Microsoft® Macro Assembler 5.1

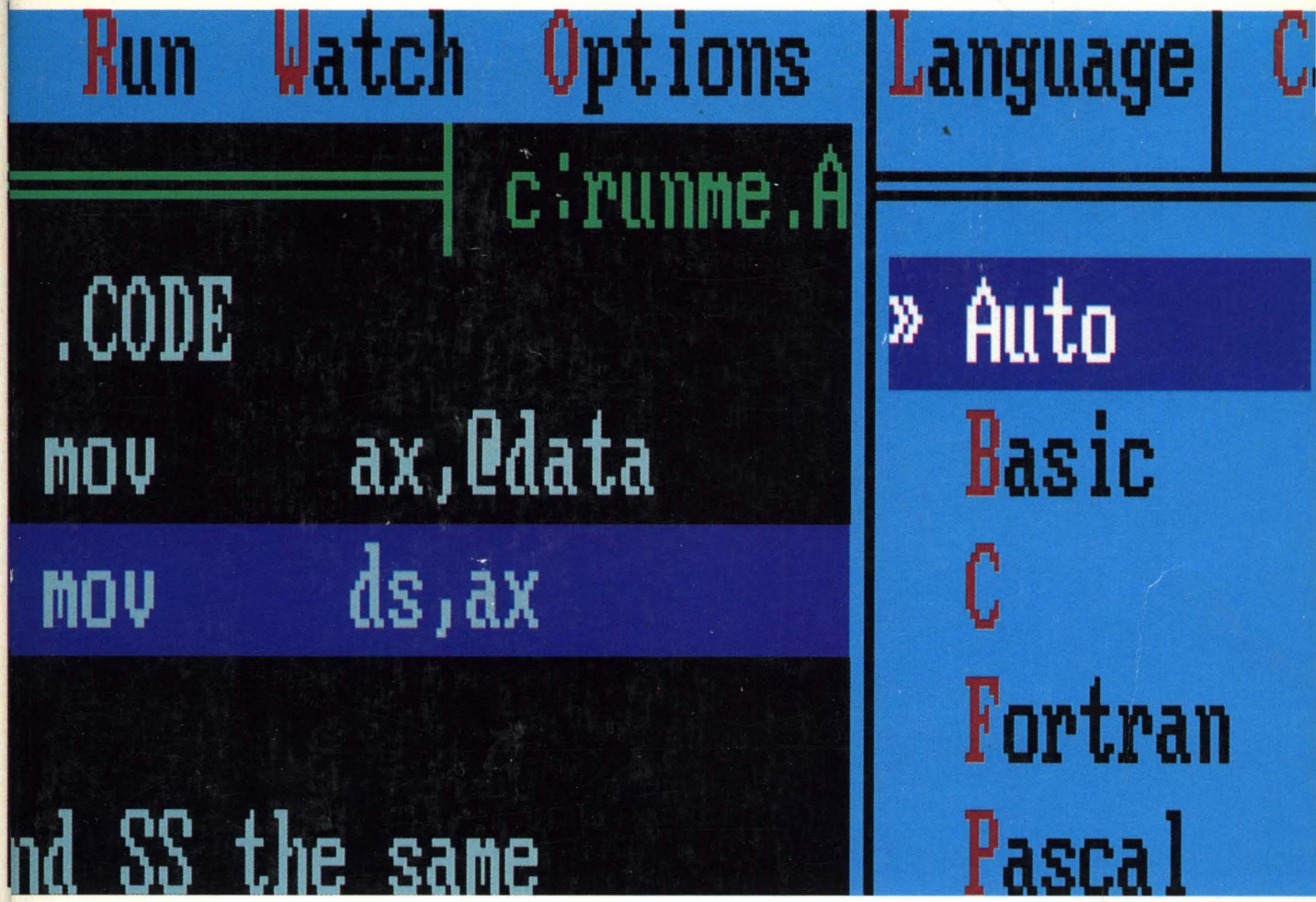
Updates and Microsoft Editor

5.1 Update

Microsoft® CodeView® and Utilities Update

Microsoft Editor

Microsoft® Macro Assembler 5.1



Microsoft®

Microsoft® Macro Assembler

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

Version 5.1 Update

Microsoft CodeView® and Utilities Update

Microsoft Editor

Microsoft Corporation

10 9 8 7

Contents

Microsoft Macro Assembler Version 5.1 Update

Microsoft CodeView and Utilities Update

Microsoft Editor User's Guide

Microsoft® Macro Assembler

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

Version 5.1 Update

Microsoft Corporation

Contents

Tables

Section 1
Overview

Microsoft® Macro Assembler Version 5.1 provides a complete and powerful set of tools for developing fast and efficient assembly programs for the MS-DOS® operating system and Microsoft Operating System/2 (MS® OS/2). With the new high-level-language support features you can easily add assembly speed and power to BASIC, Pascal, C, and FORTRAN programs.

This update describes new features of the language, in particular its operation within OS/2. The pages that follow use the term "OS/2" systems—MS OS/2 and IBM® OS/2. Similarly, the term "DOS" refers to both MS-DOS and PC-DOS operating systems. The name of the specific operating system is used when it is necessary to note features that are unique to that system.

This document describes the new features of the Version 5.1 Macro Assembler. Section 1.2 below describes the new features in greater detail. Briefly, the new features include the following:

- **Faster assembly.** Version 5.1 makes better use of far memory to hold as much of your source code as possible, so your programs assemble faster.
- **OS/2 support.** Version 5.1 runs under both OS/2 and DOS operating systems, enabling you to develop powerful applications for OS/2.
- **Text-macro and directive extensions.** New directives, text-macro directives, and predefined text macros let you write more sophisticated macros for powerful and portable code.
- **High-level-language support.** With Version 5.1, writing assembly routines for use with high-level languages becomes almost automatic. Features of the **MIXED.INC** macro file shipped with Version 5.0 are now built into the macro assembler, along with additional supporting directives and extensions.

Note

When you assemble using the **/ML** switch, you must now follow the conventions used in Section 3.1.4, "Predefined Text Macros," in order for **MASM** to recognize a predefined text macro. That is, each word in the name of the text macro is initial capped.

1.1 Using This Update

This first section of the update provides an overview of the update and the new features. Section 1.2 summarizes the new features; Section 1.3 describes the distribution disks; and Section 1.4 describes how to use the setup batch file.

The remaining sections in the update describe the new features and demonstrate how to use them in your programs:

- Section 2, "OS/2 Systems Support," shows you how to prepare a simple OS/2 application.
- Section 3, "Macros, Conditional Assembly, and Local Labels," describes the new text-macro features, the new predefined text macros, the **ELSEIF** directives, and the automatic generation of local labels for jump instructions.
- Section 4, "Directives," tells you how to use the extensions to the **.TYPE** operator and to the **COMM** directive, and describes a change in the **.CODE** directive. Section 4 also explains the new way to declare variables as pointers so that you can use the CodeView debugger to examine the variables.
- Section 5, "High-Level-Language Support," describes the new high-level--language calling features that replace the macros supplied with Version 5.0 in the **MIXED.INC** file.
- Section 6, "Other Features," describes how to use the new line-continuation and listing features.

Note

Information in the update supersedes that in the Version 5.0 manuals and files. Be sure to check your Version 5.1 Macro Assembler disks for a **README.DOC** file. This file contains information unavailable when this update was printed.

1.2 Summary of New Features

The Microsoft Macro Assembler Version 5.1 surpasses and extends the power and ease-of-use of Version 5.0 by providing the following new features:

- **OS/2 Systems Support**

  Version 5.1 runs under OS/2, as well as DOS, so you can use it to develop new, powerful OS/2 software. Version 5.1 includes the necessary supporting libraries to write full OS/2 programs. A bound version and a real-mode version are included in Version 5.1.

- **Macros, Conditional Assembly, and Local Labels**

  Extensions to text macros let you write more flexible and powerful macros for your programs. Version 5.1 includes new built-in text macros as well as text macro directives that let you directly manipulate string equates. An extension to the expression operator (%) allows you to evaluate text macros anywhere in your source.

  The **ELSEIF** directive simplifies the writing of complex conditional assembly sections in your macros and programs.

  Version 5.1 of the Macro Assembler allows automatic generation of local labels for jump instructions. Automatic local labels free you from having to write labels for small code sections.

- **Directives**

  Extensions to the **.TYPE** directive provide additional information about an operand. The **COMM** directive now accepts items of any size, rather than being limited to 1-, 2-, 4-, 6-, 8-, or 10-byte items.

  A new type of data declaration makes it possible to generate Microsoft CodeView® information for pointer variables.

  A change in the register assumptions used in the **.CODE** directive makes it easier to create multiple code segments.

- **High-Level-Language Support**

  Version 5.1 contains several features that simplify the writing of assembly routines to be called from high-level languages. The high-level-language support features incorporate and extend the features of the mixed-language programming macros distributed with Version 5.0. The new high-level-language features include the following:

  - A new argument on the **.MODEL** directive sets naming conventions, parameter order, and return conventions.
  - Extensions to the **PROC** directive automatically save specified registers and generate text macros for arguments passed on the stack.
  - A new **LOCAL** directive allocates stack space for local variables and generates text equates for the variables.
  - Labels can be made local to a procedure, allowing better isolation of the procedures.

- **Line Continuation**

  Version 5.1 lets you use a continuation character to continue program lines. By continuing lines you can combine physical lines into logical lines up to 512 characters long.

- **Performance**

  Version 5.1 is significantly faster than Version 5.0 because it uses dynamically allocated memory to perform file caching. File caching stores significant parts of the source and include files to be stored in memory.

  In addition, Version 5.1 allows significantly more structure definitions because it now uses far memory to store these definitions.

1.3 Disk Contents

Be sure to make backup copies of the assembler disks before using any of the programs in the package. Keep the copies in a safe place so you can use them to restore the originals if they are damaged or lost.

All files on the disks are listed in the **PACKING.LST** file on Disk 1.

The files on the disk are not copy protected. You may make one backup copy for your own use. You may not distribute any executable, object, or library files on the disk. The sample programs are in the public domain.

No license is required to distribute executable files created with the assembler.

1.4 Setup

Your distribution disks contain two versions of **MASM**: a version that runs only under DOS, and a bound version. (A "bound" program is an OS/2 program that runs under DOS 3.0 or higher as well as under OS/2. See the **PACKING.LST** file for the location of the different versions.) The bound version runs slightly slower under DOS than the DOS version and uses approximately 15K more memory.

To set up **MASM** on your hard disk, run the **SETUP** program from your distribution disks. The **SETUP** program prints directions for its use on the screen and prompts you for information.

Note

To set up **MASM** for OS/2, you must run the **SETUP** program in real mode. The **SETUP** program does not run under protected-mode OS/2.

If you are going to develop OS/2 applications, you may want to set up separate directories—a **\BINB** directory for the bound versions of **MASM** and the utilities, and a **\BINP** directory for protected-mode programs. This allows you to keep different versions of the same program separated. You may also want to place the **\BINB** and **\BINP** directories later in your path so that the faster DOS version of **MASM** is used whenever appropriate.

Section 2
OS/2 Systems Support

This section gives you a taste of writing **MASM** applications for OS/2 by showing you how to prepare a simple OS/2 application to run under the OS/2 or the DOS operating system. In addition, the section also briefly discusses some differences between OS/2 and DOS system calls.

You will not learn how to write full-fledged OS/2 applications from this section. To find out more about OS/2 programming, consult the references in Section 2.5.

The first part of this section shows and explains a short OS/2 program. Section 2.2 discusses linking OS/2 programs and binding applications so that they run under either the OS/2 or the DOS operating system. The next section, Section 2.3, tells you about the differences between the OS/2 and DOS system calls. Section 2.4, "Register and Memory Conventions," tells you what various registers contain when OS/2 starts your program. The final section contains references to books and articles that will help you learn about writing OS/2 programs.

2.1 A Sample Program

The following program is an OS/2 version of the sample programs on pages 14-16 of the *Microsoft Macro Assembler Programmer's Guide*:

```
TITLE   hellos2
          INCLUDELIB doscalls.lib     ; Include DOSCALLS.LIB so
                                      ; you don't need to
                                      ; specify on linking
          .286
          .MODEL  small

          ; Define a push macro so the bound version can use
          ; 8088/8086 instructions when necessary
pushc     MACRO   pushed, pushed2 ;; Define bind from command line
          IFDEF   bind                ;;   if you want to bind the
program
          mov     ax, pushed pushed2 ;; Push constant for
8088/8086
          push    ax
          ELSE
```

```
push    pushed pushed2      ;; Push constant for 80186+
          ENDIF
          ENDM

          .STACK   ; Use default 1K stack for OS/2 calls
          .DATA
message   DB       "Hello, world.",13,10
lmessage  EQU      $ - message
bytesout  DD       ?

          .CODE
          EXTRN    DosWrite:FAR, DosExit:FAR  ; Declare OS/2 calls
start:                                 ; Notice that DS is
automatically set

; DosWrite function used to write to screen

          pushc    1                   ; Push 1 as handle for
standard output
          push     ds                  ; Push far address of
          pushc    OFFSET message      ;   "message"
          pushc    lmessage            ; Push length of "message"
          push     ds                  ; Push far address of
          pushc    OFFSET bytesout     ;   "bytesout"

          call     DosWrite            ; Make API call
                                       ; AX contains error code
                                       ; Variable "bytesout"
contains
                                       ;   number of bytes written

; DosExit function used to return to DOS

          pushc    1                   ; Push action 1 to end all
threads
          pushc    0                   ; Push return code 0
          call     DosExit             ; Exit

          END      start
```

If you are assembling and linking under DOS, you may use either version of **MASM** from your distribution disks. To assemble under protected-mode OS/2, you must use the bound version of **MASM**.

To assemble the program, type the following command:

```
MASM hellos2;
```

Linking the program and binding it—making the program able to run under either DOS or OS/2—is discussed in the next section.

If you want to assemble the program so you can bind it to run under either OS/2 or DOS, use this command line:

```
MASM /Dbind hellos2;
```

The `/D` option defines the constant `bind`. The `pushc` macro in the sample program checks for this constant to tell whether or not to use the 80186-80286/80386-specific-push constant instruction. Macros provide a convenient way to write a single program for different processors.

2.2 Linking and Binding OS/2 MASM Programs

To link the sample program, use the following command:

```
LINK hellos2;
```

If you omit the **INCLUDELIB** directive from the program, you must include the **DOSCALLS.LIB** file in the link command. In this case, the command line would appear as follows:

```
LINK hellos2,,DOSCALLS.LIB;
```

The **DOSCALLS.LIB** file must be in the current directory or in the path described by the **LIB** environment variable. Using the **INCLUDELIB** directive ensures that the **DOSCALLS.LIB** file is always included with the program.

The **DOSCALLS.LIB** file is necessary because OS/2 applications use dynamic linking which does not resolve procedure calls until the program is loaded into memory. When the program is loaded, the code required by a call is extracted from a dynamic-link library and loaded with the program. All OS/2 system calls are contained in dynamic-link libraries.

The **DOSCALLS.LIB** file does not contain any code for OS/2 system calls. Instead, it contains dynamic-link reference records. The linker uses these records to make the connection between the OS/2 system call in the application and the dynamic-link library containing the system procedure.

The **LINK** command includes a field for an OS/2 definition file, as described in the *Microsoft CodeView and Utilities Update*. For the sample program, this field is not necessary.

You can write OS/2 programs that run under either OS/2 or DOS 3.0 or higher by restricting the OS/2 system calls your program uses. OS/2 function calls are known collectively as the Applications Program Interface (API). If you restrict your program to a subset of these functions, known as the Family API, you can write programs that run under both OS/2 and DOS 3.0 or higher. See the *Microsoft Operating System/2 Programmer's Reference* for a list of the Family API functions.

Once you have written, assembled, and linked a program using only Family API functions, you may bind the program by using the **BIND** utility. Binding your program produces a version that runs under either operating system. The process of binding resolves references to dynamic-link routines so that the application runs without the dynamic-link libraries.

Because the sample program uses only Family API calls, you can bind it with the following command:

```
BIND hellos2 path1DOSCALLS.LIB path2API.LIB
```

The *path1* argument is the path to the **DOSCALLS.LIB** file; *path2* is the path to the **API.LIB** file.

2.3 Using OS/2 System Calls

DOS implements system calls by placing values in registers and using a software interrupt. Although most DOS functions have OS/2 equivalents, the OS/2 mechanism is different.

To call an OS/2 function, you first push all of the arguments onto the stack, and then invoke the function with a far call. The fragment below shows a call to the **DOSWRITE** function, the equivalent of DOS function 40h:

```
EXTRN   DosWrite:FAR,DosExit:FAR ; DOS functions FAR external
    .
    .
    .

; DosWrite function used to write to screen
; This code is only for the 186, 286, and 386 processors
            push    1                   ; Push 1 as handle for
standard output
            push    ds                  ; Push far address of
            push    OFFSET message      ;   "message"
            push    lmessage            ; Push length of "message"
            push    ds                  ; Push far address of
            push    OFFSET bytesout     ;   "bytesout"

            call    DosWrite            ; Make API call
                                        ; AX contains error code
                                        ; Variable "bytesout"
contains
                                        ;   number of bytes written
```

The **EXTRN** directive declares **DOSWRITE** a far label. All OS/2 functions are invoked by far calls and must be declared as far labels. Notice that before the program

calls the function, a series of push instructions place the required arguments on the stack. These include arguments containing values used by the function as well as locations for returned values.

Notice also that after the call, there are no corresponding pop instructions. In OS/2, the called procedure removes all arguments from the stack. All OS/2 functions use the stack for arguments. Also, as the comments note, the **AX** register is used to report errors. If the register is nonzero, it contains an error code. If the register is zero, there is no error.

Note

Because OS/2 uses the stack to pass arguments, you may need to allocate a larger stack segment than you would for a DOS program.

Although the function name is given in mixed case, the routine names in the OS/2 libraries are all uppercase. You may use mixed case because **MASM** converts all names to uppercase. If you are writing routines for use with a case-sensitive language such as C, you should use uppercase letters for the OS/2 function names and use the **/MX** or **/ML** assembler option.

See the *Microsoft Operating System/2 Programmer's Reference* for a complete list of OS/2 functions and their arguments.

2.4 Register and Memory Conventions

OS/2 initializes registers and allocates memory differently than DOS. These differences stem from the fact that OS/2 programs do not have a program segment prefix (PSP) and they allocate memory only for the data and code required by the program.

On program startup, the **AX** register points to the segment value of the start of the program's environment. Under DOS, the start of the environment was pointed to by the word at 2Ch in the PSP.

OS/2 also places the program's command-line arguments in the environment. The starting offset of the arguments is placed in the **BX** register. These arguments include the program's name, so that **AX:BX** points to the program name. The program name is followed by a null (zero) byte and the command-line arguments exactly as they were typed. A second null marks the end of the arguments. Under DOS, the PSP contained the program's arguments: byte 80h contained the length of the arguments, which began at byte 81h in the PSP.

Under OS/2, the data segment register, **DS**, contains the segment of the automatic data segment so that you do not have to initialize the register yourself. The **CX** register contains the length of the automatic data segment. The segment is named **DGROUP** and contains both data and the stack. If you use simplified segment directives, this is the **.DATA** segment. You must place one data segment in a group called **DGROUP** if you do not use the simplified directives:

```
_DATA     SEGMENT WORD PUBLIC 'DATA'
          .
          .
          .
_DATA     ENDS

DGROUP    GROUP _DATA
          ASSUME DS:DGROUP
```

Calling the group anything other than **DGROUP**, or not having a **DGROUP** causes an error. In contrast to OS/2, the DOS operating system places the start of the program segment prefix in register **DS** when the program starts up.

Only the memory required by the program is allocated by OS/2. When the program starts, the **DS** and **SS** registers both point to the automatic data segment, which is used for both data and the stack. The **CS** register and the instruction pointer point to the beginning of the code segment. DOS allocates all memory to a program on startup. If a DOS program needs to allocate dynamic memory, it must first adjust the allocated memory to the actual memory needed by using a DOS function call.

The following table summarizes register values when an OS/2 program starts:

Table 2.1
Register Values at Program Start

Register	Contents at Program Start
AX	Segment of program's environment
BX	Offset of command-line arguments within the environment
CX	Length of automatic data segment
SP	Offset of the top of the stack within the automatic data segment
CS	Program's entry point
DS	Segment of the automatic data segment
SS	Segment of the automatic data segment

For assembly-language programs, the values in **CX** and **SP** are the same at program startup. The values may be different in a high-level-language program.

Do not do arithmetic on segment registers under OS/2. Under OS/2, segment registers do not contain actual memory addresses—they contain segment selectors managed by OS/2. Segment arithmetic causes a protection violation which terminates your program.

2.5 For Further Reading

The following books and articles may help you learn to write OS/2 applications in assembly language:

Duncan, Ray. *Advanced OS/2*. Redmond, Wash.: Microsoft Press, in press.

Duncan, Ray. "A Programmer's Introduction to OS/2," *Byte*, September 1987.

Duncan, Ray. "Porting MS-DOS Assembly Language Programs to the OS/2 Environment," *Microsoft Systems Journal*, July 1987, pp. 9-17. (*Microsoft Systems Journal* is a continuing source of information about programming in OS/2.)

Microsoft. *Microsoft Operating System/2 Programmer's Guide*. Redmond, Wash.: 1988.

Microsoft. *Microsoft Operating System/2 Programmer's Reference*. Redmond, Wash.: 1988.

You may also want to see "Environments," a column in *PC World* by Charles Petzold. Beginning the September 29, 1987, issue of *PC World*, Petzold provides a series of tutorials on OS/2 programming.

With the exception of its own publications, Microsoft Corporation does not endorse these books and articles over others on the same subject.

Section 3

Macros, Conditional Assembly, and Local Labels

This section tells you how to use the **MASM** Version 5.1 extensions to text macros and conditional assembly directives, and how to use local labels to simplify writing code using jump instructions.

- Section 3.1 tells you how to use the new text-macro features—now you can use text macros as one-line macros and perform operations on strings.
- Section 3.2 describes how to use the **ELSEIF** directives to write simpler, more readable conditional-assembly blocks.
- Section 3.3 shows you how to use local labels to indicate the targets of jump instructions without using a specific name.

3.1 Text Macros

This section describes the Version 5.1 extensions to text macros. Sections 3.1.1 and 3.1.2 tell you how Version 5.1 changes the evaluation of text macros to make them more powerful and flexible. Section 3.1.3 describes the text-macro string directives. Finally, Section 3.1.4 describes three new predefined text macros—**@Cpu**, **@WordSize**, and **@Version**—that you can use to control assembly.

3.1.1 Text-Macro Evaluation

In Version 5.1, text macros now follow the same evaluation rules as regular macros. This change means that text is substituted for the text-macro name when it appears in the operation field. You can now use text macros as short, one-line macros. For example, the following lines define and use a text macro, `NotOp`:

```
NotOp   EQU  <NEG>
        .
        .
        .
        NotOp    AX
```

Previous versions of MASM did not evaluate text macros appearing in the operation field.

When assembling the lines, MASM substitutes `NEG` for `NotOp` in the final line.

Note

MASM Version 5.1 displays the values of text macros in the program listing when you assign the text macro a value. Text-macro values appear in the left column preceded by an equal sign (=).

3.1.2 Using the Expression Operator with Text Macros

You can use the expression operator (%) to substitute the values of text macros for the macro names anywhere a text-macro name appears. When the expression operator is the first thing on a line and is followed by one or more blanks or tabs, the line is scanned for text macros and the values of the macros are substituted. Using the expression operator, you can force substitution of text macros wherever they appear in a line. MASM re-scans the line until there are no more text macros to make substitutions for.

Note

Text macros are always evaluated when they appear in the name or operation fields. The expression operator is required to evaluate a text macro only when the macro appears in the operand field.

This use of the expression operator eliminates the need to do a macro call in order to evaluate a text macro. For example, in the following macro, MASM 5.0 requires a separate macro, `popregs`, to evaluate the text macro `regpushed`:

```
regpushed       EQU     <ax,bx,cx>
                .
                .
                .
RestRegs        MACRO
                popregs %regpushed
                ENDM

popregs MACRO   reglist
                IRP     reg,<reglist>
                   pop       reg
```

```
                ENDM
                ENDM
```

The new use of the expression operator to evaluate text macros in a line makes the `popregs` macro unnecessary:

```
regpushed       EQU     <ax,bx,cx>
                .
                .
                .
RestRegs        MACRO
  %             IRP     reg,<regpushed>   ;; % operator makes
separate macro
                  pop   reg             ;; unnecessary
                ENDM
                ENDM
```

Note

You cannot use the **EQU** directive to assign a value to a text macro in a line evaluated with the expression operator, unless the text macro evaluates to a valid name. For example, the following lines generate an error:

```
strpos          EQU  <[si]+12>
.
.
.
% wpstrpos      EQU  <WORD PTR strpos>
```

On Pass 1, `wpstrpos` is defined as a text macro that is expanded on Pass 2. Thus, on Pass 2 the second **EQU** directive becomes

```
WORD PTR [si]+12      EQU <WORD PTR [si]+12>
```

and generates an error.

Instead, use the **CATSTR** directive to assign values to text macros (see Section 3.1.3, "Text-Macro String Directives," for more information about **CATSTR** and other text-macro string directives). The previous example should be rewritten as follows:

```
strpos          EQU <[si]+12>
.
.
```

```
.
wpstrpos        CATSTR  <WORD PTR >, strpos
```

If the text macro evaluates to a valid name, there is no error when you use **EQU**. The following lines do not generate an error, but define two names, one (`numlabel`) with the value 5, the other (`tmacro`) with the value `<numlabel>`:

```
tmacro    EQU <numlabel>
% tmacro EQU 5
```

You can also use the substitution operator (&) with text macros just as you would inside a macro:

```
SegName           EQU    <MySeg>
.
.
.
% SegName&_text   SEGMENT PUBLIC 'CODE'
```

The final line, after expanding the text macro, becomes:

```
MySeg_text   SEGMENT PUBLIC 'CODE'
```

The substitution operator separates the text-macro name from the text that immediately follows it. The name appears to **MASM** as `segName_text` without the substitution operator, and **MASM** fails to recognize the text macro.

3.1.3 Text-Macro String Directives

Version 5.1 of the Macro Assembler includes four text-macro string directives that let you manipulate literal strings or text-macro values. You use the four directives in much the same way you use the equal sign (=) directive. For example, the following line assigns the first three characters (`abc`) of the literal string to the label `three` by using the **SUBSTR** directive:

```
three    SUBSTR  <abcdefghijklmnopqrstuvwxyz>,1,3
```

Each of the directives assigns its value—depending on the directive—to a numeric label or a text macro. The following list summarizes the four directives and the type of label that the directives should be used with:

Directive	Description
SUBSTR	Returns a substring of its text macro or literal string argument. The **SUBSTR** directive requires a text-macro label.

CATSTR	Concatenates a variable number of strings (text macros or literal strings) to form a single string. The **CATSTR** directive requires a text-macro label.
SIZESTR	Returns the length, in characters, of its argument string. The **SIZESTR** directive requires a numeric label.
INSTR	Returns an index indicating the starting position of a substring within another string. The **INSTR** directive requires a numeric label.

Strings used as arguments in the directives must be text enclosed in angle brackets (< and >), previously defined text macros, or expressions starting with a percent sign (%). Numeric arguments can be numeric constants or expressions that evaluate to constants during assembly.

The four sections below describe the directives in more detail.

■ The SUBSTR Directive

The **SUBSTR** directive returns a substring from a given string. The directive has the following syntax:

textlabel **SUBSTR** *string,start*[[,*length*]]

The **SUBSTR** directive takes the following arguments:

Argument	Description
textlabel	The text label the result is assigned to.
string	The string the substring is extracted from.
start	The starting position of the substring. The first character in the string has a position of one.
length	The number of characters to extract. If omitted, the directive **SUBSTR** returns all characters to the right of position *start*, including the character at position *start*.

In the following lines, the text macro `freg` is assigned the first two characters of the text macro `reglist`:

```
reglist   EQU  <ax,bx,cx,dx>
          .
          .
          .
freg      SUBSTR  reglist,1,2 ; freg = ax
```

■ The CATSTR Directive

The **CATSTR** directive concatenates a series of strings and has the following syntax:

textlabel **CATSTR** *string*[[,*string*]]...

The directive takes the following arguments:

Argument	Description
textlabel	The text label the result is assigned to
string	The string or strings concatenated and assigned to *textlabel*

The following lines concatenate the two literal strings and assign the result to the text macro `lstring`:

```
lstring     CATSTR <a b c d e f g>,  < h i j k l m n o p>
                ; lstring = <a b c d e f g h i j k l m n o p>
```

■ The SIZESTR Directive

The **SIZESTR** directive assigns the length of its argument string to a numeric label and has the following arguments:

numericlabel **SIZESTR** *string*

The **SIZESTR** directive takes the following arguments:

Argument	Description
numericlabel	The numeric label **MASM** assigns the string length to
string	The string whose length is returned

The following line sets `slength` to 8—the length of the text macro `tstring`:

```
tstring     EQU  <ax bx cx>
            .
            .
            .
slength     SIZESTR  tstring ; slength = 8
```

A null string has a length of zero.

■ The INSTR Directive

The **INSTR** directive returns the position of a string within another string. The directive returns 0 if the string is not found. The first character in a string has a position of one. The directive has the following syntax:

numericlabel **INSTR** [[*start*,]]*string1*,*string2*

The following list describes the arguments:

Argument	Description
start	The starting position for the search. When omitted, the **INSTR** directive starts searching at the first character. The first character in the string has a position of one.
numericlabel	The numeric label the substring's position is assigned to.
string1	The string being searched.
string2	The string to look for.

The following lines set `colpos` to the character position of the first colon in `segarg`:

```
segarg   EQU  <ES:AX>
         .
         .
         .
colpos   INSTR  segarg,<:> ; colpos = 3
```

■ Examples

The following macro sets up a series of word storage locations with labels consisting of a name followed by a number:

```
mstore  MACRO slabel,s1,s2
 maxct  = s2 - s1 + 1       ;; Calculate number of locations
 count  = s1
        REPT maxct
 lbl    CATSTR <&slabel>,%count ;; Create label. % forces
                                ;; evaluation of count
        count = count +1
  lbl   DW      0          ;; Create storage location
        ENDM
        ENDM
```

Invoking the macro with the following line creates four storage locations with the names `stuff2`, `stuff3`, `stuff4`, and `stuff5`:

```
mstore <stuff>,2,5
```

The macro first calculates the number of storage locations to allocate, and then loops to generate the required labels and **DW** directives. Notice the use of the expression operator to evaluate `count` so it can be concatenated with the base label name. The angle brackets (<>) around `label` in the line creating the label are necessary so the macro argument becomes a string—the argument type **CATSTR** requires.

The following example uses the text-macro string directives **CATSTR**, **INSTR**, **SIZESTR**, and **SUBSTR**. It defines two macros, `SaveRegs` and `RestRegs`, that save and restore registers on the stack. The macros are written so that `RestRegs` restores only the most recently saved group of registers.

The `SaveRegs` macro uses a text macro, `regpushed`, to keep track of the registers pushed onto the stack. The `RestRegs` macro uses this string to the restore the proper registers. Each time the `SaveRegs` macro is invoked, it adds a pound sign (#) to the string to mark the start of a new group of registers. The `RestRegs` macro restores the most recently saved group by finding the first pound sign in the string, creating a substring containing the saved register names, and then looping and generating **PUSH** instructions.

```
; Initialize regpushed to the null string
regpushed       EQU     <>


; SaveRegs
; Loops and generates a push for each argument register. Saves
; each register name in regpushed.

SaveRegs        MACRO   r1,r2,r3,r4,r5,r6,r7,r8,r9
   regpushed    CATSTR  <#>,regpushed   ;; Mark a new group of
regs

                IRP reg,<r1,r2,r3,r4,r5,r6,r7,r8,r9>
                   IFNB <reg>
                      push reg          ;; Push and record a
register
   regpushed          CATSTR    <reg>,<,>,regpushed
                   ELSE
                      EXITM             ;; Quit on blank argument
                   ENDIF
                ENDM
                ENDM
```

```
; RestRegs
; Generates a pop for each register in the most recently
; saved group

RestRegs        MACRO
   numloc       INSTR  regpushed,"#"    ;; Find location of #
   reglist      SUBSTR regpushed,1,numloc-1 ;; Get list of
registers to pop
   reglen       SIZESTR regpushed       ;; Adjust numloc if # is
not last
                IF reglen GT numloc     ;; item in the string
                   numloc = numloc + 1
                ENDIF
   regpushed    SUBSTR regpushed,numloc ;; Remove list from
regpushed
   %            IRP reg,<reglist>       ;; Generate pop for each
register
                IFNB <reg>
                   pop reg
                ENDIF
                ENDM
                ENDM
```

The following lines from a **MASM** listing show the sample code the macros would generate (a "2" marks lines generated by the macros):

```
SaveRegs      ax,bx
2                     push ax           ;
2                     push bx           ;
  SaveRegs      cx
2                     push cx           ;
  SaveRegs      dx
2                     push dx           ;
  RestRegs
2                     pop dx
  RestRegs
2                     pop cx
  RestRegs
2                     pop bx
2                     pop ax
```

3.1.4 Predefined Text Macros

Version 5.1 of the Macro Assembler includes three new predefined text macros: **@WordSize**, **@Cpu**,and **@Version**. The **@WordSize** text macro returns the word size of the segment word size in bytes. It returns 4 when the word size is 32 bits and 2 when the word size is 16 bits. By default, the segment word size is 16 bits with the

80286 and other 16-bit processors, and 32 bits with the 80386. See Chapter 5, "Defining Segment Structure," in the *Microsoft Macro Assembler Programmer's Guide* for information about using 16- and 32-bit segment word sizes on the 80386.

Note

Version 5.1 of the Macro Assembler requires the use of a special convention when you assemble using the /ML switch—each word in a name must begin with an uppercase letter. Thus, while in MASM Version 5.0 you used **@filename** when assembling with the /ML switch, in MASM Version 5.1 you must use **@FileName**.

This convention does not apply to the predefined equates used with segment directives, such as the equate **@curseg**.

The **@Cpu** text macro returns a 16-bit value containing information about the selected processor. You select a processor by using one of the processor directives such as the **.286** directive. You can use the **@Cpu** text macro to control assembly of processor-specific code. Individual bits in the value returned by **@Cpu** indicate information about the selected processor.

Bit	If Bit = 1
0	8086 processor
1	80186 processor
2	80286 processor
3	80386 processor
7	Privileged instruction enabled (286 and 386)
8	8087 coprocessor instructions enabled
10	80287 coprocessor instructions enabled
11	80387 coprocessor instructions enabled

Because the processors are upwardly compatible, selecting a higher-numbered processor automatically sets the bits indicating lower-numbered processors. For example, selecting an 80286 processor automatically sets the 80186 and 8086 bits.

Bits 4 through 6, 9, and 12 to 15 are reserved for future use and should be masked off when testing.

Note

The **@Cpu** text macro provides only information about the processor selected during assembly by one of the processor directives. It does not provide information about the processor actually used when a program is run.

The following example uses the **@Cpu** text macro to select more efficient instructions available only on the 80186 processor and above:

```
; Use the 186/286/386 pusha instruction if possible
P186  EQU (@Cpu AND 0002h) ; Only test 186 bit--286 and
                           ; 386 set 186 bit as well
.
.
.
IF P186     ; Non-zero if 186 processor or above
   pusha
ELSE
   push ax  ; Do what the single pusha instruction
   push cx  ; does
   push dx
   push bx
   push sp
   push bp
   push si
   push di
ENDIF
```

The **@Version** text macro returns a string containing the version of **MASM** in use. With the **@Version** macro you can write macros for future versions of **MASM** that take appropriate actions when used with inappropriate versions of **MASM**. Currently, the **@Version** macro returns 510 as a string of three characters.

