 6-2, 7-4,
 10-4, 10-5, 10-12, 10-13,
 B-1, D-3
 PUBLIC, 3-3, 5-8, 6-1, 6-2,
 6-4 to 6-6, 10-4, 10-5,
 10-10, 10-11, 11-8, B-1
 PURGE, 7-3, B-2
 .RADIX, 5-1, 5-9, B-1
 RECORD, 5-9, B-4, C-1
 REPT, 7-5, 10-4, 11-3, B-2
 .SALL, 9-6, B-3
 SEGMENT, 3-2 to 3-7
 .SFCOND, 1-5, 9-5, B-3
 STRUC, 4-6, 5-1, 5-11 to 5-13,
 11-3, 11-4, 11-6, 11-7, C-2
 SUBTTL, 9-1 to 9-3, B-3
 .TFCOND, 1-5, 9-5, 9-6, B-3
 TITLE, 5-6, 9-1 to 9-3, 12-6,
 12-7, B-3
 .XALL, 7-8, 9-6, B-3
 .XCREF, 9-6, 9-7, B-3
 .XLIST, 9-4, 9-5, B-3
 DUP, 3-1, 4-4, 4-6, 4-7, 5-3,
 5-10 to 5-13, 6-5, 10-4,
 10-5, 11-3, 11-4, 11-7, C-2

E

Editor, 1-3, 1-10
 ELSE, 8-1, 11-2, 11-3, 11-6,
 B-3,
 END, 1-5, 3-2, 3-3, B-2, D-30

ENDIF, 8-1, 9-4, 11-6, B-3
 ENDM, 7-1, 7-4 to 7-7, 7-9,
 10-4, B-2
 ENDP, 5-1, 5-7, 5-8, 7-4, 10-5
 ENDS, 2-1, 3-2, 3-3, B-2
 EQU directive, 10-5, 10-12, 5-1,
 5-4
 Equal sign, 5-4
 Error messages, 1-10,
 11-1 to 11-9, C-1
 Error messages of CREF, 12-5
 exclamation mark, 7-7, 7-8
 Executable, creating a file, 1-10
 EVEN directive, 5-5, 11-3
 EXITM, B-2
 expressions, 3-9
 extensions
 default, 1-2, 1-4 to 1-6, 1-8,
 1-9, 2-4, 3-2, 4-9, 5-1,
 5-2, 5-8, 5-9, 6-4, 7-8,
 9-2, 9-4, 9-6, 10-1,
 12-1 to 12-3, 12-7, D-5
 EXTRN directive, 6-2, 6-3, 10-9
 to 10-12

F

FAR, 3-7, 4-6, 4-8, 4-10, 5-6,
 5-8, 5-9, 6-2, 10-4, 10-5,
 10-12, 10-13, B-1, D-3,
 D-13, D-25
 Flags, 2-5, 2-6, D-1 to D-31
 Fields of records, 4-1
 Format directives, 9-1
 PAGE, 9-1 to 9-3, B-3
 TITLE, 9-1 to 9-3, B-3
 SUBTTL, 9-1 to 9-3, B-3
 Formats of listing and symbol
 tables, 10-1
 Formatting, 12-6

G

General purpose registers, 2-2,
 2-3
 GROUP directive, 4-5, 4-10, 5-2,
 6-6

H

HIGH, 4-11, B-4

INDEX (continued)

I

Identifiers, 3-6
 IF, 1-1, 1-3 to 1-5, 8-1 to 8-4,
 B-3
 IF1, 8-1, 8-2, 9-3, 9-4, B-3
 IF2, 8-1, 8-3, 9-3, 9-4, B-3
 IFB, 8-1, 8-3, B-3
 IFDEF, 8-1 to 8-3, B-3
 IFDIF, 8-1, 8-3, B-3
 IFE, 8-1, 8-2, B-3
 IFIDN, 8-1, 8-3, B-3
 IFNB, 8-1, 8-4, B-3
 IFNDEF, 8-1 to 8-3, B-3
 Immediate operands, 3-10
 INCLUDE, 5-1, 5-3, 5-5, 5-12,
 7-3, 9-2, 10-1, 10-6, 10-9
 Indexed memory operands, 3-10,
 3-11
 Indirect addressing, 2-3, 2-4
 Index registers, 2-2 to 2-4,
 3-9, 3-10
 Instruction set, D-1
 Interrupt flag, D-4, D-28
 I/O handler errors, 11-1, 11-8
 Invoking a macro, 7-2
 Invoking the assembler, 1-5, 3-2
 method one, 1-4, 1-8, 1-9,
 3-2
 method two, 1-4, 1-5, 1-7,
 1-8
 the "Other" option, 1-6
 IRP, 3-1, 7-5 to 7-7, 11-3, B-2
 IRPC, 7-6, B-2