Because the **@Version** macro is undefined in earlier versions, you can use **@Version** to make sure that files using Version 5.1 features are assembled with **MASM** 5.1:

```
IFNDEF @Version
   ; Error if not Version 5.1 or higher
   IF2
      .ERR2
      %out Requires Version 5.1 or higher
   ENDIF
ELSE

; Uses the MASM 5.1 high-level-language features
        .MODEL MEDIUM,C
        .CODE
copyst  PROC  arg1:NEAR PTR, arg2:NEAR PTR
        .
        .
        .
copyst  ENDP

ENDIF
END
```

3.2 The ELSEIF Directive

Version 5.1 of the Macro Assembler includes an **ELSEIF** conditional assembly directive corresponding to each of the **IF** directives. The **ELSEIF** directives provide a more compact and better-structured way of writing some sequences of **ELSE** and **IF** directives. **MASM** Version 5.1 includes the following **ELSEIF** directives:

ELSEIF
ELSEIF1
ELSEIF2
ELSEIFB
ELSEIFDEF
ELSEIFDIF
ELSEIFDIFI
ELSEIFE
ELSEIFIDN
ELSEIFIDNI
ELSEIFNB
ELSEIFNDEF

The following nested **IF** and **ELSE** blocks can be rewritten to use **ELSEIF** directives:

```
; Macro to load register for high-level-language return
; See section 6.1.6 in the mixed-language guide
```

```
FuncRet MACRO arg,length
        LOCAL tmploc
        IF length EQ 1
           mov al,arg
        ELSE
           IF length EQ 2
              mov ax,arg
           ELSE
              IF length EQ 4
                 .DATA
        tmploc   DW     ?
                 DW     ?
                 .CODE
                 mov ax,WORD PTR arg
                 mov tmploc,ax
                 mov ax,WORD PTR arg+2
                 mov tmploc+2,ax
                 mov dx,SEG tmploc
                 mov ax,OFFSET tmploc
              ELSE
                 %OUT Error in FuncRet expansion
                 .ERR
              ENDIF
           ENDIF
        ENDIF
        ENDM
```

This macro can be rewritten as follows, using the **ELSEIF** directives:

```
FuncRet MACRO arg,length
        LOCAL tmploc
        IF length EQ 1
                 mov al,arg
        ELSEIF length EQ 2
                 mov ax,arg
        ELSEIF length EQ 4
                 .DATA
        tmploc   DW     ?
                 DW     ?
                 .CODE
                 mov ax,WORD PTR arg
                 mov tmploc,ax
                 mov ax,WORD PTR arg+2
                 mov tmploc+2,ax
                 mov dx,SEG tmploc
                 mov ax,OFFSET tmploc
        ELSE
                 %OUT Error in FuncRet expansion
                 .ERR
        ENDIF
        ENDM
```

3.3 Local Labels

Version 5.1 of the Macro Assembler provides a way to generate automatic labels for jump instructions. To define a label, use two at signs (@@) followed by a colon (:). To jump to the nearest preceding local label, use **@B** (back) in the jump instruction's operand field; to jump to the nearest following local label, use **@F** (forward) in the operand field.

Local labels are best used for labeling targets of jump instructions when a label would not help someone understand what your program is doing. Major divisions of a program should be marked by regular labels.

Local labels in some cases also provide a convenient way to avoid relying on the size of an instruction. For example, the *Microsoft Macro Assembler Programmer's Guide* uses the following lines to code a conditional far jump (page 191):

```
            cmp   ax,bx
            jge   $+5
            jmp   longjump
            .
            .
longjump:   .
```

Coding with a local label avoids having to know the exact size of the **JMP** instruction:

```
            cmp   ax,bx
            jge   @F
            jmp   longjump
@@:         .
            .
longjump:   .
```

The following lines show the example from page 338 in the *Microsoft Macro Assembler Programmer's Guide:*

```
; DX is 20, unless CX is less than -20, then make DX 30
            mov dx,20
            cmp cx,-20
            jge greatequ
            mov dx,30
greatequ:
```

Here are the same lines rewritten to use a local label:

```
; DX is 20, unless CX is less than -20, then make DX 30
            mov dx,20
            cmp cx,-20
            jge @F
            mov dx,30
@@:
```

Section 4
Directives

Version 5.1 of the Macro Assembler extends the information provided by the .TYPE operator and removes limitations on the use of the COMM directive. Version 5.1 also changes assumptions made after a .CODE directive. In addition, Version 5.1 includes explicit pointer declarations so that you can use the CodeView debugger to view the variable as a pointer during debugging.

4.1 Extensions to .TYPE

The .TYPE operator now returns the bit settings shown in Table 4.1.

Table 4.1
.TYPE Operator and Variable Attributes

Bit Position	If Bit=0	If Bit=1
0	Not program related	Program related
1	Not data related	Data related
2	Not a constant value	Constant value
3	Addressing mode is not direct	Addressing mode is direct
4	Not a register	Expression is a register
5	Not defined	Defined
7	Local or public scope	External scope

If bits 2 and 3 are both zero, the expression involves a register-indirect expression. Bit 6 is reserved.

The use of bits 2, 3, and 4 is new with this version of the Macro Assembler.

The following macro pushes a value or register onto the stack by using the .TYPE operator to test whether or not the argument is a constant:

```
anypush MACRO pushed
        IF ((.TYPE pushed) AND 0004h) ; Non-zero for constant
           mov  ax,pushed             ; Push constant through ax
           push ax
        ELSE
           push pushed                ; Push anything else
directly.
        ENDIF
        ENDM
```

If the macro is invoked with `anypush 1`, it generates the following code:

```
mov  ax,1
           push ax
```

If the argument is not a constant, the register or variable is pushed directly.

4.2 COMM Extension

The COMM directive now accepts structure names for the *size* argument. This enhancement lets you declare communal variables of any size.

In the following example,the COMM directive is used to make the structure variable `today` a communal variable:

```
date  STRUC

 month  DB  ?
 day    DB  ?
 year   DB  ?

date  ENDS

.DATA
COMM   today:date
.
.
.
```

4.3 Changes to the .CODE Directive

In Version 5.1, the .CODE directive always assumes CS is the current segment. This change makes it easier to use multiple code segments in a single module. Version 5.0

assumed CS once at the beginning of the program, making it necessary to use an **ASSUME** for a second segment.

4.4 Data Declarations and CodeView® Information

Version 5.1 of the Macro Assembler extends data definitions to include explicit allocation of a pointer. Pointer-data definitions may now have the following form:

symbol[[**DW** | **DD** | **DF**]]**type PTR** *initialvalue*

For example, in the following fragment, `ndptr` is declared as a near pointer to a `date` structure and is initialized to zero:

```
date        STRUC
  month     DB    ?
  day       DB    ?
  year      DB    ?
date        ENDS
ndptr       DW  date PTR 0
```

Similarly, the following lines declare a string and two pointers to the string. The declaration also initializes the pointers to the address of the string:

```
string      DB  "from swerve of shore to bend of bay"
pstring     DW  BYTE PTR string  ; Declares a near pointer.
fpstring    DD  BYTE PTR string  ; Declares a far pointer.
```

Using an explicit pointer declaration generates CodeView information, allowing the variable to be viewed as a pointer during debugging.

Note

This use of **PTR** is in addition to the use of **PTR** to specify the type of a variable or operator. **MASM** 5.1 determines the meaning of **PTR** by context.

Section 5
High-Level-Language Support

Version 5.1 of the Macro Assembler includes several features that simplify writing assembly language routines for use in high-level-language programs. These features also make it easier to use a single routine with more than one high-level language. The high-level-language features include the following:

- An extension to the **.MODEL** directive automatically sets up naming, calling, and return conventions.
- A modification of the **PROC** directive handles most of the procedure entry automatically. The **PROC** directive saves specified registers, defines text macros for arguments and the types of arguments, and generates stack setup code on entry and stack tear-down code on exit.
- The new **LOCAL** directive allocates local variables from the stack and defines text macros for the variables.
- Version 5.1 provides both local and global labels when you use the high-level-language support features.

This section describes the Macro Assembler's high-level-language features. It does not teach you how to write procedures called from other languages. See Chapter 6, "Assembly-to-High-Level Interface," in the *Microsoft Mixed-Language Programming Guide* for detailed directions on writing assembly-language routines for use in other languages.

5.1 Overview of High-Level-Language Features

The Version 5.1 high-level-language features allow you to write simpler, cleaner routines as demonstrated by the following two routines. The first routine uses the old high-level-language techniques. It is a procedure that can be called from a C program as a function returning an integer. The procedure adds two integers—passed by value—and returns the result in the **AX** register (used by C for returning integer-function values):

```
; Assemble with /MX or /ML to preserve case of procedure name
                PUBLIC _myadd
                .MODEL MEDIUM
                .CODE
_myadd          PROC FAR

arg1            EQU     <WORD PTR [bp+6]>
arg2            EQU     <WORD PTR [bp+8]>

                push    bp              ; Set up stack frame
                mov     bp,sp

                mov     ax,arg1         ; Load first argument
                add     ax,arg2         ; Add second argument

                pop     bp
                ret

_myadd          ENDP
                END
```

Here is the same procedure written using the new high-level-language features:

```
.MODEL MEDIUM,C
                .CODE
myadd           PROC arg1:WORD, arg2:WORD

                mov     ax,arg1         ; Load first argument
                add     ax,arg2         ; Add second argument

                ret

myadd           ENDP
                END
```

In Version 5.0 of **MASM**, a procedure required a **PUBLIC** directive to make the name of the procedure available outside the module. The **PUBLIC** directive is no longer required—when you use a second parameter on the **.MODEL** directive, **MASM** makes all procedure names public. Also notice that the procedure name no longer starts with an underscore: specifying C calling conventions in the **.MODEL** directive automatically adjusts procedure names and labels to follow C conventions.

In the older version of `myadd`, the type and number of arguments were handled as displacements from the base pointer, **BP**. The extension to the **PROC** directive specifies the type and number of arguments passed to the procedure on the stack and generates text macros for the arguments. These macros are equivalent to the ones used in the original version of the procedure.

Notice too that the stack frame has to be set up in the older version. Now, **MASM** generates the code for setting up the stack and restoring it based on information in the high-level-language directives.

Along with simplifying the process of writing procedures to be called from other languages, the high-level-language calling features also make it easier to use a procedure with more than one language. For example, to change `myadd` so that it can be called from Pascal, rather than C, you would change the **.MODEL** directive (assuming the Pascal routine was using call by value):

```
.MODEL  MEDIUM,PASCAL
```

■ Overview of Sections 5.2–5.6

The next section, Section 5.2, describes the new syntax of the **.MODEL** directive and tells you how to use the directive to set naming and calling conventions in your programs. Section 5.3 describes how to use the extensions to the **PROC** directive to declare your procedure's arguments. Using the new **LOCAL** directive to declare local variables on the stack is described in Section 5.4. The final section describes the new type casts that help you debug your procedures with the CodeView debugger, and discusses the new variable and label scoping features.

Note

MASM does not normally display the code generated by the high-level-language support features. You can see the code produced by these features by using the **.LALL** directive or the **/LA** command line option. See Section 6, "Other Features," for information about the **/LA** option.

5.2 Using the .MODEL Directive To Set Naming and Calling Conventions

The extension to the **.MODEL** directive controls three things: how public and external names are handled, the order in which arguments appear on the stack, and what kind of return is done. The **.MODEL** directive has the following syntax:

.MODEL *memorymodel*[[,*language*]]

The *memorymodel* still specifies the memory model to use. However, the *language* parameter is new and tells **MASM** to follow the naming, calling, and return conventions appropriate to the indicated language. In addition, if you use the *language* argument,

MASM automatically makes all procedure names public. You can use C, PASCAL, FORTRAN, or BASIC as the *language* argument. For example, the following .MODEL directive tells MASM to use the naming, calling, and return conventions for BASIC:

```
.MODEL   MEDIUM,BASIC
```

The paragraphs below describe in detail the specific naming, calling, and return conventions indicated by the *language* argument.

If you use C for the language parameter, all public and external names are prefixed with an underscore (_) in the .OBJ file. Specifying any other language has no effect on the names.

Note

The only change MASM makes to a procedure name is to add an underscore for C. MASM does not truncate names in order to match the conventions of specific languages such as FORTRAN or Pascal. See the *Microsoft Mixed-Language Programming Guide* for specific information about name length limitations in specific languages.

In addition to changing the naming conventions, the *language* parameter also affects how arguments passed on the stack are interpreted. If you specify FORTRAN, PASCAL, or BASIC, then MASM assumes the arguments have been pushed onto the stack from left to right—the last argument is nearest to the top of the stack. Specifying C assumes arguments have been pushed on the stack in the opposite order.

MASM makes no assumptions about whether arguments are passed by value or passed by reference. Your assembly routine must explicitly handle the appropriate convention. See the *Microsoft Mixed-Language Programming Guide* for information about passing arguments by value or by reference.

Pascal, FORTRAN, and BASIC programs require the called procedure to remove arguments from the stack. If you specify PASCAL, FORTRAN, or BASIC as the *language* argument, MASM replaces your return instruction (**ret**) with a return that removes the correct number of bytes from the stack. For example, in a procedure called from BASIC with two integer arguments, MASM would replace the **ret** instruction with the following instruction so the two integer arguments (four bytes) are removed from the stack:

```
ret  4
```

In C, the calling program removes arguments from the stack, and MASM leaves the return instruction unchanged.

Note

To write procedures for use with more than one language, you can use text macros for the memory model and language arguments, and define the values from the command line. For example, the following **.MODEL** directive uses text macros for the *memory* and *language* arguments:

```
%   .MODEL memmodel,lang   ; Use % to evaluate memmodel, lang
```

The values of the two text macros can be defined from the command line using the **/D** switch:

```
MASM /Dmemmodel=MEDIUM /Dlang=BASIC
```

5.3 Declaring Parameters with the PROC Directive

The **PROC** directive in Version 5.1 of the Macro Assembler includes new arguments that specify automatically saved registers, define arguments to the procedure, and set up text macros to use for the arguments. For example, the following **PROC** directive could be placed at the beginning of a procedure called from BASIC that takes a single argument passed by value and that uses (and must save) the **DI** and **SI** registers:

```
myproc      PROC  FAR USES DI SI, arg1:WORD
```

The **PROC** directive has the following syntax:

name **PROC** [[**NEAR**|**FAR**]] [[**USES** [[*reglist*]],]] [[*argument*[[,*argument*]]...]]

The **NEAR** and **FAR** keywords indicate whether you invoke the procedure with a near call or a far call, just as they did in the old form of the directive.

The following list describes the other parts of the **PROC** directive:

Argument	Description
name	The name of the procedure. **MASM** automatically adds an underscore to the beginning of the name if you specify **C** as the language in the **.MODEL** directive.
reglist	A list of registers that the procedure uses and that should be saved on entry. Registers in the list must be separated by blanks or tabs.

argument	The arguments passed to the procedure on the stack. See the discussion below for the syntax of the *argument*.

The *argument* indicates the type of each of the procedure's arguments and is separated from the *reglist* argument by a comma if there is a list of registers. The *argument* has the following syntax:

argument[[:[[[**NEAR**|**FAR**]]**PTR**]]*type*]]...

The *argument* is the name of the argument. The *type* is the type of the argument and may be **WORD**, **DWORD**, **FWORD**, **QWORD**, **TBYTE**, or the name of a structure defined by a **STRUC** structure declaration (see Chapter 6, "Defining Labels and Variables," in the *Microsoft Macro Assembler Programmer's Guide* for more information about types). If you omit *type*, the default is the **WORD** type (the **DWORD** type when a **.386** directive is used).

MASM creates a text macro for each argument. You can use these text macros to access the arguments in the procedure. The text macro is similar to the text macros that appear in the first example procedure in Section 5.1, "Overview," above.

The **FAR**, **NEAR**, **PTR**, and *type* arguments are all optional. If you omit all of them, **MASM** assumes the variable is a **WORD** type. If you use only the *type* argument, **MASM** assumes the variable has the indicated type.

Note

If you are writing a routine to be called from BASIC, FORTRAN, or Pascal, and the routine returns a function value, you must declare an additional parameter if you return anything other than a two- or four-byte integer. See Section 6.1.6, "Returning a Value," in the *Microsoft Mixed Language Programming Guide* for more information.

If you specify that the variable is a pointer, **MASM** sets up a text macro to access the variable on the stack, but also generates CodeView information so that the variable is treated as a pointer during debugging. **MASM** assumes specific sizes for the variable, depending on the combination of **NEAR**, **FAR**, and **PTR** arguments you specify. The lines below show some example combinations of **NEAR**, **FAR**, **PTR**, and *type*. Each example is discussed below.

```
myproc    PROC var1:PTR, var2:PTR DWORD
          .
          .
          .
myproc    ENDP

proc2     PROC var3:FAR PTR, var4:NEAR PTR
          .
          .
          .
proc2     ENDP
```

If you use the **PTR** argument alone, as in the declaration for `var1`, **MASM** makes the variable a **WORD**, **DWORD**, or **FWORD** type depending on the memory model and segment word size (segment word size can only be changed in code for the 386 processor). The **WORD** type is used for small and medium memory models; the **DWORD** type for all other memory models using a 16-bit segment word size; the **FWORD** type for compact, large, and huge memory models when you specify 32-bit segment word sizes for a 386 processor; and the **DWORD** type for small and medium memory models with 32-bit segments.

When you use a combination of **PTR** and *type*, as in the declaration for `var2`, **MASM** assigns the type exactly the same way as when you use **PTR** alone. Thus, for a given combination of memory model and segment word size, using either **PTR** alone or in combination with a *type* generates the same text macro. For example, all of the following declarations of `procvar` produce the same text macro for the variable name, although they generate different CodeView information:

```
aproc   PROC procvar:PTR

aproc   PROC procvar:PTR DWORD

aproc   PROC procvar:PTR BYTE
```

Specifying a particular type changes only the CodeView information, not the text macro produced.

If you specify a **NEAR PTR** or **FAR PTR** argument, as in the declarations of `var3` and `var4`, **MASM** ignores the memory model you've selected and assigns a **WORD** type for a **NEAR PTR** argument and a **DWORD** type for a **FAR PTR** argument. **MASM** assigns an **FWORD** type for a **FAR PTR** argument or a **DWORD** type for a **NEAR PTR** argument when you specify 32-bit segment word size for a 386 processor.

MASM does not generate any code to get the value or values the pointer references: your program must still explicitly treat the argument as a pointer. For example, the procedure in Section 5.1 can be rewritten for use with BASIC so that it gets its argument by near reference (the BASIC default):

```
; Call from BASIC as a FUNCTION returning an integer

        .MODEL MEDIUM,BASIC
        .CODE
myadd   PROC  arg1:NEAR PTR WORD, arg2:NEAR PTR WORD

        mov   bx,arg1  ; Load first argument
        mov   ax,[bx]
        mov   bx,arg2  ; Add second argument
        add   ax,[bx]

        ret

myadd   ENDP
        END
```

In the example above, even though the arguments are declared as near pointers, you still must code two move instructions in order to get the values of the arguments—the first move gets the address of the argument, the second move gets the argument.

You can use conditional assembly directives to make sure that your pointer arguments are loaded correctly for the memory model. For example, the following version of `myadd` treats the arguments as far arguments if necessary:

```
.MODEL MEDIUM,C
                .CODE
myadd           PROC arg1:PTR, arg2:PTR

                IF @DataSize
                   les bx,arg1           ; Far arguments
                   mov ax,WORD PTR [bx]
                   les bx,arg2
                ELSE
                   mov bx,arg1           ; Near arguments
                   mov ax,[bx]
                   mov bx,arg2
                ENDIF
                add ax,[bx]           ; Add the values

                ret

myadd           ENDP
                END
```

Note

When you use the high-level-language features and **MASM** encounters a return instruction, it automatically generates instructions to pop saved registers, remove local variables from the stack, and, if necessary, remove arguments.

5.4 Local Variables

With the **LOCAL** directive, you can allocate local variables from the stack. Usually this is done by decrementing the stack pointer the required number of bytes after setting up the stack frame. For example, the following sequence allocates two, two-byte variables from the stack and sets up text macros to access them:

```
locvar1   EQU  <WORD PTR [bp-2]>
locvar2   EQU  <WORD PTR [bp-4]>
          .
          .
          .
          push  bp
          mov   bp,sp
          sub   sp,4
          .
          .
          .
          mov   locvar1,ax
```

With the **LOCAL** directive you can do the same thing in a single line:

```
LOCAL locvar1:WORD, locvar2:WORD
          .
          .
          .
          mov   locvar1,ax
```

The **LOCAL** directive has the following syntax:

LOCAL *vardef* [[,*vardef*]]...

Each *vardef* has the form:

variable[[[*count*]]][[:[[[**NEAR** | **FAR**]]**PTR**]]*type*]]]...

The **LOCAL** directive arguments are as follows:

Argument	Description
variable	The name given to the local variable. **MASM** automatically defines a text macro you may use to access the variable.
count	The number of elements of this name and type to allocate on the stack. Using *count* allows you to allocate a simple array on the stack. The brackets around *count* are required.
type	The type of variable to allocate. The *type* argument may be one of the following: **WORD**, **DWORD**, **FWORD**, **QWORD**, **TBYTE**, or the name of a structure defined by a **STRUC** structure declaration.

MASM sets aside space on the stack, following the same rules as for procedure arguments.

MASM does not initialize local variables. Your program must include code to perform any necessary initializations. For example, the following code fragment sets up a local array and initializes it to zero:

```
arraysz EQU     20

aproc   PROC
        LOCAL var1[arraysz]:WORD, var2:WORD
        .
        .
        .
; Initialize local array to zero
        mov     cx,arraysz
        xor     ax,ax
        xor     di,di           ; Use di as array index
repeat: mov     var1[di],ax
        inc     di
        inc     di
        loop    repeat
; Use the array...
        .
        .
        .
```

5.5 Variable Scope

When you use the extended form of the **.MODEL** directive, **MASM** makes all identifiers inside a procedure local to the procedure. Labels ending with a colon (:), procedure arguments, and local variables declared in a **LOCAL** directive are undefined outside of the procedure. Variables defined outside of any procedure are available inside a procedure. For example, in the following fragment, `var1` can be used in `proc1` and `proc2`, while `var2`—because it is defined in `proc2`—is not available to `proc1`:

```
.MODEL MEDIUM,C
        .DATA
var1    DW      256     ; Available to proc1 and proc2

        .CODE
proc1   PROC
        .
        .
        .
exit:   ret
proc1   ENDP

proc2   PROC
        LOCAL var2:WORD ; This var2 only available in proc2
        .
        .
        .
exit:   ret
proc2   ENDP
```

If `proc1` contained a **LOCAL** directive defining `var2`, that `var2` would be a completely different variable than the `var2` in `proc2`.

Notice that both procedures contain the label `exit`. Because labels are local when you use the language option on the **.MODEL** directive, you may use the same labels in different procedures. You can make a label in a procedure global (make it available outside the procedure) by ending it with two colons:

```
proc3     PROC
          .
          .
          .
label1::
          .
          .
          .
proc3     ENDP
```

In the preceding example, `label1` is available throughout the file containing `proc3`.

Section 6
Other Features

This chapter describes the new line-continuation and listing features of Version 5.1 of the Macro Assembler.

6.1 Line Continuation

You can create program lines that extend over more than one physical line by using the line-continuation character (\):

```
PUBLIC  putstr, getstr, compstr, savestr, strcat, \
            indexstr, strsub
        .MODEL  medium
        .CODE
putstr PROC
        .
        .
        .
```

The backslash must be the last character in the line. Using the line-continuation character, you may have program lines up to 512 characters. Physical lines are still limited to 128 characters.

Continued lines are marked with a backslash (\) in the listing file.

Note

A backslash in a comment does not continue the line. For example, the backslash at the end of the following line is part of the comment and does not continue the line.

```
xor ax,ax  ; A line is not continued if it's a comment\
```

6.2 List All (/LA)

Version 5.1 of the Macro Assembler includes a new command-line option that shows all code generated by **MASM**. The **/LA** option displays the results of the simplified segment directives and the code generated by the high-level-language support features. For example, the following command line assembles `STRPKG.ASM` and includes all code in the listing file:

```
MASM /LA STRPKG.ASM
```

Microsoft® CodeView® and Utilities

Software Development Tools

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

Update

Microsoft Corporation

Contents

Figures

Section 1
Introduction

This update supplements the Microsoft® CodeView® and Utilities manual, and describes utilities that are designed for use with Microsoft Windows and the OS/2 systems. The update also describes improvements which apply to both real-mode and protected-mode environments, including new features of the Microsoft CodeView debugger. The pages that follow use the term "OS/2" to refer to both Microsoft Operating System/2 (MS® OS/2) and IBM® OS/2. Similarly, the term "DOS" is used to refer to both MS-DOS® and IBM Personal Computer DOS.

The development of a protected-mode program under OS/2 differs from the development of a real-mode program in the following way: to make calls to OS/2 you must call a dynamic-link library. (As explained in Section 3, "About Linking in OS/2," a dynamic-link library is not linked to the program but is loaded separately at run time.) The use of a dynamic-link library, in turn, requires that the program know where its dynamic-link functions are defined. Module-definition files and import libraries, described below, serve this purpose. OS/2 programs can also take advantage of multiple threads, which are parts of your program that run concurrently. (Threads are like processes, but they are faster to create, and they share the same code segment.)

The following list describes what you can do with the new utilities and the new version of the CodeView debugger:

- **View structures with the CodeView debugger**

  In addition to the capabilities described in the Microsoft CodeView and Utilities manual, both versions of the debugger (protected mode and real mode) provide the ability to watch a C or **MASM** structure, Pascal record, or BASIC user-defined type. The debugger displays, labels, and dynamically updates each element of the structure, and allows you to trace through a linked list with a simple keystroke or mouse selection.

- **Debug multiple-thread programs with CVP**

  The protected-mode CodeView debugger, **CVP**, expands the capabilities of the debugger as described in the Microsoft CodeView and Utilities manual. The protected-mode debugger can debug code in dynamic-link libraries, and it helps you debug multiple-thread programs by providing a new command. This command lets you view the state of the machine while one thread or another is being traced. You can also freeze some threads while the others run concurrently.

- **Link real- and protected-mode programs**

  Version 5.0 of the Microsoft Segmented-Executable Linker (**LINK**) takes an additional field for module-definition files (module-definition files are documented in Section 7). The use of a module-definition file makes it possible for you to create dynamic-link libraries, specify dynamic-link entry points for functions, and provide other kinds of information for your OS/2 program modules.

- **Create import libraries with IMPLIB**

  Import libraries (described in Section 3, "About Linking in OS/2," and Section 6, "The IMPLIB Utility") can speed up the development process for OS/2 applications. When you create a dynamic-link library, you can provide an import library to the application developer who calls your dynamic-link library. The import library is easy to link, and saves the developer the trouble of creating a module-definition file. The Microsoft Import Library Manager utility (**IMPLIB**) generates import libraries for you.

- **Use BIND to create dual-mode applications**

  By using the Microsoft Operating System/2 Bind utility (**BIND**), you can convert an OS/2 program so that it can run in either OS/2 protected mode or in real mode (DOS 3.x or compatibility box). Section 5 gives instructions for using the **BIND** utility.

- **Link faster with ILINK**

  For large OS/2 and Windows programs, the Microsoft Incremental Linker (**ILINK**) can speed up linking by as much as 20 times. This utility takes advantage of the new segmented-executable file format, by relinking only those modules which have changed. Section 9 explains how the process works.

1.1 System Requirements

To use all of the utilities presented in this manual, you need to have MS OS/2 installed and running in protected mode.

In addition, if you want to call the operating system or take advantage of OS/2-specific features such as threads or semaphores, you will need to have documentation on all the Application Program Interface (API) calls (see the *Microsoft Operating System/2 Programmer's Reference*).

The following programs will also run in real mode (DOS 3.x or OS/2 compatibility box):

- **LINK**
- **CV** (but not **CVP**)
- **MAKE**
- **ILINK**

1.2 Installation

The MS OS/2 languages include two versions of the Microsoft CodeView debugger, one for each OS/2 operating mode. For debugging programs running in the protected mode, the CodeView debugger's executable file is **CVP.EXE**, and the help file is **CVP.HLP**. Both should be installed in a directory listed in the **PATH** environment variable. To debug programs running in real mode, use the **CV.EXE** executable file and its help file, **CV.HLP**. Both of these files should also be installed in a directory listed in the **PATH** environment variable.

Finally, you should also install the executable files **LINK**, **EXEC**, **ILINK**, **BIND**, and **IMPLIB** in a directory listed in the **PATH** environment variable.

Note

This document uses certain notational conventions to convey example and syntax information for various utilities. See the "Introduction" to the Microsoft CodeView and Utilities manual for an explanation of these conventions.

Within this update, command-line options are preceded by a forward slash (/). However, in all cases where a slash is used, you can enter either a slash or a dash (-).

Section 2
Using the CodeView Debugger

This chapter first presents two new features—structure watching and text selection—which are included in both the protected-mode CodeView debugger (**CVP.EXE**) and the real-mode CodeView debugger (**CV.EXE**). The chapter then describes the special features that are included only in the protected-mode debugger. Finally, the chapter describes the Microsoft Debug Information Compactor utility (**CVPACK**), which reduces the size of the executable file.

2.1 New Debugging Features

The CodeView debugger now provides two direct ways to examine values of members of structures. First, you can now specify a structure name in a Watch command or Evaluate Expression command (see Section 2.1.1, "Placing Structures in the Watch Window"). Second, the debugger provides a new command for viewing structures in a dialog box. This new command is described in Section 2.1.2, "Using the Graphic Display Command."

Note

For ease of discussion, Section 2.1.1, "Placing Structures in the Watch Window," uses the general term "structures" to refer to Pascal records and BASIC user-defined types, as well as C structures. The machine-level implementation of these records, types, and structures is similar, so the debugger handles them in similar ways.

The debugger also provides text selection, which permits you to use a mouse (Microsoft Mouse or compatible) to select text on screen as input to commands. This capability is described in Section 2.1.3, "Selecting Text."

2.1.1 Placing Structures in the Watch Window

Assume that you have declared a structure as follows:

```
struct stype {
int   a;
int   b;
struct {
    int   x;
    long  y;
} c;
    struct stype *new;
} sample = {11, 12, {100, 200} };
```

If you give the Watch command `W?sample`, then the debugger displays the following line in the watch window:

```
sample : { a=11, b=12, c={x=100, y=200}, new=0x0000:0x0000 }
```

Note the following features, as shown in the above example:

- Nested structures are displayed in a nested pair of braces. (The debugger displays structures nested to any level!)
- Fields other than nested structures are displayed in their default format, as described in the Microsoft CodeView and Utilities manual. For example, a pointer is always displayed in the standard *segment:offset* form. (The example above assumes the C hexadecimal notation.)

2.1.2 Using the Graphic Display Command

The new Graphic Display command (**??**) is even more powerful than the Watch and Evaluate Expression commands. This command is especially useful for examining nested structures and linked lists of pointers. The syntax of the command is simple:

?? *variable*[[,**c**]]

In the syntax display above, *variable* can be any recognized data symbol. The optional format specifier **c** can be used to specify that one-byte-length fields be displayed as ASCII (American Standard Code for Information Interchange) characters.

The debugger responds by displaying a dialog box. If *variable* is a structure, then the dialog box contains the name and value of each field. For example, if the structure `sample` is defined as described in the previous section, then the command `??sample` produces the following dialog box:

```
                          x
-------------------------------------------------
a                         11
b                         12
c                         {...}
new                       0x0000:0x0000
```

Nested structures, such as c in the example above, are evaluated as { . . . }. In addition, the dialog box displays a null-terminated ASCII string next to any field which contains a character pointer.

You can use the Graphic Display command with variables other than structures. With nonstructure variables, the command displays just one field.

The Graphic Display command enables you to expand structures and dereference pointers by selecting a field. (These actions are defined below.) To select a field with the keyboard, press the up and down DIRECTION keys to move the cursor to the field you wish to select, and then press ENTER. To select a field with the mouse, simply click the left mouse button on the field you wish to select (or anywhere on the same line). Selecting a field has the following effect:

1. If the field contains a nested structure, then the structure is "expanded"; the nested structure becomes the new subject of the dialog box. The dialog box displays each field of the nested structure.

2. If the field contains a pointer, then the pointer is "dereferenced"; in other words, the debugger locates the data which the pointer addresses. This data becomes the new subject of the dialog box.

   The pointer's type determines how the debugger displays the dereferenced data. The debugger uses this type information even if the pointer does not currently address any meaningful data. If the pointer addresses a structure, then each field of the new structure is displayed.

3. If the field contains neither a pointer nor a nested structure, then selection has no effect. The debugger beeps to tell you that the selected field was neither a pointer nor a structure.

You can return to the previous dialog-box display (before expansion or dereference took place) by pressing the backspace key or by clicking right (press the righthand mouse button).

While the dialog box is on screen, you cannot execute other CodeView-debugger commands. To remove the dialog box and resume normal debugger operation, press ESC, or click left while the mouse cursor is outside the box.

Note

You can take advantage of the new Watch-command capability in either window or sequential mode. To use the Graphic Display command, however, you need to run the debugger in window mode.

2.1.3 Selecting Text

Text selection is a technique that you can use with the mouse. Select text from either the display or dialog window by holding down the left mouse button and dragging the mouse cursor to the left or right. All text up to the mouse cursor is selected when you release the button. Once selected, you can use the text in one of two ways:

1. The selected text automatically appears in the next dialog prompt box. For example, when you select Find from the Search menu, a dialog box prompts you for the search string. Your selected text appears in this box. You can edit the text or press ENTER immediately.

2. The selected text appears in the dialog window (at the end of the dialog-window buffer) when you press SHIFT+INS. If you then press ENTER, the text is given to the debugger as a command.

The selected text can only be used once. To use the same text repeatedly, you need to reselect the text after each use.

2.2 The Protected-Mode CodeView Debugger

The protected-mode CodeView debugger (**CVP.EXE**) differs from the real-mode CodeView debugger (**CV.EXE**) in three principal ways:

1. The View Output Screen command (\) works differently.

2. **CVP** takes an additional command-line option for use in debugging dynamic-link modules.

3. **CVP** can debug multiple-thread programs. In order to deal with the multiple-thread capability of OS/2, **CVP** has a new command that is not present in **CV**, and some of the commands for tracing and execution in **CV** work differently in **CVP**.

Each of these differences is described in the sections below. You should also bear in mind the following general limitations when using **CVP** in the OS/2 environment:

- Only one copy of the CodeView debugger can be run at a time in the protected mode. Multiple copies cannot be run in concurrent screen groups.

- When you debug a program without using the **/2** option, and the program makes dynamic-link calls to functions outside the **API**, the debugger will not have access to the program's environment or the current drive and directory.

In all other respects, the CodeView debugger's operation as described in the Microsoft CodeView and Utilities manual applies to both versions.

2.2.1 Using the Debugger's View Output Command

When you switch display to the output window with **CVP**, by using the View Output command (\), you won't stay there indefinitely as you would with the real-mode CodeView debugger. Instead, you will jump back to the CodeView screen after a 3-second delay. A different delay period (as measured in seconds) can be specified with a number following the View Output command, as in the following example:

```
\60
```

The example above directs the debugger to display the output window for 60 seconds, before returning to the debugging screen.

Another way to view the output is to go back to the Session Manager screen and select the screen group labeled **CVP APP**. This is the screen group owned by the application that is being debugged. When you have finished viewing the output window, switch back to the **CVP.EXE** screen group. You can use ALT+ESC to toggle between screen groups.

2.2.2 Debugging Dynamic-Link Modules

The protected-mode CodeView debugger (**CVP**) can debug dynamic-link modules, but only if it is told what libraries to search at run time. For more information on dynamic-link libraries, refer to the *Microsoft Operating System/2 Programmer's Guide*, and to the **IMPLIB** and module-definition sections in this update (Sections 6 and 7, respectively).

When you place a module in a dynamic-link library, neither code nor symbolic information for that module is stored in the executable (**.EXE**) file; instead, the code and symbols are stored in the library and are not brought together with the main program until run time.

Thus, the protected-mode debugger needs to search the dynamic-link library for symbolic information. Because the debugger does not automatically know what libraries to look for, **CVP** has an additional command-line option which enables you to specify dynamic-link libraries:

■ **Syntax**

/L *file*

The **/L** option directs the CodeView debugger to search *file* for symbolic information. When you use this option, at least one space must separate **/L** from *file*.

■ **Example**

```
CVP /L DLIB1.DLL /L GRAFLIB.DLL PROG
```

In the example above, **CVP** is invoked to debug the program **PROG.EXE**. To find symbolic information needed for debugging each module, **CVP** will search the libraries `DLIB1.DLL` and `GRAFLIB.DLL`, as well as the executable file `PROG.EXE`.

2.2.3 Debugging Multiple-Thread Programs

A program running in OS/2 protected mode has one or more threads. As explained in the programmer's guide, threads are the fundamental units of execution; OS/2 can execute a number of different threads concurrently. A thread is similar to a process, yet it can be created or terminated much faster. Threads begin at a function-definition heading, in the same program in which they are invoked.

The existence of multiple threads within a program presents a dilemma for debugging. For example, thread 1 may be executing source line 23 while thread 2 is executing source line 78. Which line of code does the CodeView debugger consider to be the current line?

Conversely, you cannot always tell which thread is executing just because you know what the current source line is. In OS/2 protected mode, you can write a program in which two threads enter the same function.

In Figure 2.1, the function `main` uses the **DOSCREATETHREAD** system call to begin execution of thread 2. The function `f2` is the entry point of the new thread. Thread 2 begins and terminates inside the function `f2`. Before it terminates, however, thread 2 can enter other functions by means of ordinary function calls.

Thread 1 begins execution in the function `main`, and thread 2 begins execution in the function `f2`. Later, both thread 1 and thread 2 enter the function `f3`. (Note that each thread returns to the proper place because each thread has its own stack.) When you use the debugger to examine the behavior of code within the function `f3`, how can you tell which thread you are tracking?

The protected-mode CodeView debugger solves this dilemma by using a modified CodeView prompt, and by providing the Thread command, which is only available with **CVP**.

The command prompt for the protected-mode CodeView debugger is preceded by a three-digit number indicating the current thread.

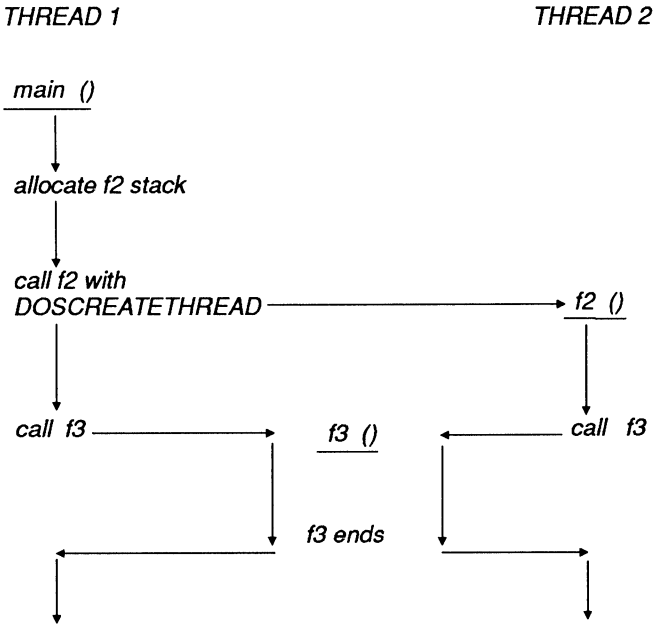


Figure 2.1 Multiple-Thread Program

■ Example

```
001>
```

The example above displays the protected-mode CodeView prompt, indicating that thread 1 is the current thread. Thread 1 is always the current thread when you begin a program. If the program never calls the **DOSCREATETHREAD** function, then thread 1 will remain the only thread.

Each thread has its own stack and its own register values. When you change the current thread, you will see several changes to the CodeView-debugger display:

- The CodeView prompt will display a different three-digit number.
- The register contents will all change.
- The current source line and current instruction will both change, to reflect the new value of **CS:IP**. If you are running the debugger in window mode, you will likely see different code in the display window.

- The Calls menu and the Stack Trace command will display a different group of functions.

The rest of this section discusses the Thread command, and lists other CodeView commands that may work differently because of multiple threads.

■ Syntax

The syntax of the Thread command is displayed below:

~[[*specifier*[[*command*]]]]

In the syntax display above, the *specifier* determines to which thread or threads the command will apply. You can specify all threads, or just a particular thread. The *command* determines what activity the debugger will carry out with regard to the specified thread. For example, you can execute the thread, freeze its execution, or select it as the current thread. If you omit *command*, the debugger displays the status of the specified thread. If you omit both the *command* and *specifier*, then the debugger displays the status of all threads.

The status display for threads consists of the two fields

thread-id thread-state

in which *thread-id* is an integer, and *thread-state* has the value `runnable` or `frozen`. All threads not frozen by the debugger are displayed as `runnable`; this includes threads that may be blocked for reasons that have nothing to do with the debugger, such as a critical section.

The legal values for *specifier* are listed below, along with their effects.

Symbol	Function
(blank)	Displays the status of all threads. If you omit the *specifier* field you cannot enter a *command*. Instead, you simply enter the tilde (~) by itself.
#	Specifies the last thread that was executed. This thread is not necessarily the current thread. For example, suppose you are tracing execution of thread 1, and then switch the current thread to thread 2. Until you execute some code in thread 2, the debugger still considers thread 1 to be the last thread executed.
*	Specifies all threads.

n	Specifies the indicated thread. The value of *n* must be a number corresponding to an existing thread. You can determine corresponding numbers for all threads by entering the command `~*`, which gives status of all threads.
.	Specifies the current thread.

The legal values for *command* are listed below, along with their meanings.

Command	Function
(blank)	The status of the selected thread (or threads) is displayed.
BP	A breakpoint is set for the specified thread or threads. As explained earlier, it is possible to write your program so that the same function is executed by more than one thread. By using this version of the Thread command, you can specify a breakpoint that applies only to a particular thread. The letters **BP** are followed by the normal syntax for the Breakpoint Set command, as described in the Microsoft CodeView and Utilities manual. Therefore you can include the optional passcount and command fields.
E	The specified thread is executed in slow motion. When you specify a single thread with **E**, then the specified thread becomes the current thread, and is executed without any other threads running in the background. The command `~*E` is a special case. It is legal only in source mode, and executes the current thread in slow motion, but lets all other threads run (except those that are frozen). You will only see the current thread executing in the debugger display.
F	The specified thread (or threads) is frozen. A frozen thread will not run in the background or in response to the debugger Go command. However, if you use the **E**, **G**, **P**, or **T** variation of the Thread command, then the specified thread will be temporarily unfrozen while the debugger executes the command.
G	Control is passed to the specified thread, until it terminates or until a breakpoint is reached. If you give the command `~*G`, then all threads will execute concurrently (except for those that are frozen). If you specify

a particular thread, then the debugger will temporarily freeze all other threads and execute the specified thread.

P The debugger executes a program step for the specified thread.

If you specify a particular thread, then the debugger executes one source line or instruction of the thread. All other threads are temporarily frozen. This version of the Thread command does not change the current thread. Therefore if you specify a thread other than the current thread, you will not see immediate results. However, the subsequent behavior of the current thread may be affected.

The command `~*P` is a special case. It is legal only in source mode, and causes the debugger to step to the next source line, while letting all other threads run (except for those that are frozen). You will only see the current thread execute in the debugger display.

S The specified thread is selected as the current thread.

This version of the Thread command can apply to only one thread at a time. Thus, the command `~*S` results in an error message. Note that the command `~.S` is legal, but has no effect.

T The specified thread is traced.

This version of the Thread command works in a manner identical to **P**, described above, except that **T** traces through function calls and interrupts, whereas **P** does not.

U The specified thread or threads are unfrozen. This command reverses the effect of a freeze.

Note

With the Thread command, only the S (select) and the E (execute) variations cause the debugger to switch the current thread. However, when a thread causes program execution to stop by hitting a breakpoint, the debugger will select that thread as the current thread.

You can prevent the debugger from changing the current thread, by including the breakpoint command `"~.S"`. This command directs the debugger to switch to the current thread rather than the thread that hit the breakpoint. For example, the following command sets a breakpoint at line `120` and prevents the current thread from changing:

```
BP .120 "~.S"
```

■ Syntax

The syntax display below summarizes all the possible entries to the Thread command:

~{#|*|*n*|.}[[BP|E|F|G|P|S|T|U]]

Note that you must include one of the symbols from the first set (which gives possible values for the specifier), but you do not have to include a symbol from the second set (which gives possible values for the command).

■ Examples

```
004>~
```

The example above displays the status of all threads, including their corresponding numbers.

```
004>~2
```

The example above displays the status of thread 2.

```
004>~5S
```

The example above selects thread 5 as the current thread. Since the current thread was 4 (a fact apparent from the CodeView prompt), this means that the current thread is changing and therefore we can expect the registers and the code displayed to all change.

```
005>~3BP .64
```

The example above sets a breakpoint at source line 64, which stops program execution only when thread 3 executes to this line.

```
005>~1F
```

The example above freezes thread 1.

```
005>~*U
```

The example above thaws (unfreezes) all threads; any threads that were frozen before will now be free to execute whenever the Go command is given. If no threads are frozen, this command has no effect.

```
005>~2E
```

The example above selects thread 2 as the current thread, then proceeds to execute thread 2 in slow motion.

```
002>~3S
003>~.F
003>~#S
002>
```

The example above selects thread 3 as the current thread, freezes the current thread (thread 3), and then switches back to thread 2. After we switched to thread 3, no code was executed; therefore the debugger considers the last-thread-executed symbol (#) to refer to thread 2.

Whether or not you use the Thread Command, the existence of threads affects your CodeView debugging session at all times. Particular debugger commands are strongly affected. Each of these commands is discussed below.

Command	Behavior in Multiple-Thread Programs
.	The Current Line command always uses the current value of **CS:IP** to determine what the current instruction is. Thus, the Current Line command applies to the current thread.
E	When the debugger is in source mode, the Execute command is equivalent to the `~*E` command; the current thread is executed in slow motion while all other threads are also running. When the debugger is in mixed or assembly mode, the Execute command is equivalent to the command `~.P`, which does not let other threads run concurrently.
BP	The Set Breakpoint command is equivalent to the `~*BP` command; the breakpoint applies to all threads.

G	The Go command is equivalent to the `~*G` command; control is passed to the operating system, which executes all threads in the program except for those that are frozen.
P	When the debugger is in source mode, the Program Step command is equivalent to the command `~*P`, which lets other threads run concurrently. When the debugger is in mixed or assembly mode, the Program Step command is equivalent to the command `~.P`, which lets no other threads run.
K	The Stack Trace command displays the stack of the current thread.
T	When the debugger is in source mode, the Trace command is equivalent to the command `~*T`, which lets other threads run concurrently. When the debugger is in mixed or assembly mode, the Trace command is equivalent to the command `~.T`, which lets no other threads run.

In general, CodeView-debugger commands apply to all threads, unless the nature of the command makes it appropriate to deal with only one thread at a time. (For example, since each thread has its own stack, the Stack Trace command does not apply to all threads.) In the later case, the command applies to the current thread only.

2.3 Saving Memory with the CVPACK Utility

After you compile and link a program with CodeView debugging information, you can use the Microsoft Debug Information Compactor utility (**CVPACK**) to reduce the size of the executable file. **CVPACK** compresses the debugging information in the file, and allows the CodeView debugger to load larger programs without running out of memory.

The **CVPACK** utility has the following command line:

CVPACK [[**/p**]] *exefile*

The **/p** option results in the most effective possible packing, but causes **CVPACK** to take longer to execute. When the **/p** option is specified, unused debugging information is discarded, and the packed information is sorted within the file. When the **/p** option is not specified, packed information is simply appended to the end of the file.

To debug a file that has been altered with **CVPACK**, you must use Version 2.10 or later of the CodeView debugger.

Section 3
About Linking in OS/2

In most respects, linking a program using the Microsoft Segmented-Executable Linker Version 5.0 (**LINK**) for the OS/2 environment is similar to linking a program for the DOS 3.x environment. The principal difference is that most programs created for the DOS 3.x environment run as stand-alone applications, whereas programs that run under OS/2 protected mode generally call one or more "dynamic-link libraries."

A dynamic-link library contains executable code for common functions, just as an ordinary library does. Yet code for dynamic-link functions is not linked into the executable (**.EXE**) file. Instead, the library itself is loaded into memory at run time, along with the **.EXE** file.

Each **.DLL** file (dynamic-link library) must use "export definitions" to make its functions directly available to other modules. At run time, functions not exported can only be called from within the same file. Each export definition specifies a function name.

Conversely, the **.EXE** file must use "import definitions" that tell where each dynamic-link function can be found. Otherwise, OS/2 would not know what dynamic-link libraries to load when the program is run. Each import definition specifies a function name and the **.DLL** file where the function resides.

Assume the simplest case, in which you create one application and one dynamic-link library. The linker requires export and import definitions for all dynamic-link function calls. The OS/2 operating system provides two ways to supply these definitions:

1. You create one module-definition file (**.DEF** extension) with export definitions for the **.DLL** file, and another module-definition file with import definitions for the **.EXE** file. The module-definition files provide these definitions in an ASCII format.

2. You create one module-definition file (**.DEF** extension) for the **.DLL** file, and then generate an import library to be linked to the **.EXE** file.

The next two sections consider each of these methods in turn.

3.1 Linking without an Import Library

Figure 3.1 illustrates the first way to supply definitions for dynamic-link function calls, in which each of the two files—the **.DLL** file and the **.EXE** file—has a corresponding module-definition file. (A module-definition file has a **.DEF** default extension.)

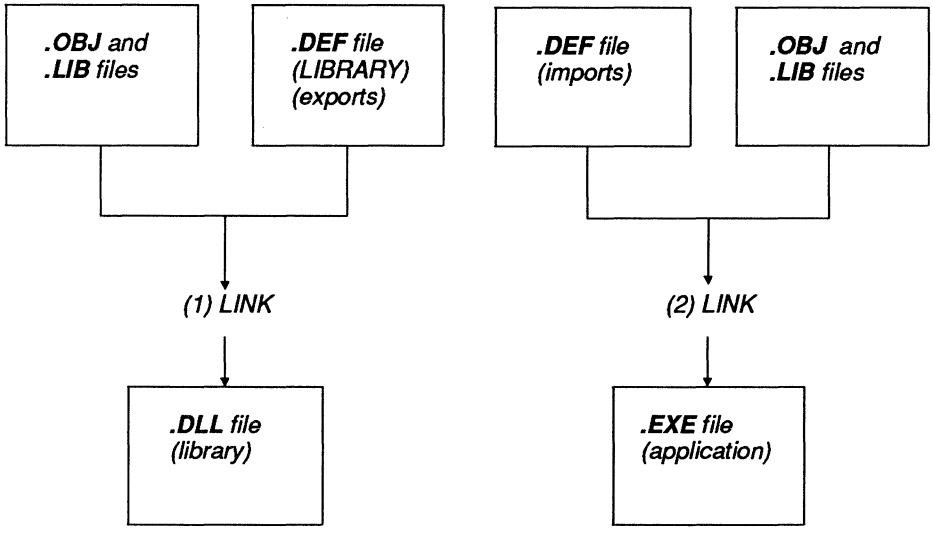


Figure 3.1 Linking without an Import Library

The two major steps may be described as follows:

1. Object files (and possibly standard-library files) are linked together with a module-definition file to create a dynamic-link library. A module-definition file for a dynamic-link library has at least two statements. The first is a **LIBRARY** statement, which directs the linker to create a **.DLL** rather than an **.EXE** file. The second statement is a list of export definitions.

2. Object files (and possibly standard-library files) are linked together with a module-definition file to create an application. The module-definition file for this application contains a list of import definitions. Each definition in this list contains both a function name and the name of a dynamic-link library.

The DOS 3.x linker has no way to accept a module-definition file as input. However, the dual-mode (OS/2) linker has an additional field for a module-definition file. This field is discussed in Section 4, "Using the OS/2 Linker."

3.2 Linking with an Import Library

Figure 3.2 illustrates the second way to supply definitions for dynamic-link function calls, in which a module-definition file is supplied for the dynamic-link library, and an import library is supplied for the application.

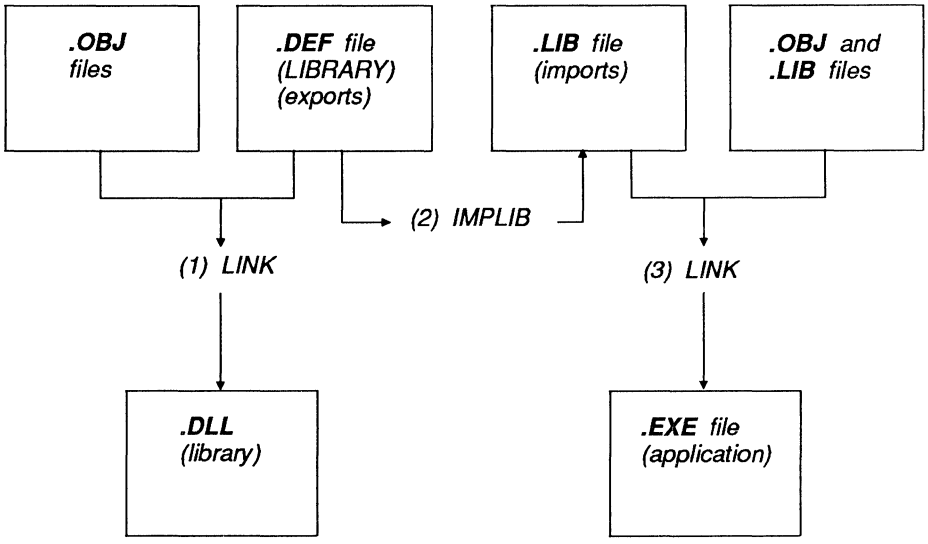


Figure 3.2 Linking with an Import Library

The three major steps may be explained as follows:

1. Object files are linked to produce a **.DLL** file. This step is identical to the first step in the previous section. Note that the module-definition file contains export definitions.

2. The **IMPLIB** utility is used to generate an import library. **IMPLIB** takes as input the same module-definition file used in the first step. **IMPLIB** knows the name of the library module (which by default has the same base name as the **.DEF** file), and it determines the name of each exported function by examining

export definitions. For each export definition in the **.DEF** file, **IMPLIB** generates a corresponding import definition.

3. The **.LIB** file generated by **IMPLIB** is used as input to **LINK**, which creates an application. This **.LIB** file does not use the same file format as a **.DEF** file, but it fulfills the same purpose: to provide the linker with information about imported dynamic-link functions.

The **.LIB** file generated by **IMPLIB** is called an import library. Import libraries are similar in most respects to ordinary libraries; you specify import libraries and ordinary libraries in the same command-line field of **LINK**, and you can append the two kinds of libraries together (by using the Library Manager). Furthermore, both kinds of libraries resolve external references at link time. The only difference is that import libraries do not contain executable code, merely records that describe where the executable code can be found at run time.

So far, only simple scenarios have been considered. Dynamic linking is flexible, and supports much more complicated scenarios. An application can make calls to more than one dynamic-link library. Furthermore, module-definition files for libraries can import functions as well as export them. It is perfectly possible for a **.DLL** file to call another **.DLL** file, and so on, to any level of complexity; the result may be a situation in which many files are loaded at run time.

3.3 Why Use Import Libraries?

At first glance, it may seem easier to create programs without import libraries, since import libraries add an extra step to the linking process. Usually, however, it is easier to use import libraries. There are two reasons why this is so.

First, the **IMPLIB** utility automates much of the program-creation process for you. To run **IMPLIB**, you specify the **.DEF** file that you already created for the dynamic-link library. Operation of **IMPLIB** is simple. If you do not use an import library generated by **IMPLIB**, then you must use an ASCII text editor to create a second **.DEF** file, where you explicitly give all needed import definitions.

Second, the first two steps in the linking process described above (creation of the **.DLL** file and creation of the import library) may be carried out only by the author of the dynamic-link library. The libraries may then be given to an applications programmer, who focuses on linking the application (third step). The application programmer's task is simplified if he links with the import library, because then he does not have to worry about editing his own **.DEF** file. The import library comes ready to link.

A good example of a useful import library is the file **DOSCALLS.LIB**. Protected-mode applications generally need to call one of the dynamic-link system libraries that are released with OS/2; the **DOSCALLS.LIB** file contains import definitions for all

calls to these system libraries. It is much easier to link with **DOSCALLS.LIB** than to create a **.DEF** file for every OS/2 program you link.

3.4 Advantages of Dynamic Linking

Why use dynamic-link libraries at all? Dynamic-link libraries serve much the same purpose that standard libraries do, but in addition, dynamic-link libraries give you the following advantages:

1. Link applications faster.

   With dynamic linking, the executable code for a dynamic-link function is not copied into the application's **.EXE** file. Instead, only an import definition is copied. Therefore, linking is usually a bit faster.

2. Save significant disk space.

   Suppose you create a library function called `printit`, and that this function is called by many different programs. If `printit` is in a standard library, then the function's executable code must be linked into each **.EXE** file that calls the function. In other words, the same code resides on your disk in many different files. But if `printit` is stored in a dynamic-link library, then the executable code resides in just one file—the library itself.

3. Make libraries and applications more independent.

   Dynamic-link libraries can be updated any number of times, without relinking the applications that use them. If you are a user of third-party libraries, this fact is particularly convenient. You receive the updated **.DLL** file from the third-party developers, and you need only copy the new library onto your disk. At run time, your applications will automatically call the updated library functions.

4. Utilize shared code and data segments.

   Code and data segments loaded in from a dynamic-link library can be shared. Without dynamic linking, this sharing is not possible because each file has its own copy of all the code and data it uses. By sharing segments with dynamic linking, you can utilize memory much more efficiently.

Section 4
Using the OS/2 Linker

This section describes how to link applications and dynamic-link libraries, and assumes that you are familiar with the concepts of dynamic linking, import libraries, and module-definition files. If you are not familiar with these concepts, then read the previous section, "About Linking in OS/2."

The linker can produce either an application that runs under DOS 3.x, an application that runs under OS/2 (or Microsoft Windows), or a dynamic-link library. The following rules determine what output the linker produces:

1. If no module-definition file or import library resolves any external references, then the linker produces an application for DOS 3.x (In other words, the linker creates a DOS 3.x application unless you specify a module-definition file or import library, and that file resolves at least one external reference.)

2. If a module-definition file with a **LIBRARY** statement is given, then the linker produces a dynamic-link library for OS/2.

3. Otherwise, the linker produces an application for OS/2.

You can therefore produce an OS/2 application by linking with an import library or a module-definition file, as long as you do not use a **LIBRARY** statement. (The **LIBRARY** statement is described in Section 7, "Using Module-Definition Files.") The file **DOSCALLS.LIB** is an import library. Thus, if you link with **DOSCALLS.LIB**, you produce either an OS/2 application or a dynamic-link library.

Note

Throughout this chapter, all references to OS/2 protected mode also apply to Microsoft Windows.

The linker produces files that run in protected mode only or in real mode only. However, OS/2 applications that make dynamic-link calls only to the Family API (a subset of the functions defined in **DOSCALLS.LIB**) can be made to run under DOS 3.x with the **BIND** utility. The **BIND** utility is discussed in the next section.

■ Syntax

Use the following command-line syntax to invoke the OS/2 linker:

LINK *objects* [[, [[*exe*]] [[, [[*map*]] [[, [[*lib*]] [[, *def*]]]]]]][[;]]

Each of the command-line fields is explained below. In the list that follows, reference is made to libraries. Unless qualified by the term "dynamic-link," the word "libraries" refers to import libraries and standard (object-code) libraries, both of which have the default extension .LIB. (Note that dynamic-link libraries have the default extension .DLL, and therefore are usually easy to tell from other libraries.) You can specify import libraries anywhere you can specify standard libraries. You can also combine import libraries and standard libraries by using the Library Manager; these combined libraries can then be specified in place of standard libraries.

Field	Description
objects	The name of one or more object-code files, to be linked into the application or dynamic-link library. Object files are output by compilers and assemblers. To specify more than one object file, separate each file name by a space or by the plus sign (+). Libraries can also be specified in this field, in which case they are considered "load libraries" by the linker. All objects in a load library (functions and data) are automatically linked into the linker's output.
exe	The name you wish the application or dynamic-link library to have. The default for an application name is the base name of the first object module on the command line, combined with an **.EXE** extension. The default for a dynamic-link-library name is the base name of the module-definition file, combined with a **.DLL** extension. Different defaults may be specified in the module-definition file.
map	The name you wish the map file to have.
libraries	The name of one or more library files, which **LINK** searches to resolve external references. You can also enter directories in this field; **LINK** searches the specified directories in order to find any libraries that it cannot find in the current directory. If you have more than one entry in this field, separate each entry by a space.

def	File name of a module-definition file. The use of a module-definition file is optional for applications, but required for dynamic-link libraries.

Note

The OS/2 linker supports overlays only when producing a real-mode application.

As with the DOS 3.x linker, you may specify command-line options after any field—but before the comma that terminates the field. The rest of this section discusses linker command-line options.

4.1 Options for Real Mode Only

Most of the options listed in Chapter 12 of the Microsoft CodeView and Utilities manual can be used with either protected-mode or real-mode programs. However, the following options can be used only when linking real-mode programs:

Option	Minimum Abbreviation
/CPARMAXALLOC	**/CP**
/DSALLOCATE	**/DS**
/HIGH	**/HI**
/NOGROUPASSOCIATION	**/NOG**
/OVERLAYINTERRUPT	**/O**

4.2 Options for Protected Mode Only

The OS/2 linker supports two new options that can be used only when linking protected-mode programs (or with Microsoft Windows applications). As mentioned above, most options described in Chapter 12 of the Microsoft CodeView and Utilities manual can be used for both protected-mode and real-mode programs.

■ **Syntax**

/A[[LIGNMENT]]:*size*

The **/ALIGNMENT** option directs **LINK** to align segment data in the executable file along the boundaries specified by *size*. The *size* argument must be a power of two. For example,

```
ALIGNMENT:16
```

indicates an alignment boundary of 16 bytes. The default alignment for OS/2-application and dynamic-link segments is 512. The minimum abbreviation for this option is **/A**.

■ **Syntax**

/W[[ARNFIXUP]]

The **/WARNFIXUP** option directs the linker to issue a warning for each segment-relative fixup of location-type "offset," such that the segment is contained within a group but is not at the beginning of the group. The linker will include the displacement of the segment from the group in determining the final value of the fixup, contrary to what happens with DOS executable files. The minimum abbreviation for this option is **/W**.

4.3 New Options for Both Modes

In addition to the options listed in Chapter 12 of the Microsoft CodeView and Utilities manual, the OS/2 linker also supports the following options for both real-mode and protected-mode programs. The **/NONULLSDOSSEG** option is primarily of interest to Windows programmers, as is the **/W** option described above.

■ **Syntax**

/NOE[[XTENDEDDICTSEARCH]]

The **/NOEXTENDEDDICTSEARCH** option prevents the linker from searching the extended dictionary, which is an internal list of symbol locations that the linker maintains. Normally, the linker consults this list to speed up library searches. The effect of the **/NOE** option is to slow down the linker. You often need to use this option when a library symbol is redefined. The linker issues error `L2044` if you need to use this option. The minimum abbreviation for this option is **/NOE**.

■ **Syntax**

/NON[[ULLSDOSSEG]]

The **/NONULLSDOSSEG** option directs the linker to arrange segments in the same order as they are arranged by the **/DOSSEG** option. The only difference is that the **/DOSSEG** option inserts 16 null bytes at the beginning of the **_TEXT** segment (if it is defined), whereas **/NONULLSDOSSEG** does not insert these extra bytes.

If the linker is given both the **/DOSSEG** and **/NONULLSDOSSEG** options, the **/NONULLSDOSSEG** option will always take precedence. Therefore you can use **/NONULLSDOSSEG** to override the **DOSSEG** comment record commonly found in run-time libraries. The minimum abbreviation for this option is **/NON**.

■ **Syntax**

INC[[REMENTAL]]

/PADC[[ODE]]:*bytes*

/PADD[[ATA]]:*bytes*

The last three options are explained in Section 9 below, "The ILINK Utility."

Section 5
The BIND Utility

The Microsoft Operating System/2 Bind utility (**BIND**) converts protected-mode programs so that they can run in real mode as well as protected mode. Not every protected-mode program can readily be converted. Programs you wish to convert should make no system calls other than calls to the functions listed in the Family API. (The Family API is a subset of the API functions and is summarized in the *Microsoft Operating System/2 Programmer's Reference.*)

The **BIND** utility must "bind" dynamic-link functions; that is, the utility brings an application program together with libraries, and links everything into a single stand-alone file which can run in real mode. The **BIND** utility also alters the executable-file format of the program, so that it is recognized as a standard executable file by both DOS 3.x and OS/2.

There are three components to the **BIND** utility:

Item	Description
BIND	This utility merges the executable file with the appropriate libraries as described above.
loader	This tool loads the OS/2 executable file when running DOS 2.x or 3.x and simulates the OS/2 startup conditions in an environment. The loader consists of code that is stored in **BIND.EXE**, and then copied into files as needed.
API.LIB	This library simulates the OS/2 API in an environment.

5.1 Binding Libraries

The **BIND** utility replaces Family-API calls with simulator routines from the standard (object-code) library **API.LIB**. However, your program may also make dynamic-link calls to functions outside the API (that is, you can make dynamic-link calls that are not system calls). This section explains how **BIND** can accommodate these calls.

If your program makes dynamic-link calls to functions outside the API, use the *linklibs* field described in Section 5.3, "The BIND Command Line." **BIND** searches each of the *linklibs* for object code corresponding to the imported functions. In addition, if you

are using import definitions with either the *ordinal* or the *internalname* option, you will need to specify import libraries so that the functions you call can be identified correctly. (For a discussion of various options within import definitions, see Section 7, "Using Module-Definition Files.")

5.2 Binding Functions as Protected Mode Only

If your program freely makes non-Family-API calls without regard to which operating system is in use, then the program cannot be converted for use in real mode. However, you may choose to write a program so that it first checks the operating system, and then restricts system calls (to the Family API) when running in real mode. The **BIND** utility supports conversion of these programs.

By using the **/n** command-line option, described below, you can specify a list of functions supported in protected mode only. If your program ever attempts to call one of these functions when running in real mode, then the **BadDynLink** system function is called and aborts your program. The advantage of this option is that it helps resolve external references. Yet it remains the responsibility of your program to check the operating-system version, and ensure that not one of these functions is ever called in real mode.

If your program makes calls (either directly or indirectly) to non-Family-API system calls, but you do *not* use the **/n** option, then **BIND** will fail to convert your program.

5.3 The BIND Command Line

Invoke **BIND** with the following command line:

BIND *infile* [[*implibs*]] [[*linklibs*]] [[**/o** *outfile*]] [[**/n @** *file*]] [[**/n** *names*]] [[**/m** *mapfile*]]

The meaning of each command-line field and option is explained below:

The *infile* field contains the name of the OS/2 application. The file name may contain a complete path name. The file extension is optional; if you provide no extension, then **.EXE** is assumed.

The *implibs* field contains the name of one of more import libraries. As explained above, use this field if your program uses an import definition with either the *ordinal* or *internalname* fields.

Note

If you want to specify a 64-kilobyte (K) default data segment when running in real mode, then specify the file **APILMR.OBJ**, which guarantees a 64K stack. The reason this object file may be necessary is that a protected-mode application is not automatically given a 64K default data segment; a protected-mode application is only allocated the space it specifically requests. If you do not specify the file **APILMR.OBJ**, then you may not have the local heap area you need when you run in real mode.

The *linklibs* field contains the name of one or more standard libraries. Use this field to supply object code needed to resolve dynamic-link calls. If this field is empty, then the library **API.LIB** is automatically included. However, if you specify any libraries, then **API.LIB** is not assumed, and you need to give a complete path name for each library you specify.

The *outfile* is the name of the bound application, and may contain a full path name. The default value of this field is *infile*. (Whatever name is used for the *infile* field also becomes the default for *outfile*.)

The **/n** option provides a way of listing functions that are supported in protected mode only. As explained above, if any of these functions are ever called in real mode, then the **BadDynLink** function will be called to abort the program. The **/n** option can be used either with a list of one or more *names* (separated by spaces), or with a *file* preceded by the **@** sign. The *file* should consist of a list of functions, one per line.

The **/m** option causes a link map to be generated for the DOS 3.x environment of the **.EXE** file. The *mapfile* is the destination of the link map. If no *mapfile* is specified with the **/m** option, then the destination of the link map is standard output.

5.4 BIND Operation

BIND produces a single executable file, which can run on either OS/2 or DOS 3.x. To complete this task, **BIND** executes three major steps:

1. Reads in the dynamic-link entry points (for imported functions) from the OS/2 executable file and outputs to a temporary object file the **EXTDEF** object records for each imported item. Each **EXTDEF** record tells the linker of an external reference that needs to be resolved through ordinary linking.

2. Invokes the linker, giving the executable file, the temporary object file, the **API.LIB** file, and any other libraries specified on the **BIND** command line. The linker produces an executable file which can run in real mode, by linking in the loader and the API-simulator routines.

3. Merges the protected-mode and real-mode executable files, to produce a single file which can run in either mode.

5.5 Executable-File Layout

OS/2 executable files have two headers. The first header has a DOS 3.x format. The second header has the OS/2 format. When the executable file is run on an OS/2 system, it ignores the first header and uses the OS/2 format. When run under DOS 3.x, the old header is used to load the file. Figure 5.1 shows the arrangement of the merged headers.

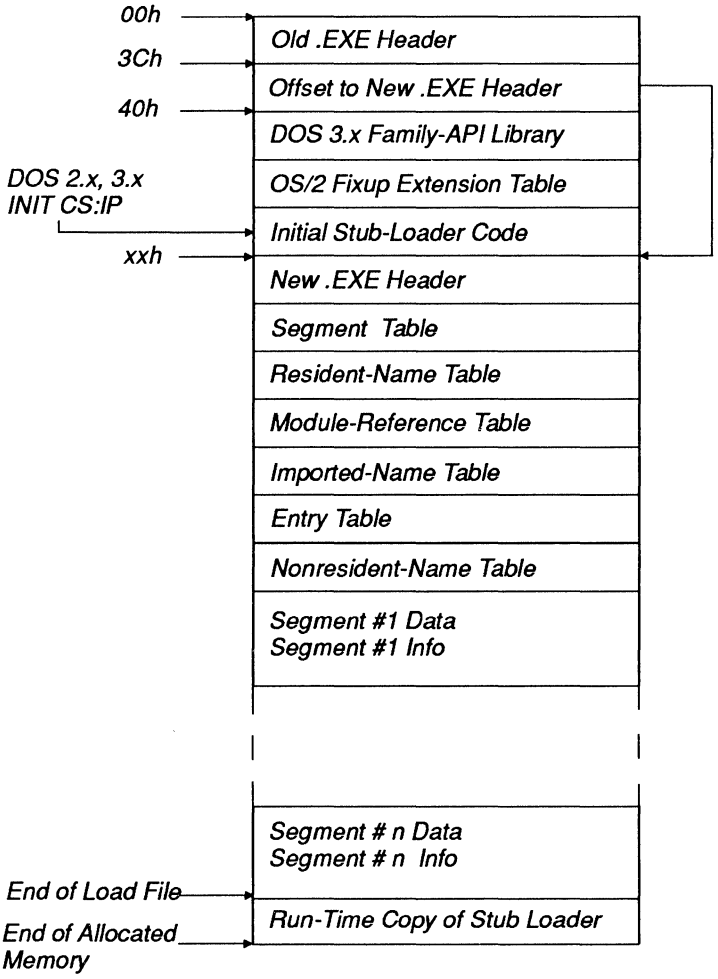


Figure 5.1 OS/2 Executable-File Header

Section 6
The IMPLIB Utility

This section summarizes the use of the Microsoft Import Library Manager utility (**IMPLIB**), and assumes you are familiar with the concepts of import libraries, dynamic linking, and module-definition files. If you are not familiar with these concepts, read Section 3, "About Linking in OS/2."

You can create an import library for use by other programmers in resolving external references to your dynamic-link library. The **IMPLIB** command creates an import library, which is a file with a **.LIB** extension that can be read by the OS/2 linker. The **.LIB** file can be specified in the **LINK** command line with other libraries. Import libraries are recommended for all dynamic-link libraries. Without the use of import libraries, external references to dynamic-link routines must be declared in an **IMPORTS** statement in the module-definition file for the application being linked. **IMPLIB** is supported only in protected mode.

■ Syntax

IMPLIB *implibname mod-def-file* [[*mod-def-file...*]]

The *implibname* is the name you wish the new import library to have.

The *mod-def-file* is the name of a module-definition file for the dynamic-link module. You may enter more than one.

■ Example

The following command creates the import library named **MYLIB.LIB** from the module-definition file **MYLIB.DEF**:

```
IMPLIB mylib.lib mylib.def
```

Section 7
Using Module-Definition Files

A module-definition file describes the name, attributes, exports, imports, and other characteristics of an application or library for OS/2 or Microsoft Windows. This file is required for Windows applications and libraries, and is also required for dynamic-link libraries that run under OS/2.

A module-definition file contains one or more "module statements." Each module statement defines an attribute of the executable file, such as its module name, the attributes of program segments, and the number and names of exported and imported functions. The module statements and the attributes they define are listed as follows:

Statement	Attribute
NAME	Names application (no library created)
LIBRARY	Names dynamic-link library (no application created)
DESCRIPTION	Describes the module in one line
CODE	Gives default attributes for code segments
DATA	Gives default attributes for data segments
SEGMENTS	Gives attributes for specific segments
STACKSIZE	Specifies local-stack size, in bytes
EXPORTS	Defines exported functions
IMPORTS	Defines imported functions
STUB	Adds a DOS 3.x executable file to the beginning of the module, usually to terminate the program when run in real mode
HEAPSIZE	Specifies local-heap size, in bytes
PROTMODE	Specifies that the module runs only in DOS protected mode
OLD	Preserves import information from a previous version of the library

REALMODE	Relaxes some restrictions that the linker imposes for protected-mode programs
EXETYPE	Identifies operating system

The following rules govern the use of these statements in a module-definition file:

1. If you use either a **NAME** or a **LIBRARY** statement, it must precede all other statements in the module-definition file.

2. You can include source-level comments in the module-definition file, by beginning a line with a semicolon (;). The OS/2 utilities ignore each such comment line.

3. Module-definition keywords (such as **NAME**, **LIBRARY**, and **SEGMENTS**) must be entered in uppercase letters.

The following sample module-definition file gives module definitions for a dynamic-link library. This sample file includes one source-level comment and five statements.

```
; Sample module-definition file

LIBRARY

DESCRIPTION 'Sample .DEF file for a dynamic-link library'

CODE        PRELOAD

STACKSIZE   1024

EXPORTS
    Init    @1
    Begin   @2
    Finish  @3
    Load    @4
    Print   @5
```

The meaning of each of these fields is explained in the sections that follow, which describe module-definition statements, and give syntax and examples.