L

LABEL, 5-6, B-1
 Labels, 1-2, 3-4 to 3-7, 3-9,
 3-10, 4-8 to 4-10, 5-1,
 5-8, 6-2, 6-4, 7-3, 11-9
 .LALL, 7-8, 9-6, B-3
 Legal characters, 3-1
 LENGTH, 4-6, B-4,
 .LFCOND, 1-5, 9-5, B-3
 .LIST, 1-4, 1-10, 9-4, 9-5, B-3
 Listing File, 1-2, 1-4 to 1-6,
 7-3, 7-4, 9-1, 9-3 to 9-5,
 10-1, 10-2, 10-4, 11-1,
 11-8

Listing file of CREF,
 12-5 to 12-7
 LOCAL, 4-8, 6-1, 7-3, 7-4,
 11-2, 11-7, B-2
 Logical operators, 4-1, 4-3, 4-4
 LOW, 4-11, B-4

M

Macro, 1-1, 1-2, 3-1, 4-8,
 7-1 to 7-9, 8-2 to 8-4,
 9-3, 9-4, 9-6, 10-1, 10-5,
 10-7, B-2
 creating a macro, 7-1
 deleting a macro, 7-3
 invoking a macro, 7-2, 7-2
 operators, 7-7 to 7-9, B-2
 MASK, 3-9, 4-11, 4-13, 4-14,
 5-9, 5-11, 10-6 to 10-9,
 B-4, D-19, D-31
 Memory combine type, 6-4, 6-10,
 6-11
 Memory operands, 3-10, 3-11
 Memory organization, 3-5, 3-9
 Menu, creating an assembly
 language selection, 1-7
 Messages, 1-10, 11-1, 11-8, 11-9,
 12-5, C-1
 assembler, 12-1 to 12-8, B-1,
 I/O handler, 11-1, 11-8
 runtime, 3-3, 11-1, 11-9
 Method one for invoking the
 assembly, 1-4
 Method two for invoking the
 assembly, 1-5
 Module, 1-8, 3-2, 3-4, 4-2, 5-6,
 6-1 to 6-3, 6-5, 7-1, 9-2,
 B-2

N

NAME, 5-8, B-1 to B-3
 NONE combine type, 6-4, 10-10
 Numeric notation, 3-7

O

Object module, 1-4

INDEX (continued)

offset, 1-2, 2-1 to 2-4, 3-2,
3-6, 3-7, 3-10, 4-4, 4-5,
4-8, 4-9, 4-15, 5-6 to 5-8,
5-13, 6-1, 6-3, 6-6, 6-7,
7-3, 10-1 to 10-3, 10-6,
10-8, 10-11, 10-13, B-4,
D-3, D-16, D-17

OFFSET, 4-5

Offset attribute, 3-7

Operands, 3-1, 3-6, 3-9 to 3-11,
4-1 to 4-4, 4-8, 5-4,
5-11, 5-13, 11-2, 11-6,
C-2, D-1 to D-31

Operators,
arithmetic, 2-3, 4-1, 11-9,
D-26, D-27
attributal, 4-1, 4-4
HIGH, 4-11
LENGTH, 4-6, B-4
logical, 4-3, 4-4, B-5
LOW, 4-11, B-4
MASK, 3-9, 4-11, 4-13,
10-6 to 10-9, B-4
OFFSET, 4-5, B-4
PTR, 4-8, 4-9, B-3, B-4
precedence, 3-1, 4-14, B-4
record specific, 4-11, 5-9, B-4
relational, 4-1, 4-2
SEG, 3-4, 4-4, 4-5, 4-15, 5-2,
6-3, 6-5 to 6-7, 11-6,
11-7, B-4, C-2, D-13,
D-16, D-19, D-22, D-23
segment override, 4-8 to 4-10,
B-3
SHORT, 4-10, B-3, B-5
Shift Count, 4-11, 4-12
SIZE, 4-7, B-4
synthetic, 4-8
THIS, 4-9, B-3, B-4
TYPE, 4-6, B-4
value returning, 4-4, 4-12,
4-13, B-4
WIDTH, 4-12, B-4