7.1 The NAME Statement

The **NAME** statement identifies the executable file as an application and optionally defines the name.

■ Syntax

NAME[[*appname*]] [[*apptype*]]

■ Remarks

If an *appname* is given, it becomes the name of the application as it is known by OS/2. This name can be any valid file name. If no *appname* is given, the name of the executable file—with the extension removed—becomes the name of the application.

The *apptype* field is used by a future version of OS/2, and should be declared for compatibility with this future version.

If *apptype* is given, it defines the type of application being linked. This information is kept in the executable-file header. You do not need to use this field unless you may be using your application in a Windows environment. The *apptype* field may have one of the following values:

Keyword	Meaning
WINDOWAPI	Real-mode Windows application. The application uses the API provided by Windows and must be executed in the Windows environment.
WINDOWCOMPAT	Windows-compatible application. The application can run inside Windows, or it can run in a separate screen group. An application can be of this type if it uses the proper subset of OS/2 video, keyboard, and mouse functions which are supported in Windows applications.
NOTWINDOWCOMPAT	Application is not Windows compatible and must operate in a separate screen group from Windows.

If the **NAME** statement is included in the module-definition file, then the **LIBRARY** statement cannot appear. If neither a **NAME** statement nor a **LIBRARY** statement appears in a module-definition file, the default is **NAME**; that is, the linker acts as though a **NAME** statement were included, and thus creates an application rather than a library.

■ Example

The following example assigns the name `calendar` to the application being defined:

```
NAME calendar WINDOWCOMPAT
```

7.2 The LIBRARY Statement

The **LIBRARY** statement identifies the executable file as a dynamic-link library, and it can specify the name of the library or the type of library-module initialization required.

■ Syntax

LIBRARY [[*libraryname*]] [[*initialization*]]

■ Remarks

If a *libraryname* is given, it becomes the name of the library as it is known by OS/2. This name can be any valid file name. If no *libraryname* is given, the name of the executable file—with the extension removed—becomes the name of the library.

The *initialization* field is optional and can have one of the two values listed below. If neither is given, then the *initialization* default is **INITGLOBAL**.

Keyword	Meaning
INITGLOBAL	The library-initialization routine is called only when the library module is initially loaded into memory.
INITINSTANCE	The library-initialization routine is called each time a new process gains access to the library.

If the **LIBRARY** statement is included in a module-definition file, then the **NAME** statement cannot appear. If no **LIBRARY** statement appears in a module-definition file, the linker assumes that the module-definition file is defining an application.

■ Example

The following example assigns the name `calendar` to the dynamic-link module being defined, and specifies that library initialization is performed each time a new process gains access to `calendar`.

```
LIBRARY calendar INITINSTANCE
```

7.3 The DESCRIPTION Statement

The **DESCRIPTION** statement inserts the specified *text* into the application or library. This statement is useful for embedding source-control or copyright information into an application or library.

■ Syntax

DESCRIPTION *'text'*

■ Remarks

The *text* is a one-line string enclosed in single quotation marks. Use of the **DESCRIPTION** statement is different from the inclusion of a comment, because comments—lines that begin with a semicolon (;)—are not placed in the application or library.

■ Example

The following example inserts the text `Template Program` into the application or library being defined:

```
DESCRIPTION 'Template Program'
```

7.4 The CODE Statement

The **CODE** statement defines the default attributes for code segments within the application or library.

■ Syntax

CODE[[*attribute...*]]

■ Remarks

Each *attribute* must correspond to one of the following attribute fields. Each field can appear at most one time, and order is not significant. The attribute fields are presented below, along with legal values. In each case, the default value is listed last. The last

three fields have no effect on OS/2 code segments and are included for use with Microsoft Windows.

Field	Values
load	**PRELOAD, LOADONCALL**
executeonly	**EXECUTEONLY, EXECUTEREAD**
iopl	**IOPL, NOIOPL**
conforming	**CONFORMING, NONCONFORMING**
shared	**SHARED, NONSHARED**
movable	**MOVABLE, FIXED**
discard	**NONDISCARDABLE, DISCARDABLE**

The *load* field determines when a code segment is to be loaded. This field contains one of the following keywords:

Keyword	Meaning
PRELOAD	The segment is loaded automatically, at the beginning of the program.
LOADONCALL	The segment is not loaded until accessed (the default).

The *executeonly* field determines whether a code segment can be read as well as executed. This field contains one of the following keywords:

Keyword	Meaning
EXECUTEONLY	The segment can only be executed.
EXECUTEREAD	The segment can be both executed and read (the default).

The *iopl* field determines whether or not a segment has I/O privilege (that is, whether it can access the hardware directly). This field contains one of the following keywords:

Keyword	Meaning
IOPL	The code segment has I/O privilege.
NOIOPL	The code segment does not have I/O privilege (the default).

The *conforming* field specifies whether or not a code segment is a 286 "conforming" segment. The concept of a conforming segment deals with privilege level (the range of

instructions that the process can execute) and is relevant only to users writing device drivers and system-level code. A conforming segment can be called from either Ring 2 or Ring 3, and the segment executes at the caller's privilege level. This field contains one of the following keywords:

Keyword	Meaning
CONFORMING	The segment is conforming.
NONCONFORMING	The segment is nonconforming (the default).

The *shared* field determines whether all instances of the program can share a given code segment. This field is ignored by OS/2, but is provided for use with real-mode Windows. Under OS/2, all code segments are shared. The *shared* field contains one of the following keywords: **SHARED** or **NONSHARED** (the default).

The *movable* field determines whether a segment can be moved around in memory. This field is ignored by OS/2, but is provided for use with real-mode Windows. Under OS/2, all segments are movable. The *movable* field contains one of the following keywords: **MOVABLE** or **FIXED** (the default for Windows).

The *discard* field determines whether a segment can be swapped out to disk by the operating system when not currently needed. This attribute is ignored by OS/2, but is provided for use with real-mode Windows. Under OS/2 systems, all segments can be swapped as needed. The *shared* attribute contains one of the following keywords: **DISCARDABLE** or **NONDISCARDABLE** (the default for Windows).

■ Example

The following example sets defaults for the module's code segments, so that they are not loaded until accessed and so that they have I/O hardware privilege:

```
CODE LOADONCALL IOPL
```

7.5 The DATA Statement

The **DATA** statement defines the default attributes for the data segments within the application or module.

■ Syntax

DATA [[*attribute...*]]

■ Remarks

Each *attribute* must correspond to one of the following attribute fields. Each field can appear at most one time, and order is not significant. The attribute fields are present below, along with legal values. In each case, the default value is listed last. The last two fields have no effect on OS/2 data segments, but are included for use with Microsoft Windows.

Field	Values
load	**PRELOAD, LOADONCALL**
readonly	**READONLY, READWRITE**
instance	**NONE, SINGLE, MULTIPLE**
iopl	**IOPL, NOIOPL**
shared	**SHARED, NONSHARED**
movable	**MOVABLE, FIXED**
discard	**DISCARDABLE, NONDISCARDABLE**

The *load* field determines when a segment will be loaded. This field contains one of the following keywords:

Keyword	Meaning
PRELOAD	The segment is loaded when the program begins execution.
LOADONCALL	The segment is not loaded until it is accessed (the default).

The *readonly* field determines the access rights to a data segment. This field contains one of the following keywords:

Keyword	Meaning
READONLY	The segment can only be read.
READWRITE	The segment can be both read and written to (the default).

The *instance* field affects the sharing attributes of the automatic data segment, which is the physical segment represented by the group name **DGROUP**. (This segment group makes up the physical segment which contains the local stack and heap of the application.) The *instance* field contains one of the following keywords:

Keyword	Meaning
NONE	No automatic data segment is created.
SINGLE	A single automatic data segment is shared by all instances of the module. In this case, the module is said to have "solo" data. This keyword is the default for dynamic-link libraries.
MULTIPLE	The automatic data segment is copied for each instance of the module. In this case, the module is said to have "instance" data. This keyword is the default for applications.

The *iopl* field determines whether or not data segments have I/O privilege (that is, whether or not they can access the hardware directly). This field contains one of the following keywords:

Keyword	Meaning
IOPL	The data segments have I/O privilege.
NOIOPL	The data segments do not have I/O privilege (the default).

The *shared* field determines whether all instances of the program can share a **READWRITE** data segment. Under OS/2, this field is ignored by the linker if the segment has the attribute **READONLY**, since **READONLY** data segments are always shared. The *shared* field contains one of the following keywords:

Keyword	Meaning
SHARED	One copy of the data segment will be loaded and shared among all processes accessing the module.
NONSHARED	The segment cannot be shared, and must be loaded separately for each process (the default).

The *movable* field determines whether a segment can be moved around in memory. This field is ignored by OS/2, but is provided for use with real-mode Windows. Under OS/2, all segments are movable. The *movable* field contains one of the following keywords: **MOVABLE** or **FIXED** (the default for Windows).

The optional *discard* field determines whether a segment can be swapped out to disk by the operating system, when not currently needed. This attribute is ignored by OS/2, but is provided for use with real-mode Windows. Under OS/2 systems, all segments can be swapped as needed. The *discard* attribute contains one of the following keywords: **DISCARDABLE** or **NONDISCARDABLE** (the default for Windows).

Note

The linker makes the automatic data segment attribute (specified by an *instance* value of **SINGLE** or **MULTIPLE**) match the sharing attribute of the automatic data segment (specified by a *shared* value of **SHARED** or **NONSHARED**). Solo data (specified by **SINGLE**) force shared data segments by default. Instance data (specified by **MULTIPLE**) force nonshared data by default. Similarly, **SHARED** forces solo data, and **NONSHARED** forces instance data.

If you give a contradictory **DATA** statement (e.g., `DATA SINGLE NONSHARED`), all segments in **DGROUP** are shared, and all other data segments are nonshared by default. If a segment that is a member of **DGROUP** is defined with a *sharing* attribute that conflicts with the automatic data type, a warning about the bad segment is issued, and the segment's flags are converted to a consistent sharing attribute. For example, the following

```
DATA SINGLE
SEGMENTS
_DATA CLASS 'DATA' NONSHARED
```

is converted to

```
_DATA CLASS 'DATA' SHARED
```

■ **Example**

The following example defines the application's data segment so that it is loaded only when it is accessed and so that it cannot be shared by more than one copy of the program:

```
DATA LOADONCALL NONSHARED
```

By default, the data segment can be read and written, the automatic data segment is copied for each instance of the module, and the data segment has no I/O privilege.

7.6 The SEGMENTS Statement

The **SEGMENTS** statement defines the attributes of one or more segments in the application or library on a segment-by-segment basis. The attributes specified by this statement override defaults set in **CODE** and **DATA** statements.

■ Syntax

SEGMENTS
segmentdefinitions

■ Remarks

The **SEGMENTS** keyword marks the beginning of the segment definitions. This keyword can be followed by one or more segment definitions, each on a separate line (limited by the number set by the linker's **/SEGMENTS** option, or 128 if the option is not used). The syntax for each segment definition is as follows:

segmentname [[**CLASS** *'classname'*]][[*attribute...*]]

Each segment definition begins with a *segmentname*, which can be placed in optional single quotation marks (**'**). The quotation marks are required if *segmentname* conflicts with a module-definition keyword, such as **CODE** or **DATA**.

The **CLASS** keyword specifies the class of the segment. The single quotation marks (**'**) are required around *classname*. If you do not use the **CLASS** argument, the linker assumes that the class is **CODE**.

Each *attribute* must correspond to one of the following attribute fields. Each field can appear at most one time, and order is not significant. The attribute fields are presented below, along with legal values. In each case, the default value is listed last.

Field	Values
load	**PRELOAD, LOADONCALL**
readonly	**READONLY, READWRITE**
executeonly	**EXECUTEONLY, EXECUTEREAD**
iopl	**IOPL, NOIPL**
conforming	**CONFORMING, NONCONFORMING**
shared	**SHARED, NONSHARED**
movable	**MOVABLE, FIXED**
discard	**DISCARDABLE, NONDISCARDABLE**

The *load* field determines when a segment is to be loaded. This field contains one of the following keywords:

Keyword	Meaning
PRELOAD	The segment is loaded automatically, at the beginning of the program.
LOADONCALL	The segment is not loaded until accessed (the default).

The *readonly* field determines the access rights to a data segment. This field contains one of the following keywords:

Keyword	Meaning
READONLY	The segment can be read only.
READWRITE	The segment can be both read and written to (the default).

The *executeonly* field determines whether a code segment can be read as well as executed. (The attribute has no effect on data segments.) This field contains one of the following keywords:

Keyword	Meaning
EXECUTEONLY	The segment can only be executed.
EXECUTEREAD	The segment can be both executed and read (the default).

The *iopl* field determines whether or not a segment has I/O privilege (that is, whether it can access the hardware directly). This field contains one of the following keywords:

Keyword	Meaning
IOPL	The segments have I/O privilege.
NOIOPL	The segments do not have I/O privilege (the default).

The *conforming* field specifies whether or not a code segment is a 286 "conforming" segment. The concept of a conforming segment deals with privilege level (the range of instructions that the process can execute) and is relevant only to users writing device drivers and system-level code. A conforming segment can be called from either Ring 2 or Ring 3, and the segment executes at the caller's privilege level. (The attribute has no effect on data segments.) This field contains one of the following keywords:

Keyword	Meaning
CONFORMING	The segment is conforming.
NONCONFORMING	The segment is nonconforming (the default).

The *shared* field determines whether all instances of the program can share a **READ-WRITE** segment. For code segments and **READONLY** data segments, this field is ignored by OS/2, but is provided for use with real-mode Windows. Under OS/2, all code segments and all **READONLY** data segments are shared. The *shared* field contains one of the following keywords: **SHARED** or **NONSHARED** (the default).

The *movable* field determines whether a segment can be moved around in memory. This field is ignored by OS/2, but is provided for use with real-mode Windows. Under OS/2, all segments are movable. The *movable* field contains one of the following keywords: **MOVABLE** or **FIXED** (the default for Windows).

The optional *discard* field determines whether a segment can be swapped out to disk by the operating system, when not currently needed. This attribute is ignored by OS/2, but is provided for use with real-mode Windows. Under OS/2 systems, all segments can be swapped as needed. The *shared* attribute contains one of the following keywords: **DISCARDABLE** or **NONDISCARDABLE** (the default for Windows).

■ Example

The following example specifies segments named `cseg1`, `cseg2`, and `dseg`. The first segment is assigned class `mycode` and the second is assigned **CODE**. Each segment is given different attributes.

```
SEGMENTS
    cseg1 CLASS 'mycode' IOPL
    cseg2 EXECUTEONLY PRELOAD CONFORMING
    dseg  CLASS 'data' LOADONCALL READONLY
```

7.7 The STACKSIZE Statement

The **STACKSIZE** statement performs the same function as the **/STACKSIZE** linker option. It overrides the size of any stack segment defined in an application. (The **STACKSIZE** statement overrides the **/STACKSIZE** option).

■ Syntax

STACKSIZE *number*

■ Remarks

The *number* must be an integer. The *number* is considered to be in decimal format by default, but you can use C notation to specify hexadecimal or octal.

■ Example

The following example allocates 4096 bytes of local-stack space:

```
STACKSIZE 4096
```

7.8 The EXPORTS Statement

The **EXPORTS** statement defines the names and attributes of the functions exported to other modules, and of the functions that run with I/O privilege. The term "export" refers to the process of making a function available to other run-time modules. By default, functions are hidden from other modules at run time.

■ Syntax

EXPORTS
exportdefinitions

■ Remarks

The **EXPORTS** keyword marks the beginning of the export definitions. It may be followed by up to 3072 export definitions, each on a separate line. You need to give an export definition for each dynamic-link routine that you want to make available to other modules. The syntax for an export definition is as follows:

entryname[[=*internalname*]] [[**@ord**[[**RESIDENTNAME**]]] [[*pwords*]] [[**NODATA**]]

The *entryname* specification defines the function name as it is known to other modules. The optional *internalname* defines the actual name of the export function as it appears within the module itself; by default, this name is the same as *entryname*.

The optional *ord* field defines the function's ordinal position within the module-definition table. If this field is used, then the function's entry point can be invoked by name or by ordinal. Use of ordinal positions is faster and may save space.

The optional keyword **RESIDENTNAME** specifies that the function's name be kept resident in memory at all times. This keyword is applicable only if the *ord* option is used, because if the *ord* option is not used, OS/2 automatically keeps the names of all exported functions resident in memory anyway.

The *pwords* field specifies the total size of the function's parameters, as measured in words (the total number of bytes divided by two). This field is required only if the function executes with I/O privilege. When a function with I/O privilege is called,

OS/2 consults the *pwords* field to determine how many words to copy from the caller's stack to the I/O-privileged function's stack.

The optional keyword **NODATA** is ignored by OS/2, but is provided for use by real-mode Windows.

Normally, the **EXPORTS** statement is only meaningful for functions within dynamic-link libraries, and for functions which execute with I/O privilege.

■ Example

The following **EXPORTS** statement defines three export functions: `SampleRead`, `StringIn`, and `CharTest`. The first two functions can be accessed either by their exported names or by an ordinal number. Note that in the module's own source code, these functions are actually defined as `read2bin` and `str1`, respectively. The last function runs with I/O privilege, and therefore is given with the total size of the parameters: six words.

```
EXPORTS
       SampleRead = read2bin  @8
       StringIn = str1        @4  RESIDENTNAME
       CharTest   6
```

7.9 The IMPORTS Statement

The **IMPORTS** statement defines the names of the functions that will be imported for the application or library. The term "import" refers to the process of declaring that a symbol is defined in another run-time module (a dynamic-link library). Typically, **LINK** uses an import library (created by the **IMPLIB** utility) to resolve external references to dynamic-link symbols. However, the **IMPORTS** statement provides an alternative for resolving these references within a module.

■ Syntax

IMPORTS
importdefinitions

■ Remarks

The **IMPORTS** keyword marks the beginning of the import definitions. This keyword is followed by one or more import definitions, each on a separate line. The only limit on the number of import definitions is that the total amount of space required for their

names must be less than 64K. Each import definition corresponds to a particular function. The syntax for an import definition is as follows:

[[*internalname*=]]*modulename.entry*

The *internalname* specifies the name that the importing module actually uses to call the function. Thus, *internalname* will appear in the source code of the importing module, though the function may have a different name in the module where it is defined. By default, *internalname* is the same as the name given in *entry*.

The *modulename* is the name of the application or library that contains the function.

The *entry* field determines the function to be imported, and can be a name or an ordinal value. (Ordinal values are set in an **EXPORTS** statement.) If an ordinal value is given, then the *internalname* field is required.

Note

A given function has a name for each of three different contexts. The function has a name used by the exporting module (where it is defined), a name used as an entry point between modules, and a name as it is used by the importing module (where it is called). If neither module uses the optional *internalname* field, then the function will have the same name in all three contexts. If either of the modules use the *internalname* field, then the function may have more than one distinct name.

■ Example

The following **IMPORTS** statement defines three functions to be imported: `SampleRead`, `SampleWrite`, and a function that has been assigned an ordinal value of 1. The functions are found in the modules `Sample`, `SampleA`, and `Read`, respectively. The function from the `Read` module is referred to as `ReadChar` in the importing module; the original name of the function, as it is defined in the `Read` module, may or may not be known.

```
IMPORTS
    Sample.SampleRead
    SampleA.SampleWrite
    ReadChar = Read.1
```

7.10 The STUB Statement

The **STUB** statement adds *filename*, a DOS 3.x executable file, to the beginning of the application or library being created. The stub will be invoked whenever the module is executed under DOS 2.x or DOS 3.x. Typically, the stub displays a message and terminates execution. (By default, the linker adds its own standard stub for this purpose.)

■ Syntax

STUB *'filename'*

■ Remarks

If the linker does not find this file in the current directory, it searches in the list of directories specified in the **PATH** environment variable.

■ Example

The following example appends the DOS executable file **STOPIT.EXE** to the beginning of the module:

```
STUB 'STOPIT.EXE'
```

The file **STOPIT.EXE** is executed when you attempt to run the module under DOS.

7.11 The HEAPSIZE Statement

The **HEAPSIZE** statement defines the size of the application's local heap, in bytes. This value affects the size of the automatic data segment.

■ Syntax

HEAPSIZE *bytes*

■ Remarks

The *bytes* field is an integer number, which is considered decimal by default. However, hexadecimal and octal numbers can be entered by using C notation.

■ **Example**

```
HEAPSIZE 4000
```

7.12 The PROTMODE Statement

The **PROTMODE** statement specifies that the module will run only in protected mode and not in Windows or dual mode. This statement is always optional, and permits a protected-mode-only application to omit some information from the executable-file header.

■ **Syntax**

PROTMODE

■ **Remarks**

If this statement is not included in the module-definition file, the linker assumes that the application can be run in either real or protected mode.

7.13 The OLD Statement

The **OLD** statement directs the linker to search another dynamic-link module for export ordinals. For more information on ordinals, consult the sections above on the **EXPORTS** and **IMPORTS** statements. Exported names in the current module that match exported names in the **OLD** module are assigned ordinal values from that module unless one of the following conditions is in effect: the name in the **OLD** module has no ordinal value assigned, or an ordinal value is explicitly assigned in the current module.

■ **Syntax**

OLD *'filename'*

■ **Remarks**

This statement is useful for preserving export ordinal values, throughout successive versions of a dynamic-link module. The **OLD** has no effect on application modules.

7.14 The REALMODE Statement

The **REALMODE** statement is analogous to the **PROTMODE** statement, and is provided for use with real-mode Windows applications.

■ Syntax

REALMODE

■ Remarks

REALMODE specifies that the application runs only in real mode. With this statement, the linker relaxes some of the restrictions that it imposes on programs running in protected mode.

7.15 The EXETYPE Statement

The **EXETYPE** statement specifies in which operating system the application (or dynamic-link library) is to run. This statement is optional and provides an additional degree of protection against the program being run in an incorrect operating system.

■ Syntax

EXETYPE [[OS2 | WINDOWS | DOS4]]

■ Remarks

The **EXETYPE** keyword must be followed by a descriptor of the operating system, either **OS2** (for OS/2 applications and dynamic-link libraries), **WINDOWS**, or **DOS4**. If no **EXETYPE** statement is given, then **EXETYPE OS2** is assumed by an operating system that is loading the program.

The effect of **EXETYPE** is simply to set bits in the header which identify operating system type. Operating system loaders may check these bits.

Section 8
Using the /X Option with MAKE

In addition to the options listed in Section 14.5 of the Microsoft CodeView and Utilities manual, "Specifying MAKE Options," the version of the Microsoft Program Maintenance Utility (**MAKE**) that accompanies Microsoft OS/2 has an additional option, which redirects error output. This option is particularly valuable if you run **MAKE** from a batch file, and you want to collect any error messages that occur.

■ Syntax

/X *file*

When you specify the **/X** option on the **MAKE** command line, then the **MAKE** utility will send all error output to *file*, which can be either a file or device. If **MAKE** cannot redirect output to *file*, then it will issue the following fatal error message:

```
U1015: file : error redirection failed
```

For example, **MAKE** issues the message shown above when you try to redirect error output to a read-only file on a DOS 3.x network.

In the discussion above, "error output" is defined as output which is written to standard error output. The file handle for standard error output is usually abbreviated as **stderr** in C programs.

By default, **MAKE** error messages are always sent to **stderr**.

Section 9
The ILINK Utility

The Microsoft Incremental Linker (**ILINK**) is a utility that can enable you to link your OS/2 or Windows application much faster. (It cannot work with DOS applications other than Windows.) You can benefit from its use when you change a small subset of the modules used to link a program. The program can use any memory model, but **ILINK** is most effective with large- and medium-memory-model programs. Furthermore, to benefit from **ILINK** you need to follow certain restrictions that are described in this chapter. Should **ILINK** fail to link your changes into the executable file, it will automatically invoke the full linker, **LINK.** You must first run the full linker with certain new options, described below, before you can use **ILINK.**

Note

You can use **ILINK** to develop dynamic-link libraries as well as applications. Everything said in this chapter about applications and executable files applies to dynamic-link libraries as well. This chapter uses the term "library" to refer specifically to an object-code library (a **.LIB** file).

This chapter covers the following topics:

- Definitions
- Guidelines for using **ILINK**
- The development process
- Running **ILINK**
- How **ILINK** works
- Incremental violations

9.1 Definitions

Incremental linking involves certain specialized concepts. You may need to review the following list of terms in order to understand the rest of this chapter:

Term	Meaning
segment	A contiguous area of memory up to 64K in size. See the definitions of "physical segment" and "logical segment" below.
module	A unit of code or data defined by one source file. In BASIC, Pascal, and large-memory-model C and FORTRAN programs, each module corresponds to a different segment. In small-memory-model programs, all code modules contribute to one code segment, and all data modules contribute to one data segment.
memory model	The memory model determines the number of code and data segments in a program. BASIC programs are always large memory model.
physical segment	A segment listed in the executable file's segment table. Each physical segment has a distinct segment address, whereas logical segments may share a segment address. A physical segment usually contains one logical segment, but it can contain more than one.
logical segment	A segment defined in an object module. Each physical segment other than **DGROUP** contains exactly one logical segment, except when you use the **GROUP** directive in a **MASM** module. (Linking with the **/PACKCODE** option can also create more than one logical segment per physical segment.)
code symbol	The address of a function, subroutine, or procedure.
data symbol	The address of a global or static data object. The concept of data symbol includes all data objects except local (stack-allocated) or dynamically allocated data.

9.2 Guidelines for Using ILINK

The incremental linker, **ILINK**, works much faster than the full linker because **ILINK** replaces only those modules which have changed since the last linking. It avoids much of the work done by **LINK**.

To enable incremental linking, you need to follow four major guidelines. If your changes exceed the scope allowed by these guidelines, then a full link is necessary.

1. Do not alter any **.LIB** files that you are using to create the executable file.

2. Put padding at the end of data and small-memory-model code modules, by using the **/PADCODE** and **/PADDATA** options presented in Section 9.3, "The Development Process."

   By putting padding at the end of a module, you enable the module to grow without forcing a full relinking. However, if the module is the only module contributing to its physical segment, then padding is not necessary.

   In practice this means that you can avoid padding if you have a BASIC, Pascal, FORTRAN, or C program (other than a small-memory-model C program), you do not call a **MASM** module that uses the **GROUP** directive, and you do not add to the size of the default data segment; consult your language documentation for information about what is placed in this area.

3. Do not delete code symbols (functions and procedures) that are referenced by another module. You can, however, move or add to these symbols.

4. Do not move or delete data symbols (global data). You can add data symbols at the end of your data definitions, but you cannot add new communal data symbols (for example, C uninitialized variables or BASIC **COMMON** statements).

9.3 The Development Process

To develop a software project with **ILINK**, follow the steps listed below:

1. Use the full linker during early stages of developing your application or dynamic-link library. You will not be ready to take advantage of **ILINK** until you have a number of different code and data segments present.

2. Prepare for incremental linking by using the special linker options described below.

3. Incrementally link with **ILINK** after any small changes are made.

4. Relink with **LINK** after any major changes are made (for example, if you want to add an entirely new module, you want to greatly expand one of the segments or modules, or you want to redefine symbols that are shared between segments).

5. Repeat steps 3 and 4 as necessary.

Three options, **/INCREMENTAL**, **/PADCODE**, and **/PADDATA**, have been added to **LINK** to allow the use of **ILINK**. Here is an example of how they might appear on the command line:

```
LINK /INCREMENTAL /PADDATA:16 /PADCODE:256 @PROJNAME.LNK
```

Sections 9.3.1–9.3.3 present the new options.

9.3.1 The /INCREMENTAL Option

■ **Syntax**

/INC[REMENTAL]

The **/INCREMENTAL** option must be used with the full linker (**LINK**) in order to prepare for subsequent linking with **ILINK**. The use of this option produces a **.SYM** file and a **.ILK** file, which contain extra information needed by **ILINK**. Note that this option is not compatible with **/EXEPACK**.

9.3.2 The /PADCODE Option

■ **Syntax**

/PADC[ODE]:*padsize*

The **/PADCODE** option causes **LINK** to add filler bytes to the end of each code module. The option is followed by a colon and the number of bytes to add. (Decimal radix is assumed, but you can specify special octal or hexadecimal numbers by using a C-language prefix.) Thus

```
/PADCODE:256
```

adds an additional 256 bytes to each module. The default size for code-module padding is 0 bytes.

Note

Code padding is usually not necessary for large- and medium-memory-model programs, but is recommended for small-, compact-, and mixed-memory-model programs, and for **MASM** programs in which code segments are grouped.

To be recognized as a code segment, a segment must be declared with class name `'CODE'`. (Microsoft high-level languages automatically use this declaration for code segments.)

9.3.3 The /PADDATA Option

■ **Syntax**

/PADD[ATA]:*padsize*

The **/PADDATA** option performs a function similar to **/PADCODE**, except that it specifies padding for data segments (or data modules, if the program uses small or medium memory model). Thus

```
/PADDATA:32
```

adds an additional 32 bytes to each module. The default size for data-segment padding is 16 bytes.

Note

If you specify too large a value for *padsize*, you may exceed the 64K limitation on the size of the default data segment.

9.4 Running ILINK

You can attempt to link the project with **ILINK** at any time after the project has been linked with the **/INCREMENTAL** option. The following two sections discuss the files needed for linking with **ILINK** and the command required to invoke **ILINK**.

9.4.1 Files Required for Using ILINK

The files that are required for linking using **ILINK** are **ILINK.EXE**, **EXEC.EXE**, and your project files, which include:

1. *projname*.**EXE** (the file to incrementally link)
2. *projname*.**SYM** (the symbol file)
3. *projname*.**ILK** (the **ILINK** support file)
4. ***.OBJ** (the changed **.OBJ** files)

It is strongly suggested that you place **EXEC.EXE** in a directory listed in the **PATH** environment variable.

9.4.2 The ILINK Command Line

The syntax for the **ILINK** command line is shown below. **ILINK** options are not case sensitive.

ILINK [[/a]] [[/c]] [[/v]] [[/i]] [[/e "*commands*"]] *projname*[[*modulelist*]]

The **/a** option specifies that all object files are to be checked to see if they have changed since the last linking process. This is done by comparing the dates and times of all **.OBJ** files with those of the executable file. An attempt is made to incrementally link all of the files that have changed.

The **/c** option specifies case sensitivity for all public symbol names.

The **/v** option specifies verbose mode, and directs **ILINK** to display more information. Specifically, when in verbose mode, **ILINK** lists the modules that have changed.

The **/i** option specifies that only an incremental link is to be attempted; if the incremental link fails, a full link is not performed.

The **/e** option specifies a list of commands to be executed if the incremental link fails. The commands are enclosed in double quotes, and if more than one command is given, they must be separated by spaces and a semicolon. The characters `%s` are replaced by *projname* when the command is carried out. In the following example, if the incremental link fails, then **ILINK** carries out the commands `link myproj.obj` and `rc myproj.exe`:

```
ILINK /e "link @%s.obj ; rc %s.exe" myproj
```

The *projname* field contains the name of the executable file that is to be incrementally linked.

You can use the optional *modulelist* field to list all the modules that have changed. (No extensions are required.) This field is an alternative to using the **/a** flag.

■ Examples

Two examples using **ILINK** are shown below. In the first example, the altered modules (`input`, `sort`, and `output`) are explicitly listed on the command line. In the second example, the **-a** option directs **ILINK** to scan all files in the project, in order to discover which modules have changed. The second example also lists commands to be executed in the case that incremental linking fails.

```
ILINK /i wizard input sort output

ILINK /a /e "link @%s.lnk ; rc %s.exe" wizard
```

9.5 How ILINK Works

Instead of searching for records and resolving external references for the entire program, **ILINK** carries out the following operations:

1. **ILINK** replaces the portion of each module that has changed since the last linking (incremental or full linking).

2. **ILINK** alters relocation-table entries for any far (segmented) code symbols that have moved within a segment. (For each reference to a far code symbol, such as a far function call, there is an entry in the relocation table in the executable file's header. Unlike the relocation table of a DOS application, the relocation table of an OS/2 application contains full addresses, not just segment addresses. Thus, by fixing relocation table entries for a code symbol, **ILINK** ensures that all references to the symbol will be correct.)

ILINK makes no modification to modules that have not been changed since the last linking.

9.6 Incremental Violations

There are two kinds of **ILINK** failures: real errors and incremental violations. Real errors are errors that will not be resolved by a full link, such as undefined symbols or invalid **.OBJ** files. If **ILINK** detects a real error, it will display `ERROR` with an explanation, and return a nonzero error code to the operating system. On the other

hand, incremental violations consist of changes that are beyond the scope of incremental linking, but can generally be resolved by full linking.

The Microsoft CodeView and Utilities manual explains real errors in detail. The rest of this section describes incremental violations.

9.6.1 Changing Libraries

An incremental violation occurs when a library changes. Furthermore, if an altered module shares a code segment with a library, then **ILINK** will need access to the library as well as to the new module.

Note

If you add a function, procedure, or subroutine call to a library that has never been called before, then **ILINK** will fail with an undefined-symbol error. Performing a full link should resolve this problem.

9.6.2 Exceeding Code/Data Padding

An incremental violation will occur if two or more modules contribute to the same physical segment, and either module exceeds its padding. As discussed in Section 9.3, "The Development Process," padding is the process of adding filler bytes to the end of a module. The filler bytes serve as a buffer zone whenever the module grows in size (that is, whenever the new version of the module is larger than the old).

9.6.3 Moving/Deleting Data Symbols

An incremental violation occurs if a data symbol is moved or deleted. To add new data symbols without requiring a full link, add the new symbols at the end of all other data symbols in the module.

9.6.4 Deleting Code Symbols

You can move or add code symbols, but an incremental violation occurs if you delete any code symbols from a module. Code symbols can be moved within a module but cannot be moved between modules.

9.6.5 Changing Segment Definitions

An incremental violation will result if you add, delete, or change the order of segment definitions. If you are programming in **MASM**, an incremental violation will also result if you alter any **GROUP** directives.

If you are programming with a high-level language, then you need only remember not to add or delete modules between incremental links.

9.6.6 Adding CodeView Debugger Information

If you included CodeView-debugger information for a module the last time you ran a full link (by compiling and linking with CodeView-debugger support), then **ILINK** fully supports CodeView-debugger information for the module. **ILINK** will maintain symbolic information for current symbols, and it will add information for any new symbols. However, if you include CodeView-debugger information for a module which previously did *not* have CodeView-debugger support, an incremental violation will result.

Section 10
The EXEHDR Utility

The Microsoft Segmented EXE File Header Utility (**EXEHDR**) displays the contents of an executable-file header. You can use **EXEHDR** with OS/2 or Windows, and you can use it with an application or dynamic-link library. So there are really four possibilities total. (With a Windows file, some of the meanings of the executable-file-header fields change; consult your Windows documentation for more information.) The principal uses of **EXEHDR** include the following:

- Determining whether a file is an application or a dynamic-link library
- Viewing the attributes set by the module-definition file
- Viewing the number and size of code and data segments

Except where noted otherwise, the specialized terms and keywords mentioned in this section are explained in Section 7, "Using Module-Definition Files."

10.1 The EXEHDR Command Line

To invoke the **EXEHDR** utility, use the following syntax:

EXEHDR [[/v]] *file*

in which *file* is an application or dynamic-link library, for either the OS/2 or Windows environment. The /v option specifies verbose mode, which is discussed in Section 10.3.

Section 10.2 presents sample output and then explains the meaning of each field of the output. Section 10.3 describes additional output that **EXEHDR** produces when it is run in verbose mode.

10.2 EXEHDR Output

This section discusses the meaning of each field in the output below—output produced when `EXEHDR LINK.EXE` is specified on the OS/2 command line. The first six fields list the contents of the segmented-executable-file header. The rest of the output lists

each physical segment in the file. (The term "physical segment" is defined in Section 9, "The ILINK Utility.")

```
Module:                    LINK
Description:               Microsoft Segmented-Executable Linker
Data:                      NONSHARED
Initial CS:IP:             seg   2 offset 3d9c
Initial SS:SP:             seg   4 offset 8e40
DGROUP:                    seg   4

no. type address  file  mem   flags
  1 CODE 00003400 0f208 0f208
  2 CODE 00012e00 05b04 05b04
  3 DATA 00018c00 01c1f 01c1f
  4 DATA 0001aa00 01b10 08e40
```

The `Module` field is the name of the application as specified in the **NAME** statement of the module-definition file. If no module definition was used to create the executable file, then this field displays the name assumed by default. If a module definition was used to create the file, but the **LIBRARY** statement appeared instead of the **NAME** statement (thus specifying a dynamic-link library), then the name of the library is given and **EXEHDR** uses the word `Library` instead of `Module`.

The `Description` field gives the contents, if any, of the **DESCRIPTION** statement of the module-definition file used to create the file being examined.

The `Data` field indicates the type of the program's automatic data segment: **SHARED**, **NONSHARED**, or **NONE**. This type can be specified in a module-definition file, but by default is **NONSHARED** for applications and **SHARED** for dynamic-link libraries. In this context, **SHARED** and **NONSHARED** are equivalent to the **SINGLE** and **MULTIPLE** attributes listed in Section 7.5. (The "automatic data segment" is the physical segment corresponding to the group named **DGROUP**.)

The `Initial CS:IP` field is the program starting address (if an application is being examined) or address of the initialization routine (if a dynamic-link library is being examined).

The `Initial SS:SP` field gives the value of the initial stack pointer.

The `DGROUP` field is the segment number of the automatic data segment. This segment corresponds to the group named **DGROUP** in the program's object modules. Note that segment numbers start with the number 1.

After the contents of the OS/2 executable header is displayed, the contents of the segment table is listed. The following list describes the meaning of each heading in the table. Note that all values are given in hexadecimal radix except for the segment index number.

Heading	Meaning
`no.`	Segment index number, starting with 1, in decimal radix.
`type`	Identification of the segment as a code or data segment. A code segment is any segment with class name ending in `'CODE'`. All other segments are data segments.
`address`	Location within the file, of the contents of the segment.
`file`	Size in bytes of the segment, as contained in the file.
`mem`	Size in bytes of the segment as it will be stored in memory. If the value of this field is greater than the value of the `file` field, then at load time OS/2 pads the additional space with zero values.
`flags`	Segment attributes, as defined in Section 7, "Using Module-Definition Files." If the /v option is not used, then only non-default attributes are listed. Attributes are given in the form specified in Section 7: **READWRITE**, **PRELOAD**, and so forth. Attributes that are meaningful to Windows but not to OS/2 are displayed as lowercase and in parentheses, (e.g., (`movable`)).

10.3 Output in Verbose Mode

When you specify the /v mode, the **EXEHDR** utility gives all the information discussed in Section 10.2, as well as some additional information. Specifically, when running in verbose mode **EXEHDR** displays the following information in this order:

1. DOS 3.x header information. All OS/2 executable files have a DOS 3.x header, whether bound or not. If the program is not bound, then the DOS 3.x portion consists of a stub that simply terminates the program. For a description of DOS executable-header fields, see the *Microsoft MS-DOS Programmer's Reference,* Chapter 5, or see the chapter on the Microsoft EXE File Header Utility (**EXEMOD**) in the Microsoft CodeView and Utilities manual.

2. OS/2-specific header fields. This output includes the fields described in Section 10.2, except for the segment table. (The segment-table display for verbose mode is described below.)

3. File addresses and lengths of the various tables in the executable file, as in the following example:

```
Resource Table:             00003273 length 0000 (0)
Resident Names Table:       00003273 length 0008 (8)
Module Reference Table:     0000327b length 0006 (6)
Imported Names Table:       00003281 length 0033 (51)
Entry Table:                000032b4 length 0002 (2)
Non-resident Names Table:   000032b6 length 0029 (41)
Movable entry points:       0
Segment sector size:        512
```

The first field in each row gives the name of the table, the second field gives the address of the table within the file, the third field gives the length of the table in hexadecimal radix, and the last field gives the length of the table in decimal radix. See the *Microsoft Operating System/2 Programmer's Reference* for an explanation of each table.

4. Segment table with complete attributes, not just the nondefault attributes. In addition to the attributes described in Section 7, verbose mode also displays two additional attributes:

   The `relocs` attribute is displayed for each segment that has address relocations. Relocations occur in each segment that references objects in other segments or makes dynamic-link references. The `iterated` attribute is displayed for each segment that has iterated data. Iterated data consist of a special code that packs repeated bytes; for example, OS/2 executables produced with the **/EXEPACK** option of **LINK**, have iterated data.

5. Run-time relocations and fixups. See the object-module information in the *Microsoft Operating System/2 Programmer's Reference* for more information.

6. Finally, **EXEHDR** lists all exported entry points. Exports are discussed in Section 3, "About Linking in OS/2," and in Section 7.8, "The EXPORTS Statement."

Section 11
LINK Error Messages

This appendix lists error messages that apply only to the protected-mode version of **LINK**, when used to create protected-mode or Windows files. When you create application under DOS 3.x, you will not receive any of the messages listed below.

Number	Linker Error Message
L1005	`/PACKCODE : packing limit exceeds 65536 bytes` The value supplied with the **/PACKCODE** option exceeds the limit of 65,536.
L1030	`missing internal name` An **IMPORT** statement specified an ordinal in the definitions file without including the internal name of the routine. The name must be given if the import is by ordinal.
L1031	`module description redefined` A **DESCRIPTION** in the definitions file was specified more than once, which is not allowed.
L1032	`module name redefined` The module name was specified more than once (via a **NAME** or **LIBRARY** statement), which is not allowed.
L1040	`too many exported entries` The definitions file exceeded the limit of 3072 exported names.
L1041	`resident-name table overflow` The size of the resident-name table exceeds 65,534 bytes. (An entry in the resident-names table is made for each exported routine designated **RESIDENTNAME**, and consists of the name plus three bytes of information. The first entry is the module name.) Reduce the number of exported routines or change some to nonresident.

L1042 `nonresident-name table overflow`

The size of the nonresident-name table exceeds 65,534 bytes. (An entry in the nonresident-names table is made for each exported routine not designated **RESIDENTNAME**, and consists of the name plus three bytes of information. The first entry is the **DESCRIPTION**.) Reduce the number of exported routines or change some to resident.

L1044 `imported-name table overflow`

The size of the imported-names table exceeds 65,534 bytes. (An entry in the imported-names table is made for each new name given in the **IMPORTS** section, including the module names, and consists of the name plus one byte.) Reduce the number of imports.

L1061 `out of memory for /INCREMENTAL`

The linker ran out of memory when trying to process the additional information required for **ILINK** support. If you were linking from within an editor or **MAKE**, try linking directly.

L1062 `too many symbols for /INCREMENTAL`

The program had more symbols than can be stored in the **.SYM** file. Reduce the number of symbols.

L1073 `file-segment limit exceeded`

The number of physical or file segments exceeds the limit of 254 imposed by OS/2 protected mode and by Windows for each application or dynamic-link library. (A file segment is created for each group definition, nonpacked logical segment, and set of packed segments.) Reduce the number of segments or group more of them and make sure that **/PACKCODE** is enabled.

L1074 *name* `: group larger than 64K bytes`

The given group exceeds the limit of 65,536 bytes. Reduce the size of the group, or remove any unneeded segments from the group (look at the map file).

L1075 `entry table larger than 65535 bytes`

The entry table exceeds the limit of 65,535 bytes. (There is a row in this table for each exported routine, and also for each address which is the target of a far relocation and for which one of the following conditions is true: the target segment is designated **IOPL**, or **PROTMODE** is not enabled and the target segment is designated **MOVABLE**.) Declare **PROTMODE** if applicable, or reduce the number of exported routines, or make some segments **FIXED** or **NOIOPL** if possible.

L1082 `stub .EXE file not found`

The linker could not open the file given in the **STUB** statement in the definitions file.

L1092 `cannot open module definitions file`

The linker could not open the definitions file specified on the command line or in the response file.

L1094 *`file`* `: cannot open file for writing`

The linker was unable to open the file with write permission. Check file permissions.

L1095 *`file`* `: out of space on file`

The linker ran out of disk space for the specified output file. Create free disk space or delete root directories.

L1100 `stub .EXE file invalid`

The file specified in the **STUB** statement is not a valid real-mode executable file.

L1126 `conflicting iopl-parameter-words value`

An exported name was specified in the definitions file with an **IOPL**-parameter-words value, and the same name was specified as an export by the Microsoft C **export** pragma with a different parameter-words value.

L2000 `imported starting address`

The program starting address as specified in the **END** statement in a **MASM** file is an imported routine. This is not supported in OS/2 or Windows.

L2010 `too many fixups in LIDATA record`

The number of far relocations (pointer- or base-type) in an **LIDATA** record, which is typically produced by the **DUP** statement in an **.ASM** file, exceeds the limit imposed by the linker. The limit is dynamic: a 1024-byte buffer is shared by relocations and by the contents of the **LIDATA** record, and there are eight bytes per relocation. Reduce the number of far relocations in the **DUP** statement.

L2022 *`name`* `(alias` *`internalname`*`) : export undefined`

The internal name of the given exported routine is undefined.

L2023 *name* (alias *internalname*) : export imported

The internal name of the given exported routine conflicts with the internal name of a previously imported routine. The set of imported and exported names can not overlap.

L2026 entry ordinal number, name *name* : multiple definitions for same ordinal

The given exported name with the given ordinal number conflicted with a different exported name previously assigned to the same ordinal. Only one name can be associated with a particular ordinal.

L2027 *name* : ordinal too large for export

The given exported name was assigned an ordinal which exceeded the limit of 3072.

L2028 automatic data segment plus heap exceed 64K

The total size of data declared in **DGROUP**, plus the value given in **HEAPSIZE** in the definitions file, plus the stack size given by the **/STACKSIZE** option or **STACKSIZE** definitions file statement, exceeds 64K. Reduce near data allocation, **HEAPSIZE**, or stack.

L2030 starting address not code (use class 'CODE')

The program starting address, as specified in the **END** statement of an **.ASM** file, should be in a code segment (code segments are recognized if their class name ends in 'CODE'). This is an error in OS/2 protected mode; the error message may be disabled by including the statement **REALMODE** in the definitions file.

L4000 seg disp. included near offset in segment *name*

This is the warning generated by the **/WARNFIXUP** option. Refer to documentation on that option.

L4001 frame-relative fixup, frame ignored near offset in segment *name*

A reference is made relative to a segment which is different from the target segment of the reference. For example, if `_foo` is defined in segment `_TEXT`, the instruction `call DGROUP:_foo` will result in this warning. The frame **DGROUP** is ignored, so the linker will treat the call as if it were `call _TEXT:_foo`.

L4002 `frame-relative absolute fixup near` *offset* `in segment` *name*

A reference is made similar to the type described in `L4001`, but both segments are absolute (defined with **AT**). It is unclear what this means in OS/2 protected mode or Windows; the linker treats the executable file as if the file were to run in real mode only.

L4010 `invalid alignment specification`

The number specified in the **/ALIGNMENT** option must be a power of 2 in the range 2–32,768 (inclusive).

L4011 `PACKCODE value exceeding 65500 unreliable`

The packing limit specified with the **/PACKCODE** option was between 65,500 and 65,536. Code segments with a size in this range are unreliable on some steppings of the 80286 processor.

L4013 `invalid option for new-format executable file ignored`

The use of overlays and the options **/CPARMAXALLOC**, **/DSALLOCATION**, **/NOGROUPASSOCIATION**, are not allowed with either OS/2 protected-mode or Windows executables.

L4014 `invalid option for old-format executable file ignored`

The **/ALIGNMENT** option is invalid for real-mode executables.

L4022 *group1*`,` *group2* `: groups overlap`

The named groups overlap. (Since a group is assigned to a physical segment, groups cannot overlap with either OS/2 protected-mode or Windows executables.) You should reorganize segments and group definitions so the groups do not overlap. Refer to the map file.

L4023 *name* `(`*internal name*`) : export internal name conflict`

The internal name of the given exported routine conflicted with the internal name of a previous import definition or export definition.

L4024 *dynlib.import* `(`*name*`) : multiple definitions for export name`

The given name was exported more than once, which is not allowed.

L4025 *dynlib.import* `(`*name*`) : import internal name conflict`

The internal name of the given imported routine (*import* is either a name or a number) conflicted with the internal name of a previous export or import.

L4026 *name* : self-imported

The given imported routine was imported from the module being linked. This is not supported on some systems.

L4027 *name* : multiple definitions for import internal-name

The given internal name was imported more than once. Previous import definitions are ignored.

L4028 *name* : segment already defined

The given segment was defined more than once in the **SEGMENTS** statement of the definitions file.

L4029 *name* : DGROUP segment converted to type data

The given logical segment in the group **DGROUP** was defined as a code segment. (**DGROUP** cannot contain code segments, because the linker always considers **DGROUP** to be a data segment. The name **DGROUP** is predefined as the automatic data segment.) The linker converts the named segment to type "data."

L4030 *name* : segment attributes changed to conform with automatic data segment

The given logical segment in the group **DGROUP** was given sharing attributes (**SHARED/NONSHARED**) which differed from the automatic data attributes as declared by the **DATA** *instance* (**SINGLE/MULTIPLE**). The attributes are converted to conform to those of **DGROUP**. Refer to Error L4029 for more information on **DGROUP**.

L4032 *name* : code-group size exceeds 65500 bytes

The given code group has a size between 65,500 and 65,536 bytes, which is unreliable on some steppings of the 80286 processor.

L4036 no automatic data segment

The application did not define a group named **DGROUP**. **DGROUP** has special meaning to the linker, which uses it to identify the automatic or default data segment used by the operating system. Most OS/2 protected-mode and Windows applications require **DGROUP**. This warning will not be issued if **DATA NONE** is declared or if the executable is a dynamic-link library.

L4042 cannot open old version

The file specified in the **OLD** statement in the definitions file could not be opened.

L4043 `old version not segmented-executable format`

The file specified in the **OLD** statement in the definitions file was not a valid OS/2 protected-mode or Windows executable.

L4046 `module name different from output file name`

The name of the executable as specified in the **NAME** or **LIBRARY** statement is different from the output file name. This may cause problems; you should consult the documentation for your operating system.

Microsoft® Editor

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

User's Guide

Microsoft Corporation

Contents

Figures and Tables

Figures

Tables

Chapter 1
Introduction

Welcome to the Microsoft® Editor. The Microsoft Editor is a powerful software development tool that runs in OS/2 systems and in DOS 2.1 and above. It lets you create source files, customize editing functions, and invoke compilers (or other utilities such as assemblers). The pages that follow use the term "OS/2" to refer to both the Microsoft Operating System/2 (MS® OS/2) and IBM® OS/2. Similarly, the term "DOS" is used to refer to both MS-DOS® and PC-DOS where appropriate.

You can use the Microsoft Editor as a simple text editor, but it is particularly useful for writing programs. The following list describes some of the flexible ways you can use the editor:

- **Compile and Link Programs from within the Editor**

  The Microsoft Editor is more than a text editor; it is a development environment. Develop programs more quickly by compiling from within the editor. If the compile fails, then view the errors, rewrite the program, and recompile—all without leaving the editor.

- **Customize the Editor**

  The Microsoft Editor lets you reassign editing functions to different keys. You can specify function assignments in the initialization file; the editor automatically recognizes these assignments each time you run it. You can change these function assignments at any time during an editing session.

- **Write New Editing Functions in C**

  If you use Microsoft C, then you can write new editing functions for the Microsoft Editor. Write a C-language module using the standard C data and control-flow structures, and call the editor's low-level editing functions to read and write to a file. The editor loads the module into memory and calls it on command.

- **Save Typing Effort with Macros**

  A macro is a command which performs a series of predefined actions; for example, a macro can insert a given phrase or word or perform an entire series of editing commands. Define a macro, then invoke it with one keystroke.

- **Edit Complex Files with Windows**

  When you edit a large file, you may want to view different parts of the file simultaneously. With the Microsoft Editor, you can split up your screen into as many as eight windows, each displaying a different part of the file.

- **Handle Multiple Source Files**

  With a simple command, you can transfer back and forth between the different files that you are working on—there is no need to leave the editor and then start it up again. Furthermore, as the editor moves between files, it saves cursor position and other relevant information. You can view portions of different files simultaneously by using windows.

1.1 System Requirements

To use the Microsoft Editor, you need to have MS OS/2 running in protected mode, or DOS 2.1 or above with at least 128 kilobytes (K) of available memory. A minimum of 150K of available memory is required to use the C extensions described in Chapter 8.

1.2 Using This Manual

Different parts of the manual address different learning needs, as explained below:

- If you have not used the Microsoft Editor before, you should read Chapter 2, "Edit Now," and Chapter 3, "Command Syntax," before proceeding.

- To start using the Microsoft Editor right away, read Chapter 2, "Edit Now." This chapter uses a specific example to describe the basic editing functions.

- To get a more general understanding of the many editing functions, read Chapter 3, "Command Syntax." This chapter explains how you can specify different kinds of arguments for editing functions. Then read Chapter 4, "A Survey of the Microsoft Editor's Commands," which explores major topics such as searching and replacing text, compiling, and creating windows.

- For definitions of terms and concepts, turn to the glossary at the back of the manual. Although all terms are defined in the text, you may find it helpful to refer to the glossary as you learn about the editor.

- After you have used the editor to perform simple editing tasks, and understand how to enter arguments, you may want to refer directly to Appendix A, "Reference Tables." These tables provide complete descriptions of all functions and commands.

- To use the utility programs (**CALLTREE**, **EXP**, **MEGREP**, **RM**, and **UNDEL**) that come with the editor, see Appendix B, "Support Programs for the Microsoft Editor."

The Microsoft Editor comes with a setup program (**MESETUP.EXE**) that configures the editor so that it uses keystroke assignments similar to Microsoft Quick languages and WordStar®, the BRIEF® editor, or the Epsilon™ editor. It is recommended that you work through Chapter 2, "Edit Now," with the standard defaults for keystrokes, before you run the setup program. See the **README.DOC** file for information on how to use the setup program.

1.3 Typographic Conventions

The following typographic conventions are used throughout this manual and apply in particular to syntax displays for commands and switches:

Example of Convention	Description
KEY TERMS	Bold letters indicate a specific term or punctuation mark that you must type in as shown. The use of uppercase or lowercase letters is not significant. For example, in a function assignment, the word **Unassigned** must be typed in as shown, but the first letter need not be capitalized.
`Example: input`	The typeface shown in the left column is used in examples to simulate the appearance of information printed on your screen. The bold version of this typeface indicates input entered in response to a prompt.
placeholders	Words in italics indicate a field or a general kind of information; you must supply the particular value. For example, *numarg* represents a numerical argument that you type in from the keyboard. You could type in a number, such as `15`, but you would not type in the word "*numarg*" itself.

[[optional items]]	Items inside double square brackets are optional.
{choice1 | choice2}	Braces and a vertical bar indicate a choice among two or more items. You must choose one of the items unless all of the items are also enclosed in double square brackets.
Repeating elements...	Three dots following an item indicate that more items having the same form may appear.

`Program`
.
.
.
`Fragment`
	A column of three dots tells you that part of a program has been intentionally omitted.

KEY NAMES	Small capital letters are used for the names of keys and key sequences, such as ENTER and CTRL+R. Notice that a plus (+) indicates a combination of keys. For example, CTRL+E tells you to hold down the CTRL key while pressing the E key. The names of the keys referred to in this manual correspond to the names printed on the IBM Personal Computer key tops. If you are using a different machine, these keys may have slightly different names. The cursor movement keys (sometimes called "arrow" keys) that are located on the numeric keypad to the right of the main keypad are called the DIRECTION keys. Individual DIRECTION keys are referred to either by the direction of the arrow on the key top (LEFT, RIGHT, UP, DOWN) or the name on the key top (PGUP, PGDN). Some of the Microsoft Editor's functions use the +, –, or number keys on the numeric keypad, rather than the ones on the top row of the main keyboard. At each instance, the text notes the use of keys from the numeric keypad. The carriage-return key is referred to as ENTER.
"Defined term"	Quotation marks usually indicate a term defined in the text.

Chapter 2
Edit Now

This chapter helps you use the Microsoft Editor right away by focusing on the functions you need to create a simple text file. Functions are built-in capabilities that you invoke to give directions to the editor. Most of the chapter consists of a tutorial that uses a specific example and features the following functions:

Function	Default Keystroke
Cursor movement	DIRECTION keys, HOME
Insertmode	INS
Sdelete (stream delete)	DEL
Ldelete (line delete)	CTRL+Y
Arg (introduce argument)	ALT+A
Cancel	ESC
Undo	ALT+BKSP
Paste	SHIFT+INS
Psearch (forward search)	F3
Exit	F8
Help	F1
Setfile (move to previous file)	F2

You can use this tutorial either by starting the editor and typing in each command as shown, or you can simply read along. Because the results are explained at each stage, you can get a good understanding of the editor just by reading.

The chapter ends by presenting the complete command line for the editor, with all the possible options you may use.

2.1 Starting the Editor

Copy the file **M.EXE** into your current directory or a directory listed in the **PATH** environment variable. To run the editor in protected mode, copy the file **MEP.EXE**. (You may want to rename the file as **M.EXE**.) Then start the editor with this command:

```
M NEW.TXT
```

The Microsoft Editor responds by asking if you want to create a new file by this name. Press Y to indicate yes. The editor creates the file, and you are ready to enter text.

2.2 The Microsoft® Editor's Screen

When you start the editor with a new file, you see a screen that is mostly blank (see Figure 2.1):

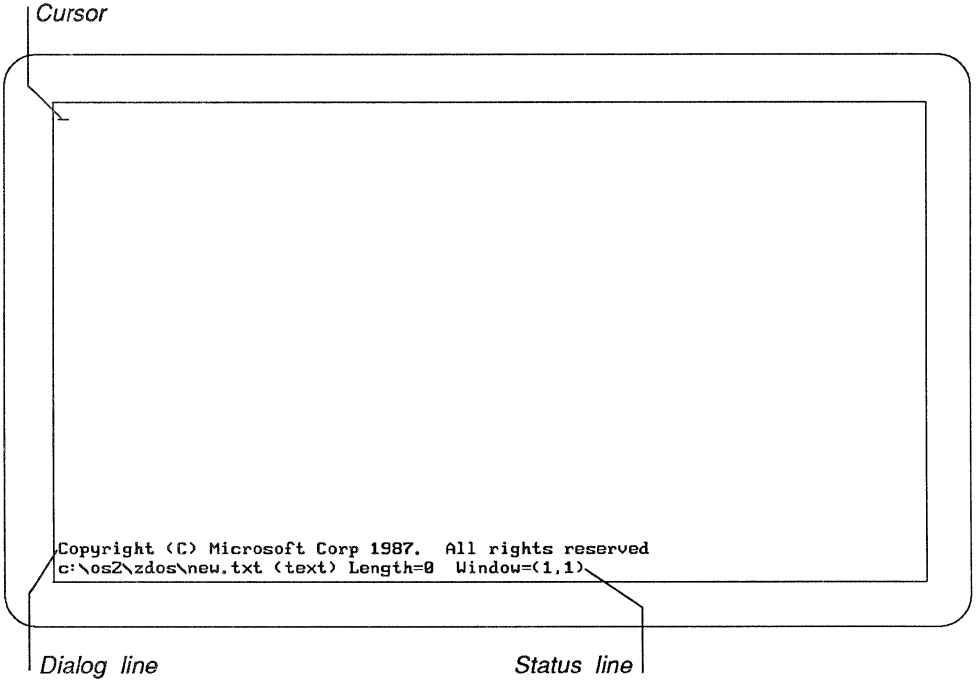


Figure 2.1 Microsoft Editor's Screen

The cursor first appears at the upper-left corner of the screen. Even though the file is empty, you can use the DIRECTION keys—denoted as UP, DOWN, LEFT, and RIGHT—to move the cursor anywhere on the screen. (The DIRECTION keys are the arrow keys on the numeric keypad.) Try experimenting with cursor movement.

The next-to-bottom line is called the "dialog line," which is reserved for displaying messages from the editor and letting you enter text arguments. The bottom line is called the "status line," and it always displays the following fields:

Field	Description
`c:new.txt`	File name, with complete path
`(text)`	Type of file
`Length=1`	Length of file, in number of lines (minimum value is 1)
`Window=(1,1)`	Window or cursor position

The field `Window=(1,1)` indicates that the upper-left corner of the screen corresponds to the first row and column of the file. As you scroll through files that are larger then one screen, the numbers in this field change. See Section 7.4.2, "Boolean Switches," to learn how to alter this field so that it displays cursor position instead of window position.

2.3 Sample Session

Once the Microsoft Editor is started, you can enter text immediately. Simply start typing, and press ENTER when you want to begin a new line. By default, the editor starts in "overtype" mode, which means that anything you type replaces the text at the cursor position.

To begin, type in the following text. There are some deliberately planted errors that you'll correct in a few moments.

```
It's mind over matter.
What is mind?
No mat matter.
Wh is matter?
Mever mind._
```

The third, fourth, and fifth lines have errors near the beginning of each line. To get to the beginning of the fifth line, you could press the LEFT key until you got to the beginning of the line. However, you can get there faster by pressing the HOME key. This key moves the cursor to the first nonblank character in the line.

Now move the cursor to the beginning of the fifth line and correct the error by typing the letter N:

```
Never mind.
```

2.3.1 Inserting Text with the Insertmode Function

To insert text in this example, move the cursor to the third position in the fourth line:

```
Wh_is matter?
```

The letters `at` need to be inserted at the end of the first word. Press the INS key to invoke the *Insertmode* function, which toggles between overtype and insert mode. You'll see the word `insert` appear at the end of the status line. Type the letters `at` to produce the following line:

```
What_is matter?
```

2.3.2 Removing a Word with the Delete Function

So far, you've used editing functions to replace old text and insert new text. The third line requires text deletion, so move the cursor to the beginning of the second word in the third line:

```
No mat matter.
```

One of the text-deletion functions is *Sdelete*, which stands for "stream delete." Invoke the *Sdelete* function by pressing the DEL key. You can use the *Sdelete* function in different ways. For example, you can delete the character at the current cursor position by just pressing the DEL key. You can also delete a group of characters with the following sequence:

ALT+A *move-cursor* DEL

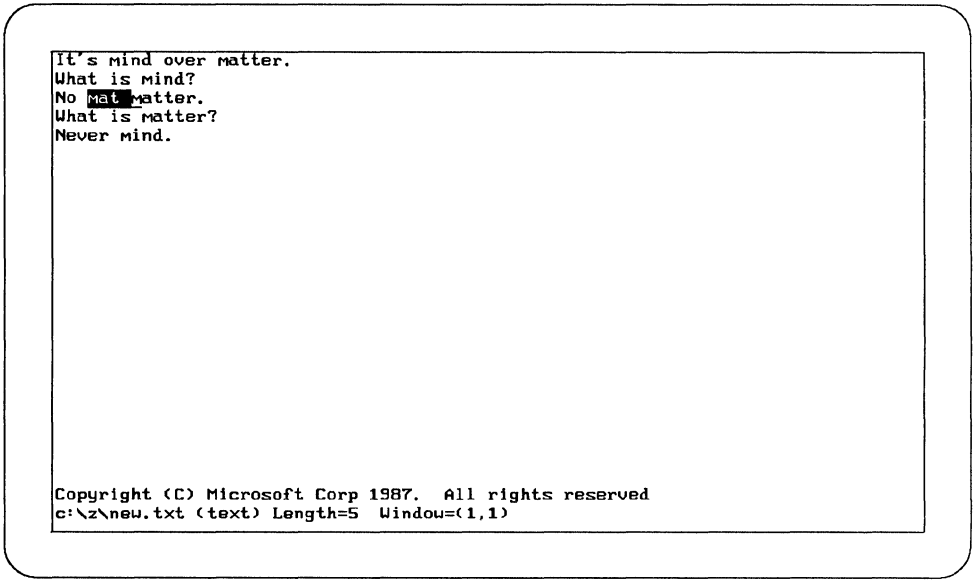
Section 2.3.3, "Introducing the Arg Function," examines this command sequence in detail.

2.3.3 Introducing the Arg Function

To invoke the *Arg* function, press ALT+A by holding down the ALT key and then pressing A.

The *Arg* function does nothing by itself; you use it to introduce an argument to another function. (An arguments is information, such as text or highlighted characters, that the function works with.) In this case you'll use the *Arg* function to highlight the group of

characters you wish to delete. After pressing ALT+A, move the cursor to the beginning of the third word. Your screen should appear as follows:



Now press DEL, and the highlighted characters are removed.

2.3.4 Canceling and Undoing Commands

If you pressed ALT+A at the wrong time but did not complete the command you were typing, you can cancel the argument by pressing the ESC key. This keystroke invokes the *Cancel* function. The *Cancel* function lets you start a command sequence over again.

If you complete a command that was incorrect, reverse the command by pressing ALT+BKSP (hold down the ALT key and then press the backspace key). This keystroke invokes the *Undo* function. If you invoke *Undo* again, it reverses the next-to-last editing command. Invoke *Undo* a third time, and it reverses the second-to-last editing command, and so on. The number of commands that the editor remembers is controlled by the **undocount** switch, discussed in Chapter 7, "Using the TOOLS.INI File." The default number of commands remembered is 10.

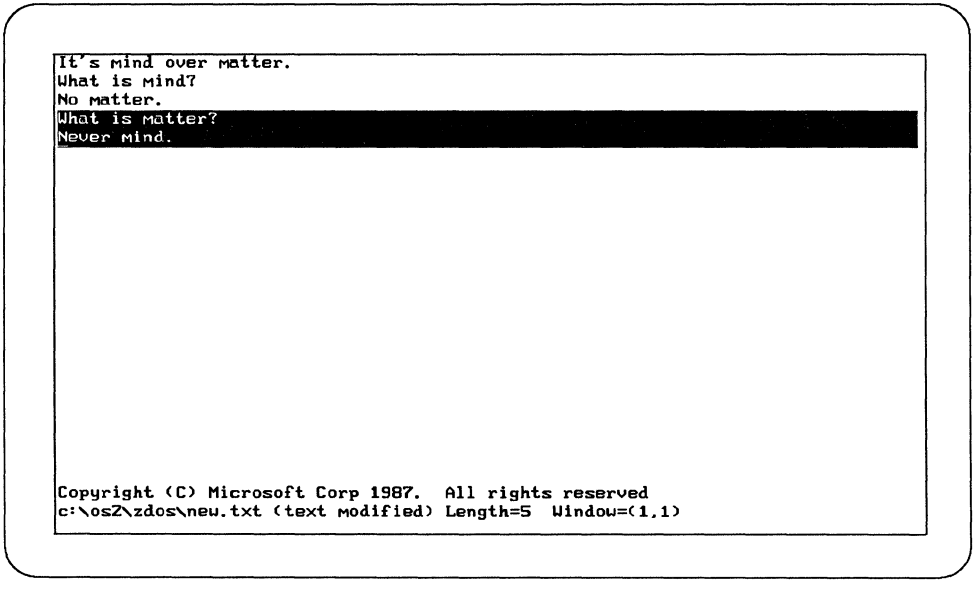
2.3.5 Using Ldelete to Move Text

As is the case with other editors, delete functions in the Microsoft Editor serve a dual purpose: deleting text and moving text. The last text deleted is placed into the "Clipboard." (The Clipboard is a special section of memory that holds text placed there by the *Copy, Ldelete,* or *Sdelete* functions.) When you press SHIFT+INS, which invokes the *Paste* function, the contents of the Clipboard are inserted into the file.

The stream-delete function (*Sdelete*, presented above) is useful for deleting a series of characters on the same line. The line-delete function, *Ldelete*, provides the most efficient way of deleting entire lines. In this section, *Ldelete* will be used to move two complete lines of text. Consider the current text:

```
It's mind over matter.
What is mind?
No Matter.
What is matter?
Never mind.
```

Move the cursor to any place in the fourth line. Then select the bottom two lines by pressing ALT+A and then pressing the DOWN key once. You should see the bottom two lines highlighted, as follows:



Now invoke the *Ldelete* function by pressing CTRL+Y. The two lines disappear. In general, the *Ldelete* function deletes whatever characters you highlight.

Having deleted a block of characters, you are now ready to use the *Paste* function (SHIFT+INS) to put the deleted text into a new location in your document. Move the cursor to the beginning of the top line and press SHIFT+INS. You should now see the following text:

```
What is matter?
Never mind.
It's mind over matter.
What is mind?
No matter.
```

You can also invoke *Paste* with an argument. Try this sequence of keystrokes:

1. Press ALT+A.
2. Type the following text:

   ```
   The Philosopher said,
   ```

3. Press SHIFT+INS.

The result is that the words `The Philosopher said,` are inserted at the current cursor position. `The Philosopher said,` is an example of a "text argument." Text arguments automatically appear on the dialog line, so you can see what you're typing. Use DEL to correct errors as you're typing a text argument.

2.3.6 Searching with Psearch

The *Psearch* function takes different kinds of arguments and applies them in a consistent way. The term *Psearch* stands for "plus search," and means the same thing as "forward search." This function, which is assigned to the F3 key, takes both text arguments and cursor-movement arguments. You can ask the editor to locate the next occurrence of the word `mind` by typing the word in as a text argument. Move the cursor to the beginning of the file, then try the following sequence of keystrokes:

1. Press ALT+A.
2. Type the following text:

   ```
   mind
   ```

3. Press F3.

You can achieve the same result by moving the cursor to the beginning of the word `mind` on the screen, then highlighting the word with the following sequence of keystrokes:

ALT+A RIGHT RIGHT RIGHT RIGHT F3

An even easier way of selecting the word is to give the keystroke sequence ALT+A F3, which selects the word at the current cursor location. This word (all characters up to the first blank or new-line character) becomes the search string.

Often when you use the *Psearch* function, you want to look repeatedly for some text string. To search for a text string previously specified, press F3 by itself.

2.3.7 Exiting the Editor

Press F8 to leave the editor. The F8 key sequence invokes the *Exit* function, which automatically saves any changes you have made to the file and exits.

2.4 Getting Help

As you work with the Microsoft Editor, you may occasionally forget which function is assigned to which key. Press F1 to get a complete list of all key assignments. You examine this list the way you edit a file; use the DIRECTION keys, PGUP, and PGDN to navigate through the list. You may see that some functions are unassigned. In later chapters, you'll learn how to assign these functions to keystrokes.

Press F2 to get back to your file. The F2 key invokes the *Setfile* function, which always takes you back to the file you were editing.

2.5 The Microsoft Editor's Command Line

Use the following command line to start up the editor:

M[[**/D**]] [[**/e** *command*]] [[**/t**]] [[*files*]]

If you are using the protected-mode version, then the name of the editor's executable file is **MEP.EXE**. You can rename this file to **M.EXE**.

The **/D** option prevents the editor from examining **TOOLS.INI** for initialization settings (see Chapter 7, "Using the TOOLS.INI File," for more information).

The **/e** option enables you to specify a command upon startup. The *command* argument is a string that follows the same syntax rules as those given for macros in Chapter 6, "Function Assignments and Macros." If *command* contains a space, then the entire string should be enclosed by double quotes.

The **/t** option specifies that any files edited in this session are temporary. The editor will not list these files in the information file after this session is terminated.

If a single *file* is specified, then the editor attempts to load the file. If the file does not yet exist, the editor asks you if you want to create the file. If multiple *files* are specified, the first file is loaded; then, when you invoke the *Exit* function, the editor saves the current file and loads the next file in the list. If no *files* are specified, then the editor attempts to load the file you were editing when you last exited the editor.

Upon startup, the status line displays at least four fields. The status line can display up to eight fields, as follows:

1. Name of the file being edited
2. Type of file (based on extension)
3. The word `modified` if the file has been changed
4. The letters `NL` if no carriage returns were found when the file was loaded (that is, if the file did not contain carriage returns to denote the end of each line, but used only line feeds)
5. The length of the file, in lines
6. Cursor position or window position of upper-left corner
7. The word `insert` if you are in insert mode
8. The word `meta` if you have invoked the *Meta* function

Chapter 3
Command Syntax

If you've worked through Chapter 2, "Edit Now," you already have an understanding of the flexibility of commands in the Microsoft Editor. Many of the editing functions accept a variety of arguments—text arguments, cursor-movement arguments (which you select by highlighting), or no argument at all. In this chapter, you'll learn about each kind of argument. The chapter also presents the syntax conventions used throughout the manual.

Topics are covered in the following order:

- Commands and functions
- Entering a command
- Argument types
- Text arguments (*numarg, markarg, textarg)*
- Cursor-movement arguments (*streamarg, linearg, boxarg)*

3.1 Commands and Functions

This manual often refers to "commands" and "functions." While these two concepts are closely related, they are not necessarily the same.

A command is a complete instruction, providing the editor with all the information that it needs to carry out a specific activity. A command may consist of a single function, or it may consist of several functions and an argument.

A function is a built-in editing capability. You invoke a function with a specific keystroke. Chapter 6 explains how to assign keys to functions, but most functions have default keys already assigned to them.

Each command can include at most one argument. An argument can consist of text that you type in, or characters on screen that you highlight with cursor movement. The argument is passed to the function that follows it.

Note

Throughout this manual, function names are given in italics and are capitalized (for example: *Paste*). Argument types are given in italics and are lowercase (for example: *textarg*). Although functions correspond to specific keystrokes, argument types are fairly broad categories. For example, *textarg* corresponds to any line of text that you explicitly type as an argument. See Sections 3.3–3.5 for more information.

3.2 Entering a Command

This section explains how to enter a command. A command can be as simple as a single function, or it may be more complex.

The following three rules describe the general syntax of a command. You do not need to memorize these rules; they are provided here for the sake of understanding.