ORG directive, 5-7

"Other" option for invoking the
assembler, 1-6

%OUT, 9-3, 9-4, 11-1, B-3, D-22

Overflow flag, 2-5

Override operators, 4-8, B-3

P

Page, 3-3, 6-4, 10-10,
12-6 to 12-8

PAGE directive, 9-1, 9-2, B-3

PARA, 10-6, 10-10, 3-3, 6-4

Parameters, 3-1, 5-8, 7-1 to 7-3,
7-5 to 7-8, 8-3, 11-6,
11-7, B-2, C-1, D-25

Parity flag, 2-5

Percent sign, 7-7, 7-8, B-2

Pointer register, 2-2 to 2-4

Port addressing, 2-5

Precedence of operators, 4-14,
B-4

PROC, 5-1, 5-7, 5-8, 6-2, 7-4,
10-4, 10-5, 10-12, 10-13,
B-1

Procedures, 5-7, 5-8, 6-2, 10-13,
11-3

PTR, 4-8, 4-9, 4-15, 5-3, 10-4,
10-5, 11-2, B-3, B-4, D-25

PUBLIC, 3-3, 5-8, 6-1, 6-2,
6-4 to 6-6, 10-4, 10-5,
10-10, 10-11, 11-8, B-1

Public combine type, 6-5, 10-10,
10-11

PURGE, 7-3, B-2

R

.RADIX, 5-1, 5-9, B-1

RECORD, 5-9, B-4, C-1

Record specific operators,
4-11 to 4-13, 5-9, B-4

Relational operators, 4-1, 4-2

Register offset addressing, 2-4

Register operands, 3-10

Registers, 2-1 to 2-5, 3-9, 3-10,
4-9, 4-10, 5-4, 5-8, 10-3,
11-1 to 11-3, D-10,
D-15 to D-17, D-20, D-22,
D-23, D-26
base, 2-1 to 2-4, 3-2, 3-4,
3-6, 3-9, 3-10, 4-4, 4-5,
4-9, 5-4, 5-8, 5-9, 5-11,
6-1, 6-3, 6-4, 6-6, 6-7,
10-9, 10-11, 11-2, 11-5,
C-2, D-19

INDEX (continued)

flag, 2-5, 2-6, D-2 to D-4,
D-10, D-15, D-17, D-18,
D-21 to D-28
general, 2-2, 2-3, 3-1, D-16,
D-17
index, 2-2 to 2-4, 3-9, 3-10,
5-4, 11-3 to 11-5, C-2,
D-16, D-17, D-28, D-31
pointer, 2-2 to 2-4
segment, 2-1 to 2-4
Repeat blocks, 7-1, 7-4
REPT, 7-5, 10-4, 11-3, B-2
Runtime error messages, 11-9

S

.SALL, 9-6, B-3
SEG, 3-4, 4-4, 4-5, 4-15, 5-2,
6-3, 6-5 to 6-7, 11-6,
11-7, B-4, C-2, D-13, D-16,
D-19, D-22, D-23
SEGMENT, 2-1 to 2-4, B-2
Segment, 2-1 to 2-4, 3-1 to 3-7,
3-9, 4-2, 4-4, 4-5,
4-8 to 4-10, 4-14, 5-1,
5-2, 5-4 to 5-8,
6-3 to 6-7, 7-4,
10-2 to 10-6, 10-10,
10-11, 10-13, 11-1 to 11-7,
B-1 to B-4, C-1, C-2, D-3,
D-13, D-16, D-17, D-19,
D-24, D-25, D-29
attributes of, 10-2 to 10-6
base values, 2-1
registers, 2-1 to 2-5
Segment combining,
align type, 3-3, 6-3, 6-4, 10-1
class name, 3-3, 6-3, 6-6,
10-11
combine type, 3-3, 6-3 to 6-5,
10-1, C-1
Segment override operator, (:)
3-6, 4-5, 4-9, 4-10, 4-14,
B-4
Semicolon, use of in macros, 7-8
.SFCOND, 1-5, 9-5, B-3
Shift count, 4-11, 4-12
SHL, 4-2, 4-15, 10-4, B-4, D-26
SHORT, 4-8, 4-10, 4-15, B-3, B-5,
D-11 to D-15, D-18
SHR, 4-2, 4-15, B-4, D-28