1. You must use the *Arg* prefix (press ALT+A) when introducing an argument.

2. You can use *Arg Arg* (press ALT+A twice) in place of *Arg*. Some functions attach a special meaning to *Arg Arg*.

3. Some functions recognize the *Meta* (F9) prefix.

The first rule is that you use the *Arg* function (ALT+A) when you want to introduce an argument. The general syntax of a command that uses the *Arg* function is:

Arg argument Function

This syntax applies regardless of the type of argument you enter. As soon as you invoke *Arg* (by pressing ALT+A), the editor highlights the current cursor position. This position stays fixed, even if you enter new text or continue to move the cursor.

The following list gives examples of the *Arg argument Function* syntax:

Command	Default Keystrokes
Arg textarg Psearch	ALT+A *type-characters* F3
Arg linearg Ldelete	ALT+A *move-cursor* CTRL+Y
Arg streamarg Sdelete	ALT+A *move-cursor* DEL

You can also use the *Arg* prefix without specifying an argument, for example, ALT+A F3. When you do not explicitly give an argument, the function following *Arg* assumes some argument based on the cursor position. For example, ALT+A F3 takes the word at the cursor position as the argument.

The second rule of syntax is that some functions recognize the prefix *Arg Arg* (press ALT+A twice) as well as the prefix *Arg*. You use *Arg Arg* to introduce an argument just as you do with *Arg*; however, the use of *Arg Arg* modifies the function's effect in some predefined way. For example, consider the following commands:

Command	Default Keystrokes
Arg textarg Psearch	ALT+A *type-characters* F3
Arg Arg textarg Psearch	ALT+A ALT+A *type-characters* F3

The first command searches for an ordinary text string, whereas the second command recognizes a special string called a "regular expression." See Chapter 5 for more information on regular expressions.

The third rule of syntax is that some functions accept the optional prefix *Meta* (F9). The *Meta* prefix alters the effect of the function in some predefined way. For example, whereas *Up* moves the cursor up one line, *Meta Up* (F9 UP) moves the cursor up to the top of the screen.

When you invoke *Meta*, the phrase `(meta)` is displayed on the status line. The *Meta* prefix, if used, should occur just before the function that it modifies. Thus the following are examples of valid commands:

Command	Default Keystrokes
Meta Right	F9 RIGHT
Arg Meta Compile	ALT+A F9 F5
Arg textarg Meta Setfile	ALT+A *type-characters* F9 F2

3.3 Argument Types

The Microsoft Editor provides two basic ways to enter arguments: you can enter text directly, as part of the command (text argument), or you can use cursor movement to highlight characters on the screen (cursor-movement argument). Each of these two methods has several variations, as shown in the following list:

1. Text argument. After you invoke *Arg* (ALT+A), continue to type characters. These characters appear on the dialog line (the line next to the bottom of the screen). You can give three different kinds of text arguments:
   a. A *numarg*, which consists of a string of digits.
   b. A *markarg*, which is a string containing the name of a previously defined file marker.
   c. A *textarg*. A text argument not recognized as a *numarg* or *markarg*; it is considered simply a *textarg*.
2. Cursor-movement argument. After you invoke *Arg*, the current cursor position is highlighted. Highlight more characters by moving the cursor to a new position. You can give three different kinds of cursor-movement arguments:
   a. A *streamarg*, in which the old and new cursor positions are in the same line
   b. A *linearg*, in which the old and new cursor positions are in a different line but in the same column
   c. A *boxarg*, in which the old and new cursor positions are in a different line and column

Sections 3.3.1 and 3.3.2 give more detailed information on each type of argument, along with examples.

3.3.1 Text Arguments (numarg, markarg, textarg)

After you invoke *Arg* (ALT+A), you can enter a text argument by typing any printable characters, including blank spaces. As soon as you begin entering text, the dialog line

on the screen (next to the bottom line of the screen) shows the word `Arg:` followed by your text. For example, if you press ALT+A and then type the letter T, you see the following items on the dialog line:

```
Arg: T
```

When you enter a text argument, you can use the following six editing capabilities:

1. Erase the character at the current cursor position with the *Sdelete* function (DEL).
2. Backspace to the left, while erasing a character, with the *Cdelete* function (CTRL+G).
3. Move back and forth nondestructively with LEFT and RIGHT. If you use RIGHT to move past the end of current input, the editor inserts a character from the previous text argument.
4. Insert a space at the cursor position with the *Sinsert* function (CTRL+J).
5. Move to beginning of the text with *Begline* (HOME) and to the end of the text with *Endline* (END).
6. Clear characters to the end of the line with the *Arg* function (ALT+A).

Sections 3.3.1.1–3.3.1.3 present the possible variations of text arguments.

3.3.1.1 The numarg Type

A *numarg* is string of digits that you enter as a text argument. Each of the three following examples is a *numarg*:

```
3
11
45
```

The number must be a valid decimal integer. A *numarg* is evaluated as a number and not as literal text. Typically, it is used to indicate a range of lines starting with the

cursor position. For example, the following command sequence deletes 10 lines starting with the cursor position:

1. Invoke *Arg* (press ALT+A).
2. Type the following text:

   ```
   10
   ```

3. Invoke *Ldelete* (press CTRL+Y).

Some functions accept text arguments but do not recognize a *numarg*. In these cases, a *numarg* is treated as an ordinary *textarg* (see Section 3.3.1.3).

3.3.1.2 The markarg Type

A *markarg* is a file-marker name that you have previously defined with the *Mark* function (CTRL+M). See Section 4.3, "Using File Markers," for information about *Mark*.

Once defined, you can enter the marker name as a *markarg*. The name is not treated as literal text, but is interpreted as an actual file position. For example, the following command sequence copies all text between the cursor position and the file position previously marked as `P1`:

1. Invoke *Arg* (press ALT+A).
2. Enter the following text:

   ```
   P1
   ```

3. Invoke *Copy* (press the + key on the numeric keypad).

Many functions accept text arguments but do not recognize a *markarg*. In these cases, the *markarg* is treated as an ordinary *textarg*.

3.3.1.3 The textarg Type

A *textarg* is similar to a *numarg* or *markarg*. The only difference is that the *textarg* has no special meaning; it is interpreted by the function as literal text. For example, the following sequence inserts the string `Happy New Year` into the file, exactly as typed:

1. Invoke *Arg* (press ALT+A).
2. Type the following:

   ```
   Happy New Year
   ```

3. Invoke *Paste* (press SHIFT+INS).

3.3.2 Cursor-Movement Arguments (streamarg, linearg, boxarg)

You enter a cursor-movement argument by invoking *Arg* (ALT+A) and then moving the cursor. When you invoke *Arg*, the current cursor position is marked with a reverse-video highlight. This position is called the "initial cursor position." You then can move the cursor; as you do, characters between the initial cursor position and the new cursor position are highlighted, as described in Sections 3.3.2.1–3.3.2.3.

Each function determines how to interpret a cursor-movement argument. Some functions work with a "stream of text" between the initial cursor position and the new position. A stream of text consists of a continuous series of characters as they are actually stored in the file. With a stream of text, the area highlighted is irrelevant; only the two positions matter.

Other functions work with the area highlighted on the screen. As explained in Sections 3.3.2.2 and 3.3.2.3, this area may be a rectangular box, or it may consist of complete lines.

Chapter 4, "A Survey of the Microsoft Editor's Commands," and Appendix A, "Reference Tables," describe how each function interprets cursor-movement arguments. For example, *Sdelete* always deletes a stream of text, whereas *Ldelete* deletes the area of text that is highlighted.

3.3.2.1 The streamarg Type

A *streamarg* consists of characters on a single line. The term *streamarg* refers to the fact that characters on a single line are always treated as a stream of text, regardless of the function involved.

After invoking *Arg*, you can move the cursor left or right. A *streamarg* consists of characters beginning with the leftmost of the two positions (initial cursor position or new cursor position), up to but not including the rightmost position, as shown in Figure 3.1:

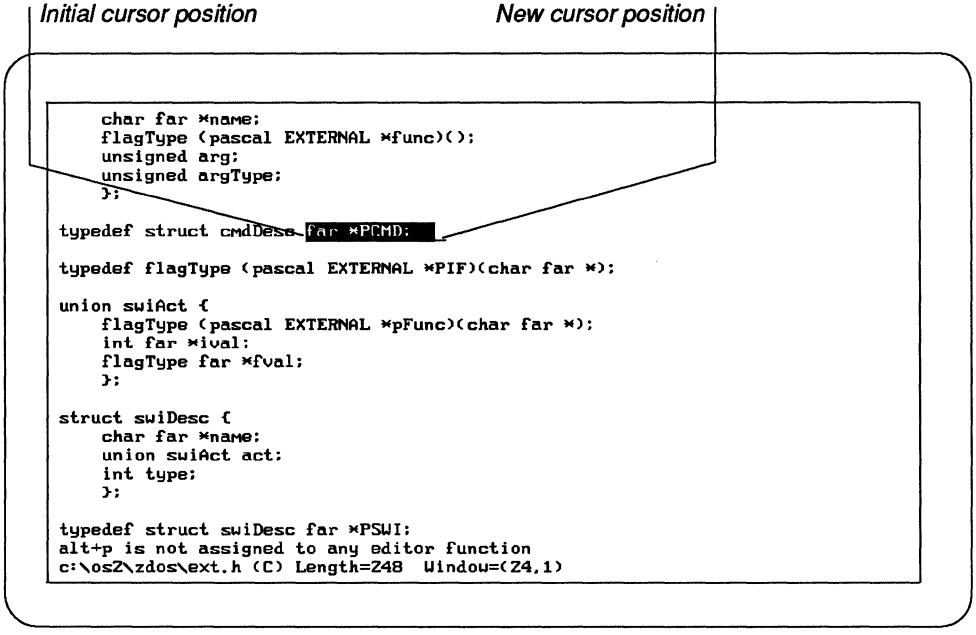


Figure 3.1 Sample streamarg

3.3.2.2 The linearg Type

A *linearg* is defined when the new cursor position is in the same column but on a different line from the initial cursor position. The editor responds by highlighting all lines between the two cursor positions, including the lines that the cursor positions are on. For example, the display in Figure 3.2 is produced by invoking *Arg* and then pressing DOWN three times:

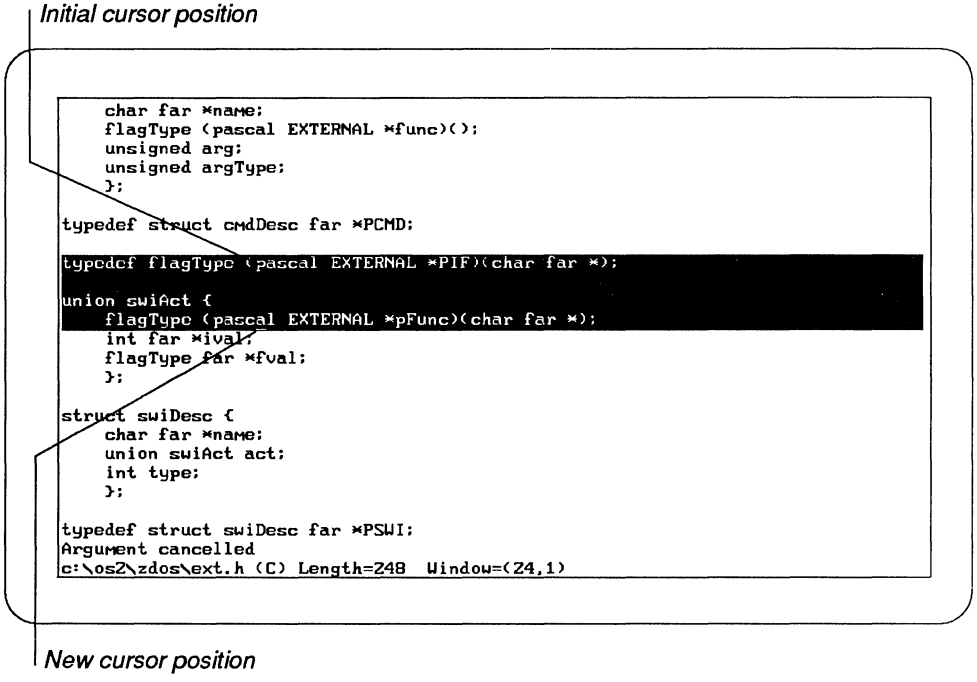


Figure 3.2 Sample linearg

3.3.2.3 The boxarg Type

A *boxarg* consists of a rectangular area on the screen. The two corners of the area are determined by the initial and new cursor positions. A *boxarg* is defined when the two positions are in both a different line and a different column from each other.

After invoking *Arg*, you can move the cursor left or right. The left edge of the box includes the leftmost of the two cursor positions. The right edge of the box includes the column just to the left of the rightmost of the two cursor positions. The box includes parts of all lines between the two positions.

For example, the display shown in Figure 3.3 is produced by invoking *Arg* and then moving the cursor 3 lines down and 10 columns over:

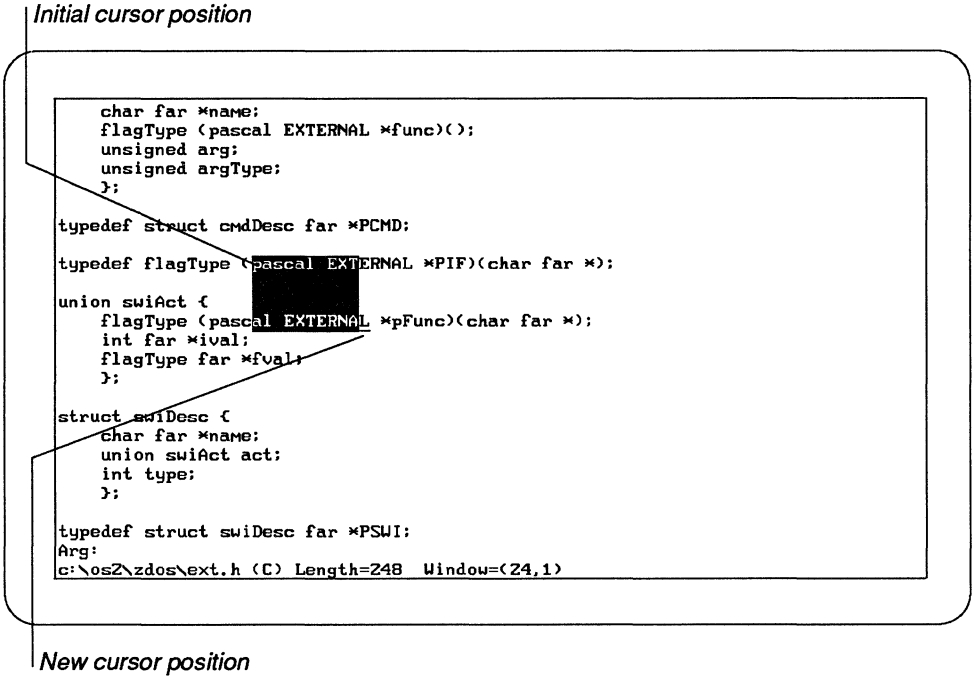


Figure 3.3 Sample boxarg

Chapter 4

A Survey of the Microsoft Editor's Commands

The Microsoft Editor features all the standard capabilities of a text editor: fast navigation through a file, and the ability to move blocks of text, search for strings, and handle multiple files. In addition, the Microsoft Editor supports a flexible windowing capability for viewing more than one file or more than one part of the same file. The Microsoft Editor can also invoke compilers and assemblers and let you easily view compile errors.

This chapter presents specific editing topics in more detail than they were covered in Chapter 2, "Edit Now." Topics are presented in this order:

- Moving through a file
- Inserting, copying, and deleting text
- Using file markers
- Searching and replacing text
- Compiling
- Using windows
- Working with multiple files

Each section presents the most common functions within the given topic and gives examples of how the functions can be used. If appropriate, the section ends with a brief description of other related functions. See Appendix A, "Reference Tables," for an exhaustive listing of the command syntax for each function.

4.1 Moving through a File

Chapter 2, "Edit Now," described how to the use DIRECTION keys to move through a file one space at a time. The DIRECTION keys correspond to the functions *Up*, *Down*, *Right*, and *Left*, to which you can assign different keys if you wish. Chapter 2 also

presented the *Begline* function (HOME), which moves the cursor to the first printable character in the current line. Similar to the *Begline* function is the *Endline* function (END), which moves the cursor just to the right of the last printable character in the current line.

Each of the four DIRECTION functions has a variation that uses the *Meta* function as a prefix, as shown in the following list. Each of these functions, when used in a command with the *Meta* prefix, moves the cursor as far as possible within the displayed screen (or window) without changing column position or causing the screen to scroll in any way.

Command (and Default Keystrokes)	Description
Meta Up (F9 UP)	Moves the cursor to the top of the screen
Meta Down (F9 DOWN)	Moves the cursor to the bottom of the screen
Meta Left (F9 LEFT)	Moves the cursor to the leftmost position on the current line
Meta Right (F9 RIGHT)	Moves the cursor to the rightmost position on the current line

4.1.1 Scrolling at the Screen's Edge

You can use the four DIRECTION functions (*Up, Down, Right, Left*) to cause scrolling. The screen (or current window) can scroll in all four directions. Although the editor does not wrap lines that are wider than the screen, you can have lines of text that are up to 250 characters wide. Use DIRECTION keys to scroll right and left when your text lines are wider than the screen or current window.

Unlike some editors, the Microsoft Editor does not automatically scroll by only one column or one line. Instead, the internal switches **hscroll** and **vscroll** control how fast the editor scrolls. For example, if **vscroll** (vertical-scroll switch) is set to 7, then the editor advances the screen position seven lines when you attempt to move the cursor off the bottom of the screen. See Chapter 7, "Using the TOOLS.INI File," for more information on these switches.

4.1.2 Scrolling a Page at a Time

The editor provides the *Ppage* (PGDN) and *Mpage* (PGUP) functions to move through a file more quickly than you can by using the DIRECTION keys to move one line or one column at a time.

The term "page" is defined as the amount of text that can be displayed in the current window or screen. To advance one page forward through a file, invoke the function *Ppage* (PGDN), which stands for "plus page."

The *Ppage* function can appear in a variety of commands that enable you to move even faster than a page at a time:

Command (and Default Keystrokes)	Description
Arg Ppage (ALT+A PGDN)	Moves the cursor forward to the end of the file
Arg numarg Ppage (ALT+A *numarg* PGDN)	Moves the cursor forward by the number of pages that you specify (*numarg*)
Arg streamarg Ppage (ALT+A *streamarg* PGDN)	Moves the cursor forward by the number of pages that you highlight on the screen (*streamarg*)

The function *Mpage* (PGUP), which stands for "minus page," is the direct inverse of *Ppage*, and it accepts the same syntax. For example, the command *Arg Mpage* (ALT+A PGUP) moves you backward to the beginning of the file.

4.1.3 Other File-Navigation Functions

The following functions also are useful for moving through a file:

Function (and Default Keystrokes)	Description
Pword (CTRL+RIGHT)	Moves the cursor forward (plus) one word
Mword (CTRL+LEFT)	Moves the cursor backward (minus) one word
Mark (CTRL+M)	Defines or moves to a marker

With the *Mark* function, you can define a marker or move to a marker. Markers constitute a special topic, which is discussed in Section 4.3, "Using File Markers."

4.2 Inserting, Copying, and Deleting Text

Often, you need to move, copy, or delete blocks of text. The Microsoft Editor is particularly powerful because it provides a variety of ways to define a block of characters.

For example, you can delete a highlighted box, a range of lines, or a stream of text between any two file positions. Sections 4.2.1–4.2.4 discuss how to work with blocks of text.

4.2.1 Inserting and Deleting Text

Chapter 2, "Edit Now," described how to use the *Sdelete*, *Insertmode*, and *Paste* functions to insert, delete, and move text. The *Sdelete* function is useful for working with single characters and with streams of text (*streamarg*). (A stream of text consists of a continuous sequence of characters between two positions in the file.) The following list presents some of the most common commands that use the *Sdelete* function:

Command (and Default Keystrokes)	Description
Sdelete (DEL)	Deletes the character at the cursor position. (This command does not join two lines of text, even if the cursor is at the end of the line.)
Arg Sdelete (ALT+A DEL)	Deletes all text from the cursor position to the end of the line, and then joins the current line of text with the next line.
Arg streamarg Sdelete (ALT+A *streamarg* DEL)	Removes all text between the two cursor positions. This command works with any cursor-movement argument.

To deal effectively with whole lines of text and with rectangular areas on the screen (*boxarg*), the Microsoft Editor provides the following functions:

Function (and Default Keystrokes)	Description
Ldelete (CTRL+Y)	Deletes a line of text or a *boxarg*
Linsert (CTRL+N)	Inserts a line of text or a *boxarg*

You can use these functions in commands that do not include an argument or prefix: *Ldelete* deletes the current line and *Linsert* inserts a blank line. These commands only insert or delete one line at a time, but you can use these commands repeatedly. You can also use these functions with arguments, as follows:

Command (and Default Keystrokes)	Description
Arg Ldelete (ALT+A CTRL+Y)	Deletes characters from the cursor position to the end of the line. Unlike *Arg Sdelete*, this command does not join lines.
Arg boxarg Ldelete (ALT+A *boxarg* CTRL+Y)	Deletes the highlighted rectangle (*boxarg*) on the screen, rather than a stream of text.
Arg boxarg Linsert (ALT+A *boxarg* CTRL+N)	Inserts a box of blank spaces into the indicated area. Text to the right of the cursor moves over as the box of blank spaces is inserted.

If you instead specify a *linearg*, the indicated number of blank lines is inserted.

4.2.2 Copying Text

To copy text without first deleting it, use the *Copy* function, which copies some range of text into an area of memory called the "Clipboard." Text in the Clipboard is inserted into the file when you invoke the *Paste* function. You invoke *Copy* with the + key. You can also invoke *Copy* with CTRL+INS. The following list presents different commands that use the *Copy* function:

Command (and Default Keystrokes)	Description
Arg boxarg Copy (ALT+A *boxarg* +)*	Copies the highlighted area into the Clipboard
Arg numarg Copy (ALT+A *numarg* +)*	Copies the specified number of lines into the Clipboard, beginning with the line that the cursor is on
Arg markarg Copy (ALT+A *markarg* +)*	Copies the stream of text between the specified marker and the cursor into the Clipboard

* The + key used is the one on the numeric keypad.

The *Paste* function (SHIFT+INS) is useful both for moving and for copying text. To move text, first delete it and then invoke *Paste* after moving the cursor to the destination.

See Section 4.3 for more information on markers.

4.2.3 Other Insert Commands

The following functions insert specific items at the current cursor position (each function is a complete command). These functions do not have preassigned keys; consult Chapter 6, "Function Assignments and Macros," for information on how to assign keys to functions.

Function	Description
Curdate	Inserts current date
Curday	Inserts current day of the week
Curfile	Inserts current file name
Curfileext	Inserts current file extension
Curfilenam	Inserts base name of current file
Curtime	Inserts current time
Curuser	Inserts name specified in **USER** environment variable

4.2.4 Reading a File into the Current File

The *Paste* function can be used in commands that read a file into the current file, as shown below:

Command (and Default Keystrokes)	Description
Arg Arg textarg Paste (ALT+A ALT+A *textarg* SHIFT+INS)	Reads the contents of the file specified by the *textarg*, and inserts these contents into the current file. The insertion occurs at the current cursor position.
Arg Arg !textarg Paste (ALT+A ALT+A *!textarg* SHIFT+INS)	Reads the output of the system-level command line given as the *textarg* into the current file. This command works similarly to the command given above.

4.3 Using File Markers

File markers help you move back and forth through large files. Once you have defined a file marker, you can move quickly to the location marked. You can also use a file marker as input to certain commands. For example, instead of moving the cursor to a marked location, you simply give the name of the marker.

The Microsoft Editor allows you to create any number of file markers. You identify each with a name consisting of alphanumeric characters.

Use the *Mark* function (CTRL+M) to create or go to a marker. The command *Mark* (CTRL+M with no argument) takes you back to the beginning of the file, just as *Arg Mpage* does. The command *Arg Mark* (ALT+A CTRL+M) moves you back to the previous cursor position. This last use of *Mark* is useful for switching back and forth quickly between two locations.

Some of the most powerful uses of the *Mark* function involve commands with arguments, as shown below:

Command (and Default Keystrokes)	**Description**
Arg numarg Mark (ALT+A *numarg* CTRL+M)	Moves the cursor to the line that you specify. The Microsoft Editor numbers lines beginning with the number 0, so the first line of the file is line 0, the second is line 1, and so forth.
Arg Arg textarg Mark (ALT+A ALT+A *textarg* CTRL+M)	Defines a marker at the current location. This command sets a marker which in turn can be used as input to other functions.
Arg textarg Mark (ALT+A *textarg* CTRL+M)	Moves the cursor directly to a marker that you have already defined as a *textarg*.

4.3.1 Functions That Use Markers

The following functions also make use of markers by accepting a previously defined marker name (a *markarg*) as an argument:

Function (and Default Keystrokes)	Description
Copy (+)*	Copies the argument into the Clipboard
Replace (CTRL+L)	Executes search and replace
Qreplace (CTRL+\)	Executes search and replace, with query for confirmation

* The + key used is the one on the numeric keypad.

If you specify a marker that the editor cannot find, the editor automatically checks the file listed in the **markfile** switch. See Chapter 7, "Using the TOOLS.INI File," for more information on the **markfile** switch.

4.3.2 Related Functions: Savecur and Restcur

The *Savecur* and *Restcur* functions have a purpose that is similar to *Mark*. The difference is that *Savecur* and *Restcur* do not take arguments. Use *Savecur* to save the current cursor position, and *Restcur* to return to that position later. With these two functions, you can save only one position at a time.

No keys are preassigned to *Savecur* or *Restcur*. See Chapter 6, "Function Assignments and Macros," for information on how to assign keys.

4.4 Searching and Replacing

The *Psearch* function (F3) directs the editor to conduct a forward search (also called a "plus search") for the next occurrence of the specified string. All searches take place from the current cursor position to the end of the file.

The most common uses of *Psearch* consist of the following commands:

Command (and Default Keystrokes)	Description
Arg textarg Psearch (ALT+A *textarg* F3)	Directs the editor to look for the string given as *textarg*. The editor scrolls the screen, if necessary, and moves the cursor to the next occurrence of *textarg* in the file.
Psearch (F3)	Directs the editor to look for the previous search string.
Arg Psearch (ALT+A F3)	Directs the editor to take the word at the current cursor position as the search string. (In other words, the search string consists of all characters from the cursor to the first blank or new line.)
Arg streamarg Psearch (ALT+A *streamarg* F3)	Directs the editor to take text highlighted on the screen as the search string.

You can search backward with *Msearch* (which stands for "minus search"). The *Msearch* function (F4) uses syntax identical to *Psearch*. Backward searches take place from the current cursor position to the beginning of the file.

4.4.1 Searching for a Pattern of Text

The commands described above search for an exact match of the string you specify. However, sometimes, you may want to search for a set of different strings: for example, any word that begins with "B" and ends with "ing."

You can search for a pattern of text by specifying a "regular expression." A regular expression is a string that specifies a pattern of text by using certain special characters. Chapter 5 describes the regular-expression character set and syntax in detail, with examples of use.

The command *Arg Arg textarg Psearch* (ALT+A ALT+A *textarg* F3) searches forward for a string that matches the regular expression specified as the *textarg*. The command *Arg Arg textarg Msearch* (ALT+A ALT+A *textarg* F4) searches backward for a string that matches the regular expression specified as the *textarg*.

4.4.2 Search-and-Replace Functions

To replace repeated occurrences of one text string by another, use the search-and-replace function *Replace* (CTRL+L). By default, the replacement happens from the cursor position to the end of the file. However, as described below, you can restrict the range over which the replacement happens.

No matter what command syntax you use with *Replace*, the editor reacts by prompting you for a search string and a replacement string, and then executing the search and replace. If you have used *Replace* or *Qreplace* before, the previous value of the search or replace string appears on the message line. To use the string displayed, press ENTER. To edit the string or enter a completely new string, use the text-editing commands given in Section 3.3.1, "Text Arguments." Note that the *Arg* function clears characters to the end of the line.

The commands *Replace* and *Arg Replace* are identical to each other, and execute replacement from the current cursor position the end of the file. You can also specify a range for the replacement by using one of the following commands:

Command	Default Keystrokes
Arg linearg Replace	ALT+A *linearg* CTRL+L
Arg numarg Replace	ALT+A *numarg* CTRL+L
Arg boxarg Replace	ALT+A *boxarg* CTRL+L
Arg markarg Replace	ALT+A *markarg* CTRL+L

If you specify a *numarg*, the replacement happens over the specified number of lines beginning with the current line. The argument *boxarg* defines a rectangular area within which the replacement takes place. And if you specify a *markarg*, then the replacement occurs in the box of text between the cursor position and the marker.

The *Replace* function is most efficient when you are sure that you want the replacement to be executed in every case. If you want to regulate how often the replacement occurs, use *Qreplace* (CTRL+\). This function is identical in every way to *Replace* and takes exactly the same syntax. The only difference is that *Qreplace* (short for "query replace") prompts you for confirmation before each replacement. *Qreplace* asks you to press Y for yes, N for no, or P, which causes replacement to proceed without further confirmation. The *Cancel* function (ESC) terminates the replacement.

The *Replace* and *Qreplace* functions both take regular expressions as search strings when you introduce the argument with *Arg Arg* instead of *Arg*. (See Chapter 5 for information on regular expressions.) Otherwise, syntax is identical, and the functions accept the same arguments.

4.5 Compiling

One of the strengths of the Microsoft Editor is that you can use it as a development environment. You can write a program and compile (or assemble) from within the editor. If the compile fails, you can make corrections to the source file at the same time that you view the errors and then compile again.

Ordinarily a compiler reports error output directly to the screen while you are outside of any editor. But when you compile from within the Microsoft Editor, it displays your errors by moving the cursor to the position where the error was found, and by reporting the corresponding message on the dialog line. This way, you can view the context of the error more easily and make corrections as soon as you see the errors.

The *Compile* function (SHIFT+F3) can be used to view errors as well as to compile. This *Compile* function appears in a variety of different commands, as shown in Sections 4.5.1–4.5.2.

4.5.1 Invoking Compilers and Other Utilities

When you run the editor in OS/2 protected mode, compiles run in the background and the editor beeps when the compile is completed. When you run the editor in real mode, you have to wait until the compile is completed before you can perform further editing commands.

With the Microsoft Editor's compile capability, you can invoke any program or utility you want, and specify any command-line options you want. To invoke a program directly, use one of the following commands:

Command	Default Keystrokes
Arg Arg textarg Compile	ALT+A ALT+A *textarg* SHIFT+F3
Arg Arg streamarg Compile	ALT+A ALT+A *streamarg* SHIFT+F3
Arg Compile	ALT+A SHIFT+F3

In the commands above, *textarg* is a system-level command line that you type in, and *streamarg* is a system-level command line that you highlight on the screen. Usually, it is most convenient to set your compile command once by setting the **extmake** switch and giving the command *Arg Compile* each time you want to compile.

The **extmake** text switch can be set to invoke a particular command line. A "text switch" is an internal string variable that affects the editor's behavior. See Chapter 7, "Using the TOOLS.INI File," for more information on text switches and how to set them.

Furthermore, the information on **extmake** in Chapter 7 describes how to make the editor sensitive to the file extension of your current file. For example, *Arg Compile* invokes one command line if the file has a **.C** extension, and another if it has an **.ASM** extension.

4.5.2 Viewing Error Output

To view error output from within the editor, you must use a compiler or assembler that outputs errors in one of the following formats:

filename row column: message
filename (row, column): message
filename (row): message
filename: row: message
"filename", row column: message

The Microsoft Editor, in turn, reads the error output directly, and responds by moving the cursor to each location where an error was reported while displaying the *message* on the dialog line. (The method for moving between error locations is described below.) The following programs output error messages in a format readable by the Microsoft Editor:

- Microsoft C Optimizing Compiler
- Microsoft Macro Assembler
- Microsoft Pascal Compiler 4.0
- Microsoft BASIC Compiler 6.0

Note

With the Pascal and BASIC compilers, you must use the **/Z** command-line option with either the **PL** or **BC** driver to generate error output that the Microsoft Editor can read. (The **extmake** switch, discussed in Chapter 7, "Using the TOOLS.INI File," uses the **/Z** option by default.)

When a compile fails and the compiler reports errors, the editor moves the cursor to the first error location reported. To view the next error, give the command *Compile* (SHIFT+F3). You can make any changes needed before advancing to the next error. If you are running in protected mode, you can move backward to the previous error by giving the command *Arg Meta Compile* (ALT+A F9 SHIFT+F3).

In protected mode, the editor processes all error messages through a pipe. In real mode, the editor redirects compile-error output to the file **M.MSG**. If the errors are not in readable format, then you can view errors by loading this file.

4.6 Using Windows

A "window" is a division of the screen that functions independently from other portions of the screen. When you have two or more windows present, each functions as a miniature screen; one window can view lines 5–15 while another window views lines 90–97. You can even use windows to view two or more files simultaneously. The cursor is never in more than one window. You can scroll each window independently.

Although windows are tiled, they can view overlapping areas of text. With multiple windows onto the same file, any change you make while in one window can affect what is displayed in another. Changes are reflected simultaneously in all windows that view the same area of altered text.

You can have up to eight windows on the screen, and you can create either horizontal or vertical divisions between windows. You move between windows by giving the command *Window* (F6 with no arguments). To create or merge a window, move the cursor to the row or column at which you want to create a new division, then give one of the following commands:

Command (and Default Keystrokes)	Description
Arg Window (ALT+A F6)	Creates a horizontal window (split at the cursor column)
Arg Arg Window (ALT+A ALT+A F6)	Creates a vertical window (split at the cursor row)
Meta Window (F9 F6)	Closes the current window by merging it with an adjacent window

Each window must have a minimum of 5 lines and 10 columns. If you try to create a window of a smaller size, then the command fails.

4.7 Working with Multiple Files

You can load a new file into the current screen or window with the *Setfile* function. Consider the following commands that use the *Setfile* function:

Command (and Default Keystrokes)	Description
Arg textarg Setfile (ALT+A *textarg* F2)	Loads the file specified in the *textarg*.
Setfile (F2)	Loads the previous file. You can use *Setfile* to move back and forth between two files.

An easier way to use to use *Setfile*, however, is to follow these steps:

1. Bring up the information file with the *Information* function (press SHIFT+F1).
2. Use the UP and DOWN keys to move to the name of a file.
3. Select the file that the cursor is on by giving the command *Arg Setfile* (press ALT+A F2).

The information file contains the names of all files that you have edited before, up to the limit specified by the **tmpsav** switch. (See Chapter 7, "Using the TOOLS.INI File," for more information about switches.) Active files—files that have been edited during this session—are listed with their current lengths.

When an old file is reloaded, the editor remembers cursor and window information from the last time you edited the file. The editor stores this information in the file **M.TMP**.

The *Arg textarg Setfile* command accepts wild-card characters (**?** matches any character and ***** matches any string) in the *textarg*. The command responds by displaying a list of files that match the *textarg*. You can then select a file by using the steps outlined above. For example, the following sequence causes the editor to list all files with a `.c` extension:

1. Invoke the *Arg* function (press ALT+A).
2. Type the following:

   ```
   *.c
   ```

3. Invoke the *Setfile* function (press F2).

Chapter 5
Regular Expressions

A regular expression is a special kind of string that you can use in a Microsoft Editor search command. Instead of matching only one string, a regular expression can match a number of different strings. For example, the regular expression `a[123]` matches any of the following strings:

```
a1
a2
a3
```

Regular expressions have their own particular syntax. This chapter explains that syntax and gives examples. Topics are covered in this order:

- Regular expressions as simple strings
- Special characters
- Matching method
- Tagged expressions
- Predefined regular expressions

You can use regular expressions with the **MEGREP** utility (see Appendix B, "Support Programs for the Microsoft Editor," for more information on this utility). You can also use regular expressions with the search functions *(Psearch, Msearch, Replace,* and *Qreplace).* Each of these functions recognizes a regular expression (rather than an ordinary text string) when you use *Arg Arg* to introduce the string.

5.1 Regular Expressions as Simple Strings

The power of regular expressions comes from the use of the special characters listed below. If you do not use these special characters, then a regular expression works as an ordinary text string.

\ { } ( ) [ ] ! ~ : ? ^ $ + * @ #

For example, the regular expression `match me precisely` matches only a literal occurrence of itself, because it contains no special characters.

5.2 Special Characters

The Microsoft Editor offers a rich set of pattern-matching capabilities. Most of the special characters described below have analogues in other editors and utilities that use regular expressions.

The list below describes some of the simpler special characters. The term *class* has a special meaning defined below. All other characters should be interpreted literally.

Expression	Description
\	Escape. Causes the editor to ignore the special meaning of the next character. For example, the expression `\?` matches `?` in the text file; the expression `\^` matches `^`; and the expression `\\` matches `\`.
?	Wildcard. Matches any single character. For example, the expression `a?a` matches `aaa`, `aBa`, and `a1a`, but not `aBBBa`.
^	Beginning of line. For example, `^The` matches the word `The` only when it occurs at the beginning of a line.
$	End of line. For example, `end$` matches the word `end` only when it occurs at the end of a line.
[*class*]	Character class. Matches any one character in the class. Use a dash (-) to specify ranges. For example, `[a-zA-Z0-9]` matches any character or digit, and `[abc]` matches `a`, `b`, or `c`.
[~*class*]	Noncharacter class. Matches any character not specified in the class.

The rest of the special characters are described in the following list, in which *X* is a placeholder that represents a regular expression that is either a single character or a group of characters enclosed in parentheses (()) or braces ({}). The placeholders *X1*, *X2*, and so on, represent any regular expression.

Expression	Description
*X**	Minimal matching. Matches zero or more occurrences of *X*. For example: the regular expression `ba*b` matches `baaab`, `bab`, and `bb`.
X+	Minimal matching plus (shorthand for *XX**). Matches one or more occurrences of *X*. The regular expression `ba+b` matches `baab` and `bab` but not `bb`.
X@	Maximal matching. Identical to *X**, except for differences in matching method explained in Section 5.3.
X#	Maximal matching plus. Identical to *X+*, except for differences in matching method explained in Section 5.3.
(*X1*!*X2*!...!*Xn*)	Alternation. Matches either *X1*, *X2*, and so forth. It tries to match them in that order, and switches from *Xi* to *Xi+1* only if the rest of the expression fails to match. For example, the regular expression `(ww!xx!xxyy)zz` matches `xxzz` on the second alternative and `xxyyzz` on the third.
~*X*	Not function. Matches nothing, but checks to see if the string matches *X* at this point, and fails if it does. For example, `^~(if!while)?*$` matches all lines that do not begin with `if` or `while`.
X^n	Power function. Matches exactly *n* copies of *X*. For example, `w^4` matches `wwww` and `(a?)^3` matches `a#aba5`.
{...}	Tagged expression. The exact use of tags is explained in Section 5.4. Characters within braces are treated as a group.
:*letter*	Predefined string. The list of predefined strings is given in Section 5.5.

The example below uses some of the special characters presented in this section. To find the next occurrence of a number (that is, a string of digits) beginning with a digit 1 or 2, perform the following sequence of keystrokes:

1. Invoke *Arg* twice (press ALT + A twice).
2. Type the following characters:

   [12][0-9]*

3. Invoke *Psearch* (press F3).

5.3 Matching Method

The "matching method" you use is significant only when you use a search-and-replace function. The term matching method refers to the technique used to match repeated expressions. For example does `a*` match as few or as many characters it can? The answer depends on the matching method. Two matching methods are available:

Method	Description
Minimal	The minimal method matches as few characters as possible in order to find a match. For example, `a+` matches only the first character in `aaaaaa`. However, `ba+b` matches the entire string `baaaaaab`, as it is necessary to match every occurrence of `a` in order to match both occurrences of `b`.
Maximal	The maximal method always matches as many characters as it can. For example, `a#` matches the entire string `aaaaaa`.

The significance of these two methods may not be apparent until you use search and replace. For example, if `a+` (minimal matching plus) is the search string and `EE` is the replacement string, then

```
aaaaa
```

is replaced with

```
EEEEEEEEEE
```

because each occurrence of `a` is immediately replaced by `EE`. However, if `a#` (maximal matching plus) is the search string, then the same string is replaced with

```
EE
```

because the entire string `aaaaa` is matched at once and replaced with `EE`.

5.4 Tagged Expressions

Like matching method, tagged expressions have no effect except when you use search-and-replace functions. Tagged expressions are useful because you may want to manipulate text rather than simply replace it with a fixed string. For example, suppose you wanted to find all occurrences of *hexdigits*H and replace them with strings of the form **16#***hexdigits*. Tagged expressions enable you to do just these kinds of operations.

The Microsoft Editor first looks for a character string that matches the entire regular expression given. Then, each substring of characters that corresponds to an expression within braces ({}) is tagged. You can tag up to nine such substrings. A tagged expression can then be generated in the replacement string by the use of the expression

$*n*

in which *n* is a digit from 0 to 9. The first tagged expression (going from left to right) is referred to as **$1**, the second as **$2**, and so forth up to **$9**. The expression **$0** always refers to the entire matched string.

To return to the original example, you can search for strings of the form *hexdigits*H by specifying the following regular expression:

```
{[0-9a-fA-F]+}H
```

and then specifying this replacement string:

```
16#$1
```

Note that # is not a special character when it appears in the replacement string. The result is that the Microsoft Editor searches for any occurrence of one or more hexadecimal digits (digits 0–9 and the letters a–f) followed by the letter **H**. The editor then replaces each such string by preserving the actual digits, but adding the prefix **16#**. For example, the string `1a000H` is replaced with the string `16#1a000`.

5.5 Predefined Regular Expressions

The following expressions are defined in Table 5.1 for your convenience. You can use them by entering : *letter* in a regular expression.

Table 5.1
Predefined Expressions

Letter	Meaning	Description
:a	[a–zA–Z0–9]	Alphanumeric
:b	([\t]#)	White space
:c	[a–zA–Z]	Alphabetic
:d	[0–9]	Digit
:f	([~"\[\]\:<|>+=;,.]#)	Portion of a file name
:h	([0–9a–fA–F]#)	Hexadecimal number
:i	([a–zA–Z_$][a–zA–Z0–9_$]@)	C-language identifier
:n	([0–9]#[0–9]@![0–9]@.[0–9]#![0–9]#)	Number
:p	(([a–z]\:!)(\\!)(:f(:f!)\\)@:f(.:f!))	Path
:q	("[~"]@"!'[~']@')	Quoted string
:w	([a–zA–Z]#)	Word
:z	([0–9]#)	Integer

Chapter 6
Function Assignments and Macros

Function assignments and macros give the Microsoft Editor flexibility and power. Function assignments allow you to assign any editing function to a new keystroke. The new keystroke can be identical to one you have used with other editors, or you can assign a keystroke that makes sense only to you.

Using macros saves time by reducing the amount of typing you do. A macro consists of a list of arguments and functions; once defined, the entire list of arguments and functions can be assigned to a single keystroke. The Microsoft Editor's macros also support conditional execution, so that you can use the results of a function (its return value) to determine what other functions to invoke.

This chapter covers the following topics:

- Using the **MESETUP** program
- Assigning functions within the editor
- Creating macros within the editor

6.1 Using the MESETUP Program

The **MESETUP** program installs the editor files in a directory that you specify and assigns the editing functions to a predefined set of keystrokes. The editor provides configurations that use keys similarly to the way they are used in several popular editors:

- Microsoft Quick languages/WordStar
- BRIEF
- Epsilon
- Default (which is used if none of the others are selected)

See the **README.DOC** file for instructions on using the **MESETUP** program.

6.2 Assigning Functions within the Editor

Assigning an editing function to a new keystroke from within the editor is easy. And once a new assignment has been made, you can use that keystroke to invoke the function at any time during the editing session.

Take into account the following important points when assigning functions to keystrokes:

1. The function assignments you make during the editing session are lost when you exit from the editor. See Chapter 7, "Using the TOOLS.INI File," for information on more permanent function assignments.

2. Assigning a function to a new keystroke does not change any other keystrokes to which the function was previously assigned. See Section 6.2.3 for information on removing assignments.

3. Only one function may be assigned to a given keystroke at a time; therefore, you are not able to use the keystroke to invoke any function which was previously assigned to it.

6.2.1 Making Function Assignments

To assign a function to a keystroke, issue the *Arg textarg Assign* command, where *textarg* uses the following syntax:

functionname:keystroke

Here, *keystroke* may be any of the following:

1. Numeric keys: 0 through 9

2. Letter keys: A through Z

3. Function keys: F1 through F12

4. Punctuation keys: ` ~ , . < > / ? ; ' " [ ] { } \ | - = _ +

5. Named keys: HOME, END, LEFT, RIGHT, UP, DOWN, PGUP, PGDN, INS, DEL, BKSP, TAB, ESC

6. Numeric-keypad keys: +, –, and 0 through 9. To assign a function to the 4 key on the numeric keypad, enter the following as the *keystroke*:

   ```
   NUM4
   ```

7. Combinations:
   a. ALT combined with items 1–5
   b. CTRL combined with items 2–6
   c. SHIFT combined with items 3–6

For example, the function *Savecur* is assigned to the keystroke CTRL+B in the following way:

1. Invoke the *Arg* function (press ALT+A).
2. Enter the function and keystroke as the *textarg* by typing the following:

   ```
   Savecur:CTRL+B
   ```

3. Invoke the *Assign* function (press ALT+= by holding down the ALT key and pressing the = key).

From this point on, pressing CTRL+B invokes the *Savecur* function and saves the current cursor position.

6.2.2 Viewing Function Assignments

The *Help* function shows you what function assignments are in effect at any time during the editing session. Invoking the *Help* function (by pressing F1) causes all of the editing functions to be listed in alphabetical order on the screen along with the keys to which they are assigned. You can scroll through this information as you would through any file. Use the *Setfile* function (F2) to return to your original file.

6.2.3 Removing Function Assignments

If you choose to remove a function assignment, assign the keystroke to the function **Unassigned** using the *Arg textarg Assign* command. The argument *textarg* uses the following syntax:

Unassigned:*key*

Here, *key* is the keystroke you want to remove.

For example, to remove the keystroke CTRL+A from any function, perform the following steps:

1. Invoke the Arg function (press ALT+A).
2. Enter the function name as **Unassigned** and the keystroke by typing the following:

   ```
   Unassigned:CTRL+A
   ```

3. Invoke the *Assign* function (press ALT+=).

After these steps are carried out, pressing CTRL+A does not invoke any functions.

6.2.4 Making Graphic Assignments

Assigning the **Graphic** function to a keystroke lets you press the keystroke to insert it literally into the file. For example, to insert a form-feed character in the file whenever CTRL+L is pressed, follow these steps:

1. Invoke the *Arg* function (press ALT+A).
2. Enter the function **Graphic** and the keystroke as the *textarg* by typing the following:

   ```
   Graphic:CTRL+L
   ```

3. Invoke the *Assign* function (press ALT+=).

Like the **Graphic** function, the *Quote* function lets you insert a literal character. However, the *Quote* function must be used every time that the keystroke is pressed. The **Graphic** function needs to be assigned only once during an editing session.

6.3 Creating Macros within the Editor

A macro is a series of functions and text arguments that you can execute with a simple keystroke. The functions may be any valid editor functions. The text arguments may serve as input to functions or as text that is to be entered into the file. Macros allow you to use the results of a function (its return value) to determine what other functions to invoke.

Take into account the following important points when creating macros:

- Macros that you create during the editing session are lost when you exit from the editor. See Chapter 7, "Using the TOOLS.INI File," for information on a more permanent way of creating macros.

- The maximum number of macros that may be defined at any one time is 1024.

6.3.1 Entering a Macro

Enter a macro by using the *Arg textarg Assign* command, where *textarg* uses the syntax described below:

macroname:={*function* | "*text*"}...

Each function must be previously defined and *macroname* must be a unique name. Spaces separate the individual functions and arguments within commands. Double quotes surround text arguments.

For example, the following macro scrolls the window down by 11 lines and places the cursor in column 1:

```
Halfscreen:=Meta Up Arg "11" Plines Begline
```

Since a macro definition must be contained on one line, it may be necessary to break up a macro function into several smaller functions as shown in the example below. The smaller functions can then be grouped together and given a name and assigned to a keystroke. Each of the following lines would be entered one at a time using the *Arg textarg Assign* command.

```
Head1:=Arg "3" Linsert "/**********************"
Head2:=Newline "** Routine:"
Head3:=Newline "**********************/" Up Endline Right
Header:=Head1 Head2 Head3
```

Macros may contain text only and not use functions at all, as in the following example:

```
Proc:="procedure();"
```

When invoked, this macro inserts the text `procedure();` into the file at the current cursor position. However, before you can directly invoke the macro, you need to assign it to a keystroke.

6.3.2 Assigning a Macro to a Keystroke

To invoke a macro, it is necessary to assign it to a keystroke. The procedure is similar to that described in Section 6.2.1 for assigning a function to a keystroke, except that you enter the name of your macro instead of an editing-function name. For example, the following steps assign ALT+H to the macro named `Header`:

1. Invoke the *Arg* function (press ALT+A).
2. Enter the macro name and keystroke as the *textarg* by typing the following:

   ```
   Header:ALT+H
   ```

3. Invoke the *Assign* function (press ALT+=).

6.3.3 Using Macro Conditionals

Macro conditionals let you alter the order that functions are invoked within the macro. An editing function returns a TRUE value if the function is successful, or a FALSE value if it fails. For example, a cursor-movement function fails if the cursor does not move or if an invalid argument is used. Table 6.1 provides a complete list of functions and return values.

Table 6.1
Editor Functions and Return Values

Function	Returns TRUE	Returns FALSE
Arg	Always	Never
Argcompile	Compile successful	Bad argument/compiler not found
Assign	Assignment successful	Invalid assignment
Backtab	Cursor moved	Cursor at left margin
Begline	Cursor moved	Cursor not moved
Cancel	Always	Never
Cdelete	Cursor moved	Cursor not moved
Compile	Compile successful	Bad argument/compiler not found
Copy	Copy successful	Bad argument
Down	Cursor moved	Cursor not moved

Table 6.1 (*continued*)

Function	Returns TRUE	Returns FALSE
Emacscdel	Cursor moved	Cursor not moved
Emacsnewl	Always	Never
Endline	Cursor moved	Cursor not moved
Execute	Last command successful	Last command failed
Exit	No return condition	No return condition
Help	Always	Never
Home	Cursor moved	Cursor not moved
Information	Always	Never
Initialize	Initialization successful	Bad argument
Insertmode	Insert mode now on	Insert mode now off
Lasttext	Function successful	Bad argument
Ldelete	Line-delete successful	Bad argument
Left	Cursor moved	Cursor not moved
Linsert	Line insert successful	Bad argument
Mark	Definition/move successful	Bad argument/not found
Meta	*Meta* now on	*Meta* now off
Mlines	Movement occurred	Bad argument
Mpage	Movement occurred	Bad argument
Mpara	Movement occurred	Bad argument
Msearch	String found	Bad argument/string not found
Mword	Cursor moved	Cursor not moved
Newline	Always	Never
Paste	Always	Never
Pbal	Balance successful	Bad argument/not balanced
Plines	Movement occurred	Bad argument
Ppage	Movement occurred	Bad argument
Ppara	Movement occurred	Bad argument
Psearch	String found	Bad argument/string not found
Pword	Cursor moved	Cursor not moved
Qreplace	At least one replacement	String not found/invalid pattern
Quote	Always	Never

Table 6.1 *(continued)*

Function	Returns TRUE	Returns FALSE
Refresh	File read in/deleted	Canceled, bad argument
Replace	At least one replacement	String not found/invalid pattern
Restcur	Position previously saved with *Savecur*	Position not saved with *Savecur*
Right	Cursor over text of line	Cursor beyond end of line
Savecur	Always	Never
Sdelete	Delete successful	Bad argument
Setfile	File-switch successful	Bad argument
Setwindow	Window-change successful	Bad argument
Shell	Shell successful	Bad argument/program not found
Sinsert	Insert successful	Bad argument
Tab	Cursor moved	Cursor not moved
Undo	Always	Never
Up	Cursor moved	Cursor not moved
Window	Successful split, join, or move	Any error

The return values listed above can be used with the conditionals shown in Table 6.2 to invoke functions conditionally.

Table 6.2
Macro Conditionals

Conditional	Description
:>*label*	Defines a label that can be referenced by any of the other macro conditionals.
=>*label*	Causes a direct transfer to *label*. If *label* is omitted, then the current macro is exited.
->*label*	Causes a direct transfer to *label* if the previous function returned the FALSE condition. If *label* is omitted, then the current macro is exited.
+>*label*	Causes a direct transfer to *label* if the previous function returned the TRUE condition. If *label* is omitted, then the current macro is exited.

For example, the following macro erases all characters from the current line:

```
Blankline:=Endline :>back Sdelete Left +>back
```

The macro executes the commands in the following order:

1. `Endline` causes the cursor to move to the end of the line.
2. `:>back` defines a label in the macro command.
3. `Sdelete` erases the character under the cursor.
4. `Left` moves the cursor one character to the left. If the cursor moves (it is not in column 1), then the return condition is true, otherwise it is false.
5. If the return condition in step 4 is true, `+>back` transfers control back to the command following the label `back`.

Steps 3–5 continue until all characters have been deleted and the cursor is in column 1.

Chapter 7
Using the TOOLS.INI File

You can place statements in the **TOOLS.INI** file to modify function assignments, set switches, and define macros for the Microsoft Editor. Each time the editor is started, it loads all of the statements from the appropriate sections of the **TOOLS.INI** file (unless the **/D** option is used). This saves you the trouble of entering the same function assignments, switch settings, and macro definitions in every editing session.

This chapter explains the **TOOLS.INI** file, as follows:

- Contents of the **TOOLS.INI** file (comments, function assignments, macros, switch settings)
- Location of statements within the **TOOLS.INI** file (using tags)

Note

The editor checks the directories listed in the **INIT** operating-system environment variable for the location of the **TOOLS.INI** file. For example, if the **TOOLS.INI** file is in the directory `C:\INIT`, then place the following statement in your **AUTOEXEC.BAT** file:

```
SET INIT=C:\INIT
```

7.1 Using Comments

Comments in the **TOOLS.INI** file serve the same purpose as comment lines found in most program source files; they provide documentation for how and why things are done. The Microsoft Editor assumes everything on the line following a semicolon is a comment and ignores it.

The comment in this example explains the macro's function and the keystroke assignment that follows it:

```
; Assign Ctrl+S to a macro for saving the current file.
Save:=Arg Arg Setfile
Save:Ctrl+S
```

7.2 Assigning Functions to Keystrokes

Function assignments allow you to assign a function to a particular keystroke, so that you can invoke the function by pressing the keystroke. You can change the default set of assignments by placing function-assignment statements in the **TOOLS.INI** file. The syntax follows:

functionname:*keystroke*

Here, *functionname* is the function you want assigned to the keystroke *key*. Section 6.2.1, "Making Function Assignments," lists the keys that you can assign to editing functions and macros.

In the following example, the statement assigns the `Window` function to the key ALT+W:

```
Window:Alt+W
```

7.3 Defining Macros

As discussed in Chapter 6, "Function Assignments and Macros," a macro is made up of arguments and predefined functions and can be executed with a single keystroke. To enter a macro in the **TOOLS.INI** file, use the following syntax:

macroname:={*function* | "*text*"}...

The argument *macroname* must be a unique name within each tagged section of the **TOOLS.INI** file. Each function used in the definition must be previously defined. A space separates individual functions and arguments in the definition; double quotes surround the arguments.

The example below shows how a multiline macro definition might appear in the **TOOLS.INI** file.

```
; This macro indents the first line of each
; paragraph in the file by five spaces.
Indent:=Meta Begline Arg Right Right Right Right Right Sinsert
Inpara:=Mark :>Repeat Indent Ppara Endline Left +>Repeat
Inpara:Alt+P
```

7.4 Setting Switches

Three types of switches control the action of the Microsoft Editor: numeric switches, Boolean switches, and text switches. You can set these switches in one of two ways:

1. Set them from within the editor using the *Arg textarg Assign* command, where *textarg* uses the syntax described in the following sections.

2. Enter them in the **TOOLS.INI** file, one per line, using the syntax described in the following sections.

7.4.1 Numeric Switches

Numeric switches allow you to give values to features such as screen colors, tabs, and other controls. The syntax for setting a numeric switch is as follows:

switchname:*numericvalue*

In the first example below, the **hscroll** switch is set to the value of 20, that is, the window is shifted left or right by 20 columns when the cursor moves out of the window. The second example sets the color of error messages to light yellow text on a black background.

```
hscroll:20
errcolor:0E
```

The numeric switches **errcolor**, **fgcolor**, **hgcolor**, **infcolor**, and **stacolor** all specify colors for various types of text. The first digit of the value specifies the background color, while the second digit specifies the text color. Table 7.1 lists the colors and their associated hexadecimal values. It should be noted that when specifying the background color, the values 8–F specify the same colors as 0–7 respectively, except that the text flashes.

Table 7.1
Colors and Numeric Values

Color	Value
Black	0
Blue	1
Green	2
Cyan	3
Red	4
Magenta	5
Brown	6
Light Gray	7
Dark Gray	8
Light Blue	9
Light Green	A
Light Cyan	B
Light Red	C
Light Magenta	D
Light Yellow	E
White	F

Table 7.2 lists each of the numeric switches, along with its purpose and default value.

Table 7.2
Numeric Switches

Numeric Switch	Description (and Default Value)
entab	Controls the degree to which the Microsoft Editor converts multiple spaces to tabs when editing a file. A value of 0 means that tabs are not used to represent white space, 1 means that all multiple spaces outside of quoted strings are converted, 2 means that all multiple spaces are converted to tabs. (Default value: 1)
errcolor	Controls the color used for error messages. The default is red text on a black background. (Default value: 04)
fgcolor	Controls the color used for the editing window. The default is light gray text on a black background. (Default value: 07)
height	Controls the number of lines that the Microsoft Editor uses in the editing window, not including the dialog and status lines. This is useful with a nonstandard display device; Enhanced Graphics Adapter (EGA) in 43-line mode on the IBM PC uses a value of 41, and Video Graphics Array (VGA) in 50-line mode uses a value of 48. (Default value: 23)
hgcolor	Controls the color for highlighted text. The default is black text on a light gray background. (Default value: 70)
hike	Specifies the ending line position of the cursor when the cursor is moved directly by editing functions. (Default value: 4)
hscroll	Controls the number of columns shifted left or right when the cursor is scrolled out of the editing window. (Default value: 10)
infcolor	Controls the color used for informative text. The default is brown text on a black background. (Default value: 06)
maxmsg	Controls the maximum number of messages retained in the *Compile* function's message buffer. This switch works in OS/2 protected mode only. To set this switch, place it in **TOOLS.INI** in a section tagged **[M-10.0]**. (Default value: 10)
noise	Controls the number of lines counted at a time when searching or loading a file. This value is displayed in the lower-right corner of the screen and may be turned off by setting **noise** to 0. (Default value: 50)

Table 7.2 *(continued)*

Numeric Switch	Description (and Default Value)
rmargin	Controls the right column margin used in wordwrap mode. Any character typed to the right of this margin causes a line break. Word-wrap mode is turned on and off with the **wordwrap** switch. (Default value: 72)
stacolor	Controls the color used for the status-line information. The default is cyan text on a black background. (Default value: 03)
tabdisp	Specifies the ASCII value of the character used to expand tabs. Normally, a space is used, but a graphic character can be used to show exactly where tabs are located. (Default value: 32)
tabstops	Controls the number of spaces between each logical tab stop for the editor. (Default value: 4)
tmpsav	Controls the maximum number of files about which information is kept between editing sessions. These are the most recently edited files, and each file will be listed only once. When you exit from the editor, the position of the cursor and window are saved, along with the layout of multiple windows if any. When you begin editing one of these files again, the screen starts up as you left it. (Default value: 20)
traildisp	Specifies the ASCII value of the character to be displayed as trailing spaces. Note that this switch has no effect unless the **trailspace** switch is turned on. (Default value: 0)
undocount	Controls the number of edit functions that you may undo. (Default value: 10)
vmbuf	Controls the number of 2K pages allocated in real memory to buffer the virtual-memory file, M*xxx*.VM. This switch works in OS/2 protected mode only. (Default value: 128)
vscroll	Controls the number of lines shifted up or down when the cursor is scrolled out of the editing window. The *Mlines* and *Plines* functions also use this value. (Default value: 7)
width	Controls the width of the display mode for displays that are capable of showing more than 80 columns. (Default value: 80)

7.4.2 Boolean Switches

Boolean switches turn certain editor activities on or off. Turn on a switch by entering the switch name followed by a colon; turn it off by typing **no** followed by the name and a colon. The syntax is summarized below:

[[**no**]]*switchname:*

In the first example below, the **case** switch is set or turned on, which results in case being significant in a search operation. In the second example the **askrtn** switch is reset or turned off, which results in the editor returning from a *Shell* command without prompting you.

```
case:
noaskrtn:
```

Table 7.3 provides a complete list of Boolean switches, including the purpose and default value for each.

Table 7.3
Boolean Switches

Boolean Switch	Description (and Default Value)
askexit	Prompts for confirmation when you exit from the editor. (Default value: Off)
askrtn	Prompts you to press ENTER when returning from a *Shell* command. (Default value: On)
autosave	Saves the current file whenever you switch away from it. If this switch is off, you must specify when you want the file to be saved. (Default value: On)
case	Considers case to be significant for search-and-replace operations. For example if **case** is on, the string `Procedure` is not found as a match for the string `procedure`. (Default value: Off)
displaycursor	Shows a position on the status line in the (*row,column*) format. When off, the position listed is that of the upper-left corner. When on, the current cursor position is given. (Default value: Off)
enterinsmode	Starts the editor up in insert mode as opposed to overtype mode. (Default value: Off)
savescreen	Saves and restores the DOS screen (for use with the *Push* and *Exit* functions). (Default value: On)
shortnames	Allows you to specify an alternate file by giving only the base name. (Default value: On)
softcr	Attempts to indent based upon the format of the surrounding text when you invoke the *Newline* or *Emacsnewl* functions. (Default value: On)
trailspace	Remembers trailing spaces in text. (Default value: Off)
wordwrap	Breaks lines of text when you edit them beyond the margin specified by **rmargin**. (Default value: Off)

7.4.3 Text Switches

Text switches specify a string that modifies the action of the editor in some way. The syntax is shown below:

switchname:*textvalue*

In the example below, the **backup** switch is set so that no backup is performed.

```
backup:none
```

Table 7.4 lists the text switches, the function of each, and a default value, if any.

Table 7.4
Text Switches

Text Switch	Description (and Default Value)
backup	Determines what happens to the old copy of a file when it is edited. A value of **none** specifies that no backup operation is to be performed and the old file is overwritten. A value of **undel** specifies that the old file is to be moved so that **UNDEL.EXE** can retrieve it. A value of **bak** specifies that the file name of the old version of the file will be changed to **.BAK**. (Default value: **undel**)

extmake	Associates a command line with a particular file extension for use by the *Compile* function. The text after the switch has this form:
	extmake:*extension commandline*
	Here, *extension* is the extension of the file to match, and *commandline* is a command line to be executed. If there is a **%s** in the command line, it is replaced with the name of the current file or with the *textarg* in the *Arg textarg Compile* command. This is the only switch that may appear more than once in the **TOOLS.INI** file; there is a separate line for each extension.
	For example, you have the following lines in **TOOLS.INI**:
	`extmake:bc /Z %s`
	`extmake:for fl /c %s`
	`extmake:pas pl /c /h %s`
	`extmake:asm masm -Mx %s;`
	`extmake:c cl /c /Zep /D LINT_ARGS %s`
	`extmake:text make %s`
	You also have a file named `foo`. The command *Arg* `foo` *Compile* invokes
	`make foo`
	This in turn invokes a compiler and linker. See the documentation for the Microsoft Program Maintenance Utility (**MAKE**) for more information.

load	Specifies the name of a C-extension executable file to be loaded.

Table 7.4 *(continued)*

Text Switch	Description (and Default Value)
markfile	Specifies the name of the file the Microsoft Editor searches when looking for a marker that is not in the in-memory set. This file can be created using the **CALLTREE** program discussed in Appendix B, "Support Programs for the Microsoft Editor," or by entering lines of the following form: *markername filename line column* Here, *line* and *column* specify the position in the file *filename* where the marker *markername* appears.
readonly	Specifies the DOS command that is invoked when the Microsoft Editor attempts to overwrite a read-only file. The current file name is appended to the command, as shown in the following example: `readonly:attrib -r` This command removes the read-only attribute from the current file so the file can be overwritten. If no command is specified, you are prompted to enter a new name under which to save the file.

7.5 Creating Sections with Tags

Tags divide the **TOOLS.INI** file into sections. All statements are associated only with the tag that they immediately follow. This allows programs other than the Microsoft Editor, **MAKE**, to use this file for configuration information. It also allows you to load only a certain section of statements by using the *Arg textarg Initialize* command. The tag must use the following syntax:

[M-*text***]**

The value of *text* is the *textarg* that you use to initialize the editor with the statements following this tag. A blank line precedes the tag.

In the example below, there are two tagged sections, one for use with C programs and one for Pascal programs.

```
[M-Pascal]
; Insert a Pascal Header
Header:=Arg "1" Linsert Newline "{ Pascal Program:"
Header:Alt+h

[M-C]
; Insert a C Header
Header:=Arg "1" Linsert Newline "/* C Program:"
Header:Alt+h
```

With this text in the **TOOLS.INI** file, you can use ALT+H to insert one of the two headers into your file, depending on which tag you use to initialize the editor. For example, to insert the C header, follow these steps:

1. Invoke the Arg function (press ALT+A).

2. Enter the name of the tagged section to load (type C).

3. Invoke the Initialize function (press F10).

   The editor reads the tagged section from the **TOOLS.INI** file.

4. Insert the C header (press ALT+H).

When the Microsoft Editor is started, the tagged sections are loaded in the following order:

1. Information specific to the operating system.

   Depending upon the operating system you are working under, one of the following tagged sections is loaded (if present):

   - **[M-3.20]** (MS-DOS)
   - **[M-10.0]** (OS/2 protected mode)
   - **[M-10.0R]** (OS/2 real mode)

   This provides a way of automatically setting the **vmbuf** and **maxmsg** switches when running in protected mode. With the DOS version tag, you should insert the version number you are using. You can specify more than one version by using a tag like **[M-3.20 M-3.30]**, which works with either version 3.20 or 3.30.

2. Information used for all editing sessions.

   All of the statements in the **[M]** section are loaded.

3. Information specific to the display.

   Depending on the video display you are using, one of the following tagged sections is loaded (if present):

   - **[M-mono]**
   - **[M-cga]**
   - **[M-ega]**
   - **[M-vga]**
   - **[M-viking]**

   You can also put statements for setting the screen dimensions and colors in these tagged sections.

Chapter 8
Programming C Extensions

C extensions offer the most powerful technique for customizing the Microsoft Editor. The term "C extension" refers to a C-language module containing new editing functions that you program. The module can also define new switches. Your functions can be attached to a key, given arguments, and used in macros just as intrinsic editing functions are. Any switches that you define can be set just as intrinsic editing switches are, and your switches can be used by the functions you define.

If you already understand the C programming language, you do not need to learn a new, specialized language to build your functions. C extensions let you use the full power of the C language: data structures, control-flow structures, and C operators. Furthermore, the C-generated code is compiled, not interpreted. Therefore your functions are fast.

Note

This chapter assumes that you already know how to program in C. Before you read the chapter, make sure that you understand the following C-language programming concepts: functions, pointers, structures, and unions. You also need to know how compile and link a C source file.

You can also write extensions with **MASM** if you simulate the C memory model specified in Section 8.5.1, "Compiling in Real Mode." However, this chapter is primarily addressed to C programmers.

This chapter develops C-extension concepts gradually. The first time you read the chapter, you should read the sections in sequential order:

- Requirements
- How C extensions work
- Writing the C extension
- How to use the low-level editing functions

- Compiling and linking
- A C-extension sample program

8.1 Requirements

To create C extensions, you need to have the following files and software present in your current directory (or directories listed in the **PATH** or **INCLUDE** environment variables, as appropriate):

- The Microsoft C Optimizing Compiler, Version 4.0 or later
- The Microsoft Overlay Linker, Version 3.60 or later, or the OS/2 version of the linker, or the Microsoft Segmented-Executable Linker Version 5.01
- **EXTHDR.OBJ** (supplied with the editor)
- **EXT.H** (supplied with the editor)
- **SKEL.C** (a template that you can replace with your own code)

You need a minimum of 150K of available memory for the editor to load a C extension at run time.

8.2 How C Extensions Work

A C-extension module is similar in the following respects to an OS/2 or Windows dynamic-link library:

- There is no function called **main** in your module. Instead, you use certain names and structures that the editor recognizes.
- You compile and link to create an executable file, but this executable file is separate from the main program, **M.EXE**.
- The editor loads your executable file into memory at run time. The editor uses a table-driven method for enabling your module to call functions within **M.EXE**.

Once your executable file is loaded, it resides in memory along with **M.EXE**. The editor can call your functions, and your functions can call the Microsoft Editor's low-level functions that perform input and output.

The following list summarizes the overall process of developing and using a C extension:

1. Compile a C module with a special memory-model option, then link the resulting object file to create an executable file.

   You also link in the object file **EXTHDR.OBJ** to the beginning of your executable file. This object file contains a special table that enables your functions to call functions within the editor effectively.

2. Start up the Microsoft Editor. Set the internal **load** switch to look for the executable file you created. (As discussed in Chapter 7, the **load** switch can be set in the **TOOLS.INI** file or manually with the *Assign* function).

   The editor loads your executable file into memory.

3. As soon as the executable file is loaded, the editor calls the function **WhenLoaded**, which is a special function that your module must define.

   At the same time, the editor examines the table **cmdTable**, which is an array of structures that your module must declare. The editor examines this table in order to recognize the editing functions that you have created. The table contains function names and pointers to functions.

4. You can assign keys to call your functions. Assign a key manually or in the **WhenLoaded** function, then press the assigned key. You can also call an editing function indirectly by placing it in a macro and calling the macro.

5. When you invoke a C-extension function, the editor responds by calling your module.

6. Your editing function is executed. It calls the Microsoft Editor's low-level functions in order to read from the text file, output to the text file, and print messages.

8.3 Writing a C Extension

To create a successful C extension, you need to follow these guidelines:

1. Check the **README.DOC** file to see what functions you can call from the standard C run-time library.

   A technical problem prevents library compatibility: to work with the Microsoft Editor, you must compile with **SS** not equal to **DS**. (The C compiler gives your module its own default data area, but your module shares the editor's stack. Therefore your stack-segment and data-segment registers are not equal.) Since standard Microsoft C libraries assume **SS** equal to **DS**, you cannot use some library functions.

2. Include the file **EXT.H**.

   This file declares all the structures and types that are required to establish an interface to the editor.

3. Include the standard items that are described in Section 8.3.1, "Required Objects." Then compile and link as directed in Section 8.5, "Compiling and Linking."

8.3.1 Required Objects

A C-extension module must have at minimum three items with the names given below:

Object Name	Description
swiTable	An array of structures that declares internal switches that you wish to create
cmdTable	An array of structures that declares editing functions that you have coded
WhenLoaded	A function that the editor calls as soon as the C-extension module is loaded

Each of these items can be as short or as long as you wish. Each table can be as short as a single row of entries. The **WhenLoaded** function can return immediately, or it can perform useful initialization tasks such as assigning keys to functions or printing a message.

8.3.2 The Switch Table

The switch table, **swiTable**, consists of a series of structures, in which each structure describes a switch you wish to create. The table ends with a structure that has all null (all zero) values. Though you may choose not to create any switches, the table must still be present. The simplest table allowed is therefore

```
struct swiDesc swiTable[] =
{
     { NULL, NULL, NULL }
};
```

The structure type **swiDesc** is defined in **EXT.H**. This structure contains the following three fields that define a switch for the editor to recognize:

1. A pointer to the name of the switch.

2. A pointer to the switch itself or to a function. If the switch is Boolean, then this field must point to the switch (an integer which assumes the value –1 or 0). If the switch is text, then this field must point to a function, as explained below. If the switch is numeric, then this field points to either an integer or to a function, depending on the value of the third field.

3. A flag that indicates the type of switch: either **SWI_BOOLEAN**, **SWI_NUMERIC**, or **SWI_SPECIAL**.

If the third field has value **SWI_SPECIAL**, then the second field must be a pointer to a function of type **int pascal**. You define this function in your code. Each time the value of the switch changes, the editor calls your function and passes the updated value in a character string. Your function should declare exactly one parameter: a far pointer to a character.

The table may have any number of rows (each row being a structure), and must at least include the final row of all null values. Here is an example of a table that creates a numeric switch with a default value of 27:

```
int n = 27;

struct swiDesc swiTable [] =
{
     { "newswitch", &n, SWI_NUMERIC } ,
     { NULL, NULL, NULL }
}
```

8.3.3 The Command Table

The command table, **cmdTable**, is similar to the table **swiTable** in its construction. Each "row" of the table consists of a structure that describes an editing function that you want the editor to recognize. The last row must contain all null values. The simplest table allowed is the following:

```
struct cmdDesc cmdTable[] =
{
     { NULL, NULL, NULL, NULL }
}
```

Usually you will want to declare at least one new editing function. The structure type **cmdDesc** is defined in **EXT.H**. This structure contains the following four fields that make an editing function recognizable to the editor:

1. A pointer to the name of the function as it will be used within the editor. This name could appear in assignments and macros.

2. The address of the function itself. Give the function name, but do not follow it with parentheses.

3. A field used internally by the editor. Always declare this field as null.

4. The type of the function. Function types are described below and define what type of argument the function will accept.