Sign flag, 2-5
SIZE, 4-7, B-4
Special macro operators,
7-7 to 7-8, B-2
Stack combine type, 6-5, 10-11
Starting,
assembler, 1-1 to 1-6,
1-8 to 1-10
CREF, 9-6, 10-1, 12-1 to 12-6,
12-8, B-3
invoking method one, 12-2
invoking method two, 12-2
Statement line format, 3-5
action, 3-5
comments, 3-6, 3-8, 7-8, 9-6,
11-3
directives, 1-1, 1-5, 3-1, 3-5,
3-9, 4-2, 4-11, 5-1, 5-7,
5-12, 6-1, 6-3, 8-1 to 8-3,
9-1, 9-3 to 9-6, 10-11,
10-12, 11-3, B-1 to B-3
expressions, 3-1, 3-6, 3-9,
4-5, 4-6, 4-14, 5-4, 5-11,
8-2, 9-4, 9-5, 10-12
instructions, 1-1, 1-7, 2-6,
3-1, 3-3, 3-5, 3-6, 3-9,
4-2, 4-8, 4-10, 5-2, 5-7,
5-8, 7-1, 10-2, 11-5, D-7,
D-19, D-22, D-23
String, 3-8
STRUCTURE, 2-2, 3-1, 3-6, 3-10,
3-11, 4-14, 5-4, 5-6,
5-11 to 5-13, 10-6 to 10-8,
11-5, B-4, C-2
Structure operands, 3-10, 3-11,
5-13
SUBTTL, 9-1 to 9-3, B-3
Switches, 1-5, 9-3, 9-5
Symbol table, 1-2, 10-6 to 10-9,
10-11 to 10-13, 11-8,
11-9, 7-2
Symbols, 1-1, 1-2, 3-4, 3-6, 3-9,
5-4, 6-1, 6-3, 7-6 to 7-8,
10-2 to 10-6, 10-11, 10-12,
12-1, 12-4, 12-7, B-1
Synthetic operators, 4-8

T

Text suppression directives,
9-4 to 9-7

INDEX (continued)

.TFCOND, 1-5, 9-5, 9-6, B-3
THIS, 4-9, B-3, B-4
TITLE, 9-1 to 9-3, B-3
TYPE, 4-6, B-3
.TYPE, 4-7, B-3 to B-5
Type, 3-6
Type attribute, 3-7, 4-9
Type operator, 4-6, 4-7

V

Value returning operators, 4-4,
B-4
Variables, 1-2, 2-2, 3-4,
3-6 to 3-10, 4-1, 4-2,
4-5, 4-8, 4-9, 5-9, 5-12,
6-4, 6-5, 7-3, 9-7, 10-4

W

WIDTH, 4-11 to 4-14, B-4
Width shift, 10-8
Word align type, 6-4, 10-10

X

/X switch, 9-5
.XALL, 7-8, 9-6, B-3
.XCREF, 9-6, 9-7, B-3
.XLIST, 9-4, 9-5, B-3

Z

Zero flag, 2-5

Help Us Help You . . .

We've worked hard to make this document useful, readable, and technically accurate. Did we succeed? Only you can tell us! Your comments and suggestions will help us improve our technical communications. Please take a few minutes to let us know how you feel.

How did you receive this publication?

- | | |
|--|--------------------------------------|
| ☐ Support or Sales Rep | ☐ Don't know |
| ☐ Wang Supplies Division | ☐ Other _____ |
| ☐ From another user | _____ |
| ☐ Enclosed with equipment | _____ |

How did you use this Publication?

- | | |
|--|--|
| ☐ Introduction to the subject | ☐ Aid to advanced knowledge |
| ☐ Classroom text (student) | ☐ Guide to operating instructions |
| ☐ Classroom text (teacher) | ☐ As a reference manual |
| ☐ Self-study text | ☐ Other _____ |

Please rate the quality of this publication in each of the following areas.

	EXCELLENT	GOOD	FAIR	POOR	VERY POOR
Technical Accuracy — Does the system work the way the manual says it does?	☐	☐	☐	☐	☐
Readability — Is the manual easy to read and understand?	☐	☐	☐	☐	☐
Clarity — Are the instructions easy to follow?	☐	☐	☐	☐	☐
Examples — Were they helpful, realistic? Were there enough of them?	☐	☐	☐	☐	☐
Organization — Was it logical? Was it easy to find what you needed to know?	☐	☐	☐	☐	☐
Illustrations — Were they clear and useful?	☐	☐	☐	☐	☐
Physical Attractiveness — What did you think of the printing, binding, etc?	☐	☐	☐	☐	☐

Were there any terms or concepts that were not defined properly? ☐ Y ☐ N If so, what were they? _____

After reading this document do you feel that you will be able to operate the equipment/software? ☐ Yes ☐ No
☐ Yes, with practice

What errors or faults did you find in the manual? (Please include page numbers) _____

Do you have any other comments or suggestions? _____

Name _____ Street _____

Title _____ City _____

Dept/Mail Stop _____ State/Country _____

Company _____ Zip Code _____ Telephone _____

Thank you for your help.

WANG

Fold



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS

PERMIT NO. 16

LOWELL, MA

POSTAGE WILL BE PAID BY ADDRESSEE

**WANG LABORATORIES, INC.
CHARLES T. PEERS, JR., MAIL STOP 1226A
ONE INDUSTRIAL AVENUE
LOWELL, MASSACHUSETTS 01851**



Fold

Cut along dotted line

WANG

The completed order form should be mailed to:

WANG LABORATORIES, INC.
Supplies Division
51 Middlesex St.
No. Chelmsford MA 01863

To Order by Phone, Call:

(800)225-0234

From Mass., Hawaii, and Alaska

(617)256-1400**TELEX 951-743**

Order Form for Wang Manuals and Documentation

① Customer Number (If Known)				
② Bill To:			Ship To:	
③ Customer Contact:			④ Date	
() ()			Purchase Order Number	
Phone			Name	
⑤ Taxable ⑥ Tax Exempt Number ⑦ Credit This Order to				
Yes ☐ A Wang Salesperson				
No ☐ Please Complete Salesperson's Name Employee No. RDB No.				
⑧ Document Number	Description	Quantity	⑨ Unit Price	Total Price
⑩ Authorized Signature _____ Date _____ ☐ Check this box if you would like a free copy of the Corporate Publications Literature Catalog (700-5294)			Sub Total	
			Less Any Applicable Discount	
			Sub Total	
			Local/State Tax	
			Total Amount	

Ordering Instructions

1. If you have purchased supplies from Wang before, and know your Customer Number, please write it here.
2. Provide appropriate Billing Address and Shipping Address.
3. Please provide a phone number and name, should it be necessary for WANG to contact you about your order.
4. Your purchase order number and date.
5. Show whether order is taxable or not.
6. If tax exempt, please provide your exemption number.
7. If you wish credit for this order to be given to a WANG salesperson, please complete.
8. Show part numbers, description and quantity for each product ordered.
9. Pricing extensions and totaling can be completed at your option. Wang will refigure these prices and add freight on your invoice.
10. Signature of authorized buyer and date.

Wang Supplies Division Terms and Conditions

1. **TAXES** — Prices are exclusive of all sales, use, and like taxes.
2. **DELIVERY** — Delivery will be F.O.B. Wang's plant. Customer will be billed for freight charges; and unless customer specifies otherwise, all shipments will go best way surface as determined by Wang. Wang shall not assume any liability in connection with the shipment nor shall the carrier be construed to be an agent of Wang. If the customer requests that Wang arrange for insurance the customer will be billed for the insurance charges.
3. **PAYMENT** — Terms are net 30 days from date of invoice. Unless otherwise stated by customer, partial shipments will generate partial invoices.
4. **PRICES** — The prices shown are subject to change without notice. Individual document prices may be found in the Corporate Publications Literature Catalog (700-5294).
5. **LIMITATION OF LIABILITY** — In no event shall Wang be liable for loss of data or for special, incidental or consequential damages in connection with or arising out of the use of or information contained in any manuals or documentation furnished hereunder.

WANG

Fold



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO. 16 NO. CHELMSFORD, MA.

POSTAGE WILL BE PAID BY ADDRESSEE

WANG LABORATORIES, INC.
Supplies Division
c/o Order Entry Dept.
M/S 5511
51 Middlesex St.
No. Chelmsford, MA 01863



Fold

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

WANG LABORATORIES, INC., ONE INDUSTRIAL AVENUE, LOWELL, MA 01851 • TEL: 617/454-5000, TWX 710-343-6769, TELEX 94-7421

WANG

Printed in U.S.A.
700-8315
7-83-7.5M