Here is an example of a command table that declares a function that takes no arguments:

```
struct cmdDesc cmdTable[] =
{
     { "newfun", newfun, NULL, NOARG } ,
     { NULL, NULL, NULL, NULL }
}
```

In the fourth field of the command table, use one or more of the values described below:

Value	Description
KEEPMETA	Does not take the *Meta* prefix. The function reserves *Meta* for next function.
CURSORFUNC	Executes cursor movement only. Highlighting and the *Arg* function are not affected. The function does not take arguments.

WINDOWFUNC	Window-movement function. Highlighting is not affected.
NOARG	Accepts absence of *Arg* prefix.
TEXTARG	Accepts a text argument.
BOXSTR	Accepts a one-line box argument (in other words, a *streamarg*). The string of text highlighted is passed as a text argument.
NULLARG	Accepts *Arg* without requiring an argument.
NULLEOL	Accepts *Arg* without requiring an argument. The function is passed pointer to *textarg* consisting of an ASCIIZ string from the cursor to the end of the line.
NULLEOW	Accepts *Arg* without requiring an argument. The function is passed pointer to *textarg* consisting of an ASCIIZ string from the cursor to the end of the word (next white space).
LINEARG	Accepts a *linearg*. If the editor detects a *linearg*, function is passed the beginning line of the range and the ending line of the range.
STREAMARG	Accepts any kind of cursor movement. The function is passed the beginning point of the range and the ending point of the range.
BOXARG	Accepts a *boxarg*. If the editor detects a *boxarg*, the function is passed the line and column boundaries of the region.
NUMARG	Accepts a *numarg*. Information is passed as a *linearg*; in other words, the function is passed a range of lines.
MARKARG	Accepts a *markarg*. Information is passed as a *streamarg*; in other words, the function is passed beginning and ending point of range defined by the cursor position and the marker.

In the descriptions above, the term "ASCIIZ string" refers to a string of characters terminated by a zero (or null) byte. The descriptions also refer to the passing of information to the function; you'll see how the function receives information in Section 8.3.5, "Writing the Editing Function."

You can combine the function types with binary or (|). For example, you can specify a function that accepts a *boxarg*, *linearg*, or *numarg* as:

```
BOXARG | LINEARG | NUMARG
```

8.3.4 The WhenLoaded Function

The function **WhenLoaded** takes no arguments and can return immediately if you want. However, you must include the function because the editor expects it to be present. The simplest version of **WhenLoaded** is:

```
WhenLoaded()
{
        return;
}
```

In Section 8.4, "Calling Low-Level Editing Functions," you'll learn how to call functions that assign keys to functions and print a message on the message line. These functions are often useful to call from within **WhenLoaded**.

8.3.5 Writing the Editing Function

This section describes how to declare an editing function and how to use information that is passed to the function from the editor. The editing function must return type **flagType**, which is an integer that takes values true (–1) or false (0) and is defined in the file **EXT.H**. Editing functions are declared with Pascal calling conventions and must be of type **EXTERNAL**. The sample function `Skel` is declared as follows:

```
#define TRUE  -1
#define FALSE 0

flagType pascal EXTERNAL Skel (argData, pArg, fMeta)
unsigned int argData;
ARG far *pArg;
flagType fMeta;
{
return TRUE;
}
```

The parameter list is described below:

Parameter	Description
argData	The value of the keystroke used to invoke the function. This parameter is generally not used.
pArg	A pointer to a structure that contains almost all the information passed by the editor. This structure is discused in detail below.
fMeta	An integer that describes whether or not a *Meta* prefix is present. This integer has value true (–1) if *Meta* is present, and value false (0) if not.

The parameter **pArg** points to a structure whose first element is always **argType**. The argument type returned in this structure uses the same values listed in Section 8.3.3, "The Command Table." Thus, you could test for the presence of a *numarg* with the following code:

```
if (pArg->argType == NUMARG) {

            /* take appropriate action for numarg */
        }
```

The rest of the structure consists of a union of structures. The C data-type union is necessary here; it enables the editor to pass data in a variety of different formats. The exact format depends on which member of the union is used. In any case, the data is passed to the same area of memory.

The declaration of the **ARG** structure in the file **EXT.H** is as follows:

```
struct argType {
    int   argType;
    union {
        struct  noargType       noarg;
        struct  textargType     textarg;
        struct  nullargType     nullarg;
        struct  lineargType     linearg;
        struct  streamargType   streamarg;
        struct  boxargType      boxarg;
    } arg;
}

typedef struct argType ARG;
```

The editor uses one of the structures in the union to return information about arguments. The choice of structures depends on the type of argument. For example, if the **argType** element is equal to **LINEARG**, the editor returns information in the structure `pArg->arg.linearg`.

Consult the file **EXT.H** to see how each structure is declared. For example, the **textarg** structure type is declared as follows:

```
struct textargType {
    int   cArg;
    LINE  y;
    COL   x;
    char  far *pText;
}
```

In the structure above, `cArg` contains an integer equal to the number of times *Arg* was invoked. The variables `y` and `x` are integers that give the cursor position, and `pText` points to the actual string text. The following code initializes variables `row` and `col`, and copies the *textarg* into a buffer:

```
LINE row;
COL  col;
int  i;
char far *p, buffer[81];

row = pArg->arg.textarg.y;
col = pArg->arg.textarg.x;
p = pArg->arg.textarg.pText;
for (i = 0; (c = *p) != NULL; i++)
        buffer[i] = c;
```

In another example, if `pArg->argType` is equal to type **NULLARG**, then you can initialize `row` and `col` as follows:

```
LINE row;
COL  col;

row = pArg->arg.nullarg.y;
col = pArg->arg.nullarg.x;
```

8.3.6 Putting It All Together

Here is a listing of the source module **SKEL.C**, which provides you with the basic template of a C extension. This code does nothing, but it is recognized by the Microsoft Editor as logically correct. You can makes use of this template by using your own function names and inserting your own statements. Before you can write useful code, however, you first need to read Section 8.4, "Calling Low-Level Editing Functions."

```
#include "ext.h"

#define  TRUE   -1
#define  FALSE  0
#define  NULL   ((char *) 0)

flagType pascal EXTERNAL Skel (argData, pArg, fMeta)
unsigned int argData;
ARG far *pArg;
flagType fMeta;
{
    return TRUE;
}

struct swiDesc swiTable[] = {
    { NULL, NULL, NULL }
};

struct cmdDesc cmdTable[] = {
    { "skel", Skel, 0,  NOARG } ,
    { NULL, NULL, NULL, NULL }
};

WhenLoaded ()
{
    return TRUE;
}
```

8.4 Calling Low-Level Editing Functions

The functions presented in this section cannot be called directly by the user. However, they can be called by higher-level editing functions to carry out specific tasks such as reading a line from a file, replacing or inserting a character, printing messages, and deleting or inserting text. These functions are used within the Microsoft Editor itself and are made available to be called by functions in a C extension.

Note

All pointers that you pass (such as character pointers) need to refer to data that are declared externally; in other words, do not pass pointers to strings that you declare locally. Because **SS** does not equal **DS**, the low-level function will not properly find stack data, such as a local (or "automatic") variable.

This section serves as a guide to the most commonly used low-level functions. You can begin writing C extensions by using the functions presented here. Later you can consult the file **EXT.DOC** for a complete listing of all low-level functions.

Sections 8.4.1–8.4.3 present groups of functions by covering the following topics:

- Reading from a file
- Writing to a file
- Initialization functions

8.4.1 Reading from a File

This section presents functions that you can call to scan a file (either the current file or any other that you specify).

8.4.1.1 The FileNameToHandle Function

To read or write to a file (including the current file), you must first call the **FileNameToHandle** function, which returns a handle to the named file. The function is declared as follows:

PFILE pascal FileNameToHandle (*pname*, *pShortName*)
char **pname*, **pShortName*;

The *pname* parameter points the file name. If *pname* points to a zero-length string, then the function returns a handle to the current file. Unless *pShortName* is a null pointer, the editor searches its list of current files (files that have been edited in this session) for a path name that includes the name pointed to by *pShortName*. If there is a match, the function uses the full path name found.

For example, the following code returns a handle to the current file:

```
PFILE  curfile;

curfile = FileNameToHandle("", NULL);
```

8.4.1.2 The GetLine Function

The **GetLine** function provides the principal means for reading text from a file.

int pascal GetLine (*line*, *buf*, *pfile*)
LINE *line*;
char far **buf*
PFILE *pfile*;

The function reads a specified line of text, and copies the line into a character-string buffer pointed to by *buf*. The *line* parameter is an integer that contains a line number. The *pfile* parameter is a pointer returned by **FileNameToHandle.**

The following example reads the line of text which includes the initial cursor position:

```
PFILE cfile;
char  buffer[256];

cfile = FileNameToHandle("", NULL);
GetLine(pArg->arg.nullarg.y, buffer, cfile)
```

8.4.1.3 The FileLength Function

The **FileLength** function is useful for doing global file operations, in which you need to know when you are at the last line. The function takes a pointer to a file handle as input, returns an integer, and is declared as follows:

LINE pascal FileLength (*pFile*)
PFILE *pfile*;

The following example stores the length of the current file in the variable n:

```
n = FileLength (cfile);
```

8.4.2 Writing to a File

This section presents functions that are useful for altering a file by replacing, inserting, or deleting text.

8.4.2.1 The Replace Function

The **Replace** function inserts or replaces characters one at a time; it is declared as follows:

flagType pascal Replace (*c*, *x*, *y*, *pFile*, *fInsert*)
char *c*;
COL *x*;
LINE *y*;
PFILE *pFile*;
flagType *fInsert*;

The *c* parameter contains the new character. The *x* and *y* parameters indicate the file position, by column and line, where the edit is to take place. The *pFile* parameter is a file handle returned by the **FileNameToHandle** function. To specify insertion, set *fInsert* to true (–1). To specify replacement, set *fInsert* to false (0). The function returns true (–1) if the edit is successful.

For example, the following code inserts the word "Hello" at line `y` and column `x` of the current file:

```
#define  TRUE  -1
char     *p;
PFILE    cfile;  /* handle to current file */
.
.
.
cfile = FileNameToHandle("", NULL);  /* initialize cfile */
for (p = "Hello"; *p; p++, y++)
        Replace(*p, x, y, cfile, TRUE);
```

8.4.2.2 The PutLine Function

The **PutLine** function replaces a line of text; it is declared as follows:

void pascal PutLine (*line*, *buf*, *pfile*)
LINE *line*;
char far **buf*;
PFILE *pfile*;

The parameter *buf* points to the string that contains the new line of text. This string should terminate with a null value, but it should not contain a new-line character. The editor takes care of inserting a new-line character at the proper position in the file. The parameter *line* contains the line number at which the replacement it to take place. Line numbers start at 0; if *line* has the value 0 then the new line of text is inserted at the beginning of the file.

The following code replaces the first line of the current file with the string pointed to by `buffer`:

```
[PutLine (0, buffer, cfile);
```

8.4.2.3 The CopyLine Function

The **CopyLine** line function can be used either to copy a group of lines from one area to another or to insert a blank line. The function is declared as follows:

void pascal CopyLine (*pFileSrc, pFileDst, yStart, yEnd, yDst*)
PFILE *pFileSrc, pFileDst*;
LINE *yStart, yEnd, yDst*;

The *pFileSrc* and *pFileDst* parameters are file handles. If *pFileSrc* is null (0), then the function inserts a blank line. Otherwise, the function inserts lines from *yStart* to *yEnd*. Lines are inserted directly before *yDst*. For example, the following code inserts a blank line at the beginning of the file:

```
CopyLines(NULL, cfile, NULL, NULL, 0);
```

8.4.2.4 The DelStream Function

The **DelStream** function deletes a stream of text beginning with a starting coordinate and going up to but not including the ending coordinate. The function is declared as follows:

void pascal DelStream (*pfile, xStart, yStart, xEnd, yEnd*)
PFILE *pfile*;
COL *xStart, xEnd*
LINE *yStart, yEnd*;

The *xStart* and *yStart* parameters are the beginning coordinates; the *xEnd* and *yEnd* parameters are the ending coordinates. The coordinates are all integers.

The following example deletes the stream of text beginning with line 2 column 3, up to but not including line 5 column 4.

```
DelStream (cfile, 3,2,4,5);
```

8.4.3 Initialization Functions

The low-level functions in this section are typically called by the **WhenLoaded** function, but they can be called by editing functions as well.

8.4.3.1 The SetKey Function

The **SetKey** function assigns an editing function to a key, and is declared as follows:

flagType pascal SetKey (*name*, *p*)
char far **name*, **far** **p*;

The *name* parameter points to a string containing the name of the function, and the *p* parameter points to a string that names the key. The rules for naming the key are the same as those given in Chapter 6, "Function Assignments and Macros." The function returns true (–1) if the assignment is successful.

The following code assigns the CTRL+X key to the newly defined function `NewFunc`:

```
SetKey("NewFunc", "ctrl+x");
```

8.4.3.2 The DoMessage Function

The **DoMessage** function outputs a message on the dialog line and returns the number of characters written.

int pascal DoMessage (*pStr*)
char far **pStr*;

The *pStr* parameter points to the message you want to write.

The following example outputs a message on the dialog line:

```
DoMessage("Hello, world.");
```

8.4.3.3 The BadArg Function

The **BadArg** function reports an error message stating that the user's argument was not accepted. Note that usually you do not need to call this function because the editor looks at the type of your function as declared in **cmdDesc** (**TEXTARG**, **STREAMARG**, and so forth) and rejects commands with the wrong type of argument. The function is declared as follows:

flagType pascal BadArg (void)

8.5 Compiling and Linking

After you've written your C module following the guidelines in the last few sections, you're ready to compile and link. The procedures for compiling and linking in protected mode are slightly different from compiling and linking in real mode. Sections 8.5.1–8.5.2 consider both environments.

8.5.1 Compiling in Real Mode

To create a C extension for real mode, follow these two steps:

1. Compile with command line options **/Gs** and **/Asfu**. These options establish the proper memory model and calling convention, and are mandatory. (If you are programming in **MASM**, use near code and far data segments, in which **SS** is not assumed equal to **DS**.) For example:

   ```
   CL /c /Gs /Asfu myext.c
   ```

2. Link with the command-line options **/NOD** and **/NOI**. Linking with **/NOD** is important because it prevents the linker from linking in standard libraries. Always link the file **EXTHDR.OBJ** first. For example:

   ```
   LINK /NOI /NOD exthdr.obj myext.obj, myext;
   ```

When you use the **CL** driver, you can accomplish both steps in one command line:

```
CL /Gs /Asfu /Femyext exthdr myext.c /link /NOD /NOI
```

When you correctly compile and link your C-extension module, you produce an executable file. You cannot execute this file directly from DOS. However, the Microsoft Editor can load the file into memory and use the functions that your module defines.

To use the C extension, make sure that your executable file is in the current directory or in a directory listed in the **PATH** environment variable. After you start up the Microsoft Editor, set the **load** switch to make the editor load your C extension. For example, after you have created the file **MYEXT.EXE**, you could place the following statement in the **TOOLS.INI** file:

```
load:myext.exe
```

The editor responds by automatically loading your C-extension module into memory whenever the editor checks the **TOOLS.INI** file for initialization.

8.5.2 Compiling in Protected Mode

To compile and link a protected-mode C extension, follow the instructions above for real mode, except in two respects:

1. Use the **/G2** and **/Lp** options when you compile. (The **/Lp** option is not required unless you compile a protected-mode application from within real mode.) The example in the previous section would therefore change to

   ```
   CL /c /Gs /Asfu /G2 /Lp myext.c
   ```

2. Instead of linking to produce an executable file, you link to produce a **.DLL** file (a dynamic-link application). Specify **SKEL.DEF** as the module-definition file, and place the resulting **.DLL** file in one of the directories listed in the **LIBPATH** directive in your **CONFIG.SYS** file. You may want to edit the **SKEL.DEF** file, to change the library name specified.

8.6 A C-Extension Sample Program

The following C-extension sample program features one simple function named `Upper`, which accepts a simple *streamarg* or *textarg*. (As explained earlier in the chapter, the **BOXSTR** function type accepts a one-line stream of text highlighted on the screen.) The function responds by replacing characters in the file, beginning at the cursor position, with characters from the *textarg* that have been converted to uppercase letters.

```
#include "ext.h"
#define TRUE  -1
#define FALSE  0
#define NULL  ((char *) 0)

flagType pascal EXTERNAL Upper (argData, pArg, fMeta)
unsigned int argData;
ARG far *pArg;
flagType fMeta;
{
     LINE row;          /* coordinates in file */
     COL  col;
     int c;             /* replacement character */
     char far *p;       /* pointer to textarg */
     PFILE cfile;       /* pointer to file handle */

     cfile = FileNameToHandle("", NULL)  /* get current file */
     row = pArg->arg.textarg.y;         /* load coordinates */
     col = pArg->arg.textarg.x;
     p = pArg->arg.textarg.pText;
     for (; *p; p++, col++) {      /* for each char in textarg */
          c = *p;                        /* get character */
          if (c >= 'a' && c <= 'z')      /* convert to upper */
               c += 'A' - 'a';
          Replace (c, col, row, cfile, FALSE);  /* put in file */
     }
     return TRUE;
}

struct swiDesc swiTable [] = {
     { NULL, NULL, NULL }
};

struct cmdDesc cmdTable [] = {
     { "Upper", Upper, 0, BOXSTR | TEXTARG },
     { NULL, NULL, NULL, NULL }
};

WhenLoaded()
{
     SetKey("Upper", "alt+u");
     DoMessage("Upper function now loaded.");
}
```

Appendix A
Reference Tables

A.1 Categories of Editing Functions

Table A.1 lists the editing functions by category and gives a brief description of each function.

Table A.1
Summary of Editing Functions by Category

Cursor Movement	Description
Backtab	Moves cursor left to previous tab stop
Begline	Moves cursor left to beginning of line
Down	Moves cursor down one line
Endline	Moves cursor to right of last character of line
Home	Moves cursor to upper-left corner of window
Left	Moves cursor left one character
Mark	Moves cursor to specified position in file
Mlines	Moves cursor back by lines
Mpage	Moves cursor back by pages
Mpara	Moves cursor back by paragraphs
Mword	Moves cursor back by words
Newline	Moves cursor down to next line
Plines	Moves cursor forward by lines
Ppage	Moves cursor forward by pages
Ppara	Moves cursor forward by paragraphs
Pword	Moves cursor forward by words
Restcur	Restores cursor position saved with *Savecur*
Right	Moves cursor right one character
Savecur	Saves cursor position for use with *Restcur*
Tab	Moves cursor right to next tab stop
Up	Moves cursor up one line

Table A.1 *(continued)*

Windows	Description
Setwindow	Redisplays window
Window	Creates, removes, and moves between windows
Searching/Replacing	**Description**
Msearch	Searches backward
Psearch	Searches forward
Qreplace	Replaces with confirmation
Replace	Replaces without confirmation
Moving/Copying Text	**Description**
Copy	Copies lines into the Clipboard
Ldelete	Deletes lines into the Clipboard
Paste	Inserts text from the Clipboard
Sdelete	Deletes stream of text, including line breaks
Inserting/Deleting Text	**Description**
Cdelete	Deletes character to left, excluding line breaks
Curdate	Inserts current date (e.g. 27-Jun-1987)
Curday	Inserts current day (Sun...Sat)
Curfile	Inserts name of current file
Curfileext	Inserts extension of current file
Curfilenam	Inserts base name of current file
Curtime	Inserts current time (e.g. 13:45:55)
Curuser	Inserts current user name
Emacscdel	Deletes character to left, including line breaks
Emacsnewl	Starts new line, breaking current line
Ldelete	Deletes lines into the Clipboard
Linsert	Inserts blank lines
Pbal	Balances parentheses and brackets
Sdelete	Deletes stream of text, including line breaks
Sinsert	Inserts blanks, breaking lines if necessary

Table A.1 (*continued*)

File Operations	Description
Argcompile	Performs the *Arg Compile* command
Compile	Performs compilation and reviews error messages
Refresh	Rereads file, discarding edits
Setfile	Switches to alternate file
Miscellaneous	**Description**
Arg	Introduces an argument or function
Assign	Assigns value to a configuration variable
Cancel	Cancels current operation
Execute	Executes an editor function
Exit	Exits the editor
Help	Displays current key assignments
Information	Displays information about an editing session
Initialize	Rereads initialization file
Insertmode	Toggles insert mode on and off
Lasttext	Recalls the last *textarg* entered
Quote	Treats next character literally
Shell	Runs the command shell
Undo	Reverses the effect of the last editing change

A.2 Key Assignments for Editing Functions

Table A.2 lists the editing functions and the assigned keys for each of the configurations provided with the setup program.

Table A.2
Function Assignments

Function	Default	Quick/ WordStar	BRIEF	EPSILON
Arg	ALT+A	ALT+A	ALT+A	CTRL+U or CTRL+X
Argcompile	F5	F5	ALT+F10	F5
Assign	ALT+=	ALT+=	F7	F1
Backtab	SHIFT+TAB	SHIFT+TAB	SHIFT+TAB	SHIFT+TAB
Begline	HOME	HOME or CTRL+QS	HOME	CTRL+A
Cancel	ESC	ESC	ESC	CTRL+C
Cdelete	CTRL+G	CTRL+G	BKSP	---
Compile	SHIFT+F3	SHIFT+F3	CTRL+N	SHIFT+F3
Copy	CTRL+INS or press + (keypad)	CTRL+INS	+ (keypad)	ALT+W
Curdate	---	---	---	---
Curday	---	---	---	---
Curfile	---	---	---	---
Curfileext	---	---	---	---
Curfilenam	---	---	---	---
Curtime	---	---	---	---
Curuser	---	---	---	---
Down	DOWN or CTRL+X	DOWN or CTRL+X	DOWN	DOWN or CTRL+N
Emacscdel	BKSP	BKSP	---	BKSP or CTRL+H
Emacsnewl	ENTER	ENTER	---	ENTER
Endline	END	END or CTRL+QD	END	CTRL+E

Table A.2 (*continued*)

Function	Default	Quick/ WordStar	BRIEF	EPSILON
Execute	F7	F10	F10	ALT+X
Exit	F8	ALT+X	ALT+X	F8
Help	F1	F1	ALT+H	F10
Home	CTRL+HOME	CTRL+HOME	CTRL+HOME	HOME
Information	SHIFT+F1	SHIFT+F1	ALT+B	SHIFT+F1
Initialize	SHIFT+F8	ALT+F10	SHIFT+F10	ALT+F10
Insertmode	INS or CTRL+V	INS or CTRL+V	ALT+I	CTRL+V
Lasttext	CTRL+O	ALT+L	ALT+L	ALT+L
Ldelete	CTRL+Y	CTRL+Y	ALT+D	CTRL+K
Left	LEFT or CTRL+S	LEFT	LEFT	LEFT or CTRL+B
Linsert	CTRL+N	CTRL+N	CTRL+ENTER	CTRL+O
Mark	CTRL+M	ALT+M	ALT+M	CTRL+@
Meta	F9	F9	F9	F9
Mlines	CTRL+W	CTRL+W	ALT+U	CTRL+W
Mpage	PGUP or CTRL+R	PGUP or CTRL+R	PGUP	PGUP or ALT+V
Mpara	CTRL+PGUP	CTRL+PGUP	CTRL+PGUP	ALT+UP
Msearch	F4	F4	ALT+F5	CTRL+R
Mword	CTRL+LEFT or CTRL+A	CTRL+LEFT	CTRL+LEFT	CTRL+LEFT or ALT+B
Newline	---	---	ENTER	---
Paste	SHIFT+INS	SHIFT+INS	INS	CTRL+Y or INS
Pbal	CTRL+[	CTRL+[	CTRL+[	CTRL+[
Plines	CTRL+Z	CTRL+Z	CTRL+Z	CTRL+Z
Ppage	PGDN or CTRL+C	PGDN or CTRL+C	PDGN	PDGN
Ppara	CTRL+PGDN	CTRL+PGDN	CTRL+PDGN	ALT+DOWN
Psearch	F3	F3	F5	F4 or CTRL+S
Pword	CTRL+RIGHT or CTRL+F	CTRL+RIGHT or CTRL+F	CTRL+RIGHT	CTRL+RIGHT or ALT+F
Qreplace	CTRL+\	ALT+F3	F6	ALT+F3 or ALT+5 or ALT+8
Quote	CTRL+P	ALT+Q	ALT+Q	CTRL+Q

Table A.2 (*continued*)

Function	Default	Quick/ WordStar	BRIEF	EPSILON
Refresh	SHIFT+F7	ALT+R	CTRL+]	ALT+R
Replace	CTRL+L	CTRL+L	SHIFT+F6	---
Restcur	---	---	---	---
Right	RIGHT or CTRL+D	RIGHT or CTRL+D	RIGHT	RIGHT or CTRL+F
Savecur	---	---	---	---
Sdelete	DEL	DEL	DEL or press – (keypad)	DEL or CTRL+D
Setfile	F2	F2	ALT+N	F2
Setwindow	CTRL+]	CTRL+]	F2	CTRL+]
Shell	SHIFT+F9	SHIFT+F9	ALT+Z	ALT+Z
Sinsert	CTRL+J	ALT+INS	CTRL+INS	ALT+INS
Tab	TAB	TAB	TAB	TAB or CTRL+I
Undo	ALT+BKSP	ALT+BKSP	* (keypad)	CTRL+BKSP
Up	UP or CTRL+E	UP or CTRL+E	UP	UP or CTRL+P
Window	F6	F6	F1	ALT+PGDN

A.3 Comprehensive Listing of Editing Functions

Table A.3 gives a comprehensive listing of the editing functions and syntax for each command. Default keystrokes, if available, are given in parentheses.

Table A.3
Comprehensive List of Functions

Function (and Default Keystrokes)	Syntax	Description
Arg (ALT+A)	*Arg*	Introduces a function or an argument for a function.
Argcompile (F5)	*Argcompile*	Performs the *Arg Compile* command.
Assign (ALT+=)	*Arg Assign*	Treats the text from the initial cursor position to the end of the line (not including the line break) as a function assignment or macro definition.
	Arg boxarg Assign	Treats each line of the *boxarg* as an individual function assignment or macro definition.
	Arg linearg Assign	Treats each line as a separate function assignment or macro definition, ignoring blank lines.
	Arg streamarg Assign	Treats the highlighted text as a function assignment or macro definition.
	Arg textarg Assign	Treats *textarg* as a function assignment or macro definition.
	Arg ? Assign	Displays the current function assignments for all functions and macros.
Backtab (SHIFT+TAB)	*Backtab*	Moves the cursor to the previous tab stop. Tab stops are defined to be every *n*th character, where *n* is defined by the **tabstops** switch.
Begline (HOME)	*Begline*	Places the cursor on the first nonblank character on the line.
	Meta Begline	Places the cursor in the first character position of the line.
Cancel (ESC)	*Cancel*	Cancels the current operation in progress.

Table A.3 *(continued)*

Function (and Default Keystrokes)	Syntax	Description
Cdelete (CTRL+G)	*Cdelete*	Deletes the previous character, excluding line breaks. If the cursor is in column 1, *Cdelete* moves the cursor to the end of the previous line. If issued in insert mode, *Cdelete* deletes the previous character, reducing the length of the line by 1; otherwise it deletes the previous character and replaces it with a blank. If the cursor is beyond the end of the line when the function is invoked, the cursor is moved to the immediate right of the last character on the line.
Compile (SHIFT+F3)	*Compile*	Reads the next error message and tries to parse it into file, row, column, and message. If it is successful, the editor reads in the file, places the cursor on the appropriate row and column, and displays the message on the dialog line. The utility **MEGREP.EXE**, Microsoft C, and the Microsoft Macro Assembler generate output compatible with this format.
	Meta Compile	Reads error messages and advances to the first message that does not refer to the current file.
	Arg Compile	Compiles and links the current file. The command and arguments used to compile the file are specified by the **extmake** switch according to the extension of the file.
	Arg streamarg Compile *Arg textarg Compile*	Compile and link the file specified by *streamarg* or *textarg*. The command and arguments used to compile the file are specified by the **extmake** switch according to the extension of the file.

Table A.3 (*continued*)

Function (and Default Keystrokes)	Syntax	Description
	Arg Arg streamarg Compile *Arg Arg textarg Compile*	Invoke the specified text as a program. The program is assumed to display its errors in the following format: file row column message This is often used to find a particular text pattern in a series of files by using **MEGREP.EXE**. See Appendix B, "Support Programs for the Microsoft Editor," for more information.
	Arg Meta Compile	Backs up to display the previous message, up to a maximum number of messages specified by the **maxmsg** switch.
Copy (CTRL+INS, or press + on keypad)	*Copy*	Copies the current line into the Clipboard.
	Arg Copy	Copies text from the initial cursor position to the end of the line and places it into the Clipboard. Note that the line break is not picked up.
	Arg boxarg Copy *Arg linearg Copy* *Arg streamarg Copy* *Arg textarg Copy*	Copy the specified text into the Clipboard.
	Arg numarg Copy	Copies the specified number of lines into the Clipboard, starting with the current line.
	Arg markarg Copy	Copies the range of text between the cursor and the location of the file marker into the Clipboard. The copied text is treated as a *streamarg*, *boxarg*, or *linearg* depending on the relative positions of the initial cursor position and the file-marker location.
Curdate	*Curdate*	Inserts the current date at the cursor in the format of Jun-27-1987.

Table A.3 (*continued*)

Function (and Default Keystrokes)	Syntax	Description
Curday	*Curday*	Inserts the current day at the cursor in the format of Sun...Sat.
Curfile	*Curfile*	Inserts the fully-qualified pathname of the current file at the cursor.
Curfileext	*Curfileext*	Inserts the extension of the current file at the cursor.
Curfilenam	*Curfilenam*	Inserts the base name of the current file at the cursor.
Curtime	*Curtime*	Inserts the current time at the cursor in the format of 13:45:55.
Curuser	*Curuser*	Inserts the name of the current user, using the **MAILNAME** environment variable, at the cursor.
Down (DOWN or CTRL+X)	*Down*	Moves the cursor down one line. If this would result in the cursor moving out of the window, the window is adjusted downward by the number of lines specified by the **vscroll** switch or less if in a small window.
	Meta Down	Moves the cursor to the bottom of the window without changing the column position.
Emacscdel (BKSP)	*Emacscdel*	Performs similarly to *Cdelete*, except that at the beginning of a line while in insert mode, *Emacscdel* deletes the line break between the current line and the previous line, joining the two lines together.
Emacsnewl (ENTER)	*Emacsnewl*	Performs similarly to *Newline*, except that when in insert mode, it breaks the current line at the cursor position.
Endline (END)	*Endline*	Moves the cursor to the immediate right of the last nonblank character on the line.

Table A.3 (*continued*)

Function (and Default Keystrokes)	Syntax	Description
	Meta Endline	Moves the cursor one character beyond the column corresponding to the rightmost edge of the window.
Execute (F7)	*Arg Execute*	Treats the line from the initial cursor position to the end as a series of Microsoft-Editor commands and executes them.
	Arg linearg Execute *Arg streamarg Execute* *Arg textarg Execute*	Treat the specified text as Microsoft-Editor commands and execute them, similar to the way macros operate.
Exit (F8)	*Exit*	Saves the current file. If multiple files were specified on the command line, the editor advances to the next file. Otherwise the editor quits and returns control to the operating system.
	Meta Exit	Performs similarly to *Exit*, except that the current file is not saved.
	Arg Exit	Performs similarly to *Exit*, except that if multiple files are specified on the command line, the editor exits without advancing to the next file.
	Arg Meta Exit	Performs similarly to *Arg Exit*, except that the editor does not save the current file.
Help (F1)	*Help*	Lists the editing functions and current key assignments.
Home (CTRL+HOME)	*Home*	Places the cursor in the upper-left corner of the current window.
Information (SHIFT+F1)	*Information*	Saves the current file and *Setfiles* to an information file that contains a list of all files in memory along with the current set of files that you have edited. The size of this list is controlled by the **tmpsav** switch, which has a default value of 20.

Table A.3 *(continued)*

Function (and Default Keystrokes)	Syntax	Description
Initialize (SHIFT+F8)	*Initialize*	Reads all the editor statements from the **[M]** section of **TOOLS.INI**.
	Arg Initialize	Reads the editor statements from the **TOOLS.INI** file, using the continuous string of nonblank characters, starting with the initial cursor position, as the tag name.
	Arg streamarg Initialize *Arg textarg Initialize*	Read all the editor statements from the **[M]** section and the **[M-***streamarg***]** or **[M-***textarg***]** section of **TOOLS.INI**.
Insertmode (INS or CTRL+V)	*Insertmode*	Toggles the insert-mode switch. The status of the insert-mode switch can be seen on the status line; if insert mode is on, `insert` appears on the status line. While in insert mode, each character that is entered is inserted at the cursor position, shifting the remainder of the line one position to the right. Overtype mode replaces the character under the cursor with the one that is entered.
Lasttext (CTRL+O)	*Lasttext*	Recalls the last *textarg*. This function is the same as invoking the *Arg* function and then retyping the previous *textarg*.
Ldelete (CTRL+Y)	*Ldelete*	Deletes the current line and places it into the Clipboard.
	Arg Ldelete	Deletes text, starting with the initial cursor position through the end of the line, and places it into the Clipboard. Note that it does not join the current line with the next line.
	Arg boxarg Ldelete *Arg linearg Ldelete* *Arg streamarg Ldelete*	Delete the specified text from the file and place it into the Clipboard.

Table A.3 (*continued*)

Function (and Default Keystrokes)	Syntax	Description
Left (LEFT or CTRL+S)	*Left*	Moves the cursor one character to the left. If this would result in the cursor moving out of the window, the window is adjusted to the left by the number of columns specified by the **hscroll** switch or less if in a small window.
	Meta Left	Moves the cursor to the leftmost position in the window on the same line.
Linsert (CTRL+N)	*Linsert*	Inserts one blank line above the current line.
	Arg Linsert	Inserts or deletes blanks at the beginning of a line to make the first nonblank character appear under the cursor.
	Arg boxarg Linsert *Arg linearg Linsert* *Arg streamarg Linsert*	Fill the specified area with blanks.
Mark (CTRL+M)	*Mark*	Moves the window to the beginning of the file.
	Arg Mark	Restores the window to its previous location. The editor remembers only the location prior to the last scrolling operation.
	Arg numarg Mark	Moves the cursor to the beginning of the line, where *numarg* specifies the position of the line in the file.
	Arg Arg textarg Mark	Defines a file marker at the initial cursor position. This does not record the file marker in the file specified by the **markfile** switch, but allows you to refer to this position as *textarg*.

Table A.3 (*continued*)

Function (and Default Keystrokes)	Syntax	Description
	Arg streamarg Mark *Arg textarg Mark*	Move the cursor to the specified file marker. If the file marker was not previously defined, the editor uses the **markfile** switch to find the file that contains file marker definitions. For more information, see Section 7.4.3, "Text Switches."
Meta (F9)	*Meta*	Modifies the action of the function it is used with. Refer to the individual functions for specific information.
Mlines (CTRL+W)	*Mlines*	Moves the window back by the number of lines specified by the **vscroll** switch or less if in a small window.
	Arg Mlines	Moves the window until the line that the cursor is on is at the bottom of the window.
	Arg numarg Mlines	Moves the window back by the specified number of lines.
Mpage (PGUP or CTRL+R)	*Mpage*	Moves the window backward in the file by one window's worth of lines.
	Arg Mpage	Moves the window to the beginning of the file.
	Arg numarg Mpage	Moves the window the specified number of windows backward in the file.
Mpara (CTRL+PGUP)	*Mpara*	Moves the cursor to the first blank line preceding the current paragraph, or if currently on a blank line the cursor is positioned before the previous paragraph.
	Meta Mpara	Moves cursor to the first previous line that has text.

Table A.3 (*continued*)

Function (and Default Keystrokes)	Syntax	Description
Msearch (F4)	*Msearch*	Searches backward for the previously defined string or pattern. If the string or pattern is found, the window is moved to display it and the matched string or pattern is highlighted. If no match is found, no cursor movement takes place and a message is displayed.
	Arg Msearch	Searches backward in the file for the string defined as the characters from the initial cursor position to the first blank character.
	Arg streamarg Msearch *Arg textarg Msearch*	Search backward for the specified text.
	Arg Arg Msearch	Searches backward in the file for the regular expression defined as the characters from the initial cursor position to the first blank character.
	Arg Arg streamarg Msearch *Arg Arg textarg Msearch*	Search backward for a regular expression as defined by *streamarg* or *textarg*.
Mword (CTRL+LEFT or CTRL+A)	*Mword*	Moves the cursor to the beginning of a word. If not in a word or at the first character, use the previous word, otherwise use the current word.
	Meta Mword	Moves the cursor to the immediate right of the previous word.

Table A.3 (*continued*)

Function (and Default Keystrokes)	Syntax	Description
Newline	*Newline*	Moves the cursor to a new line. If the **softcr** switch is set, the editor tries to place the cursor in an appropriate position based on the type of file. If the file is a C program, the editor tries to tab in based on continuation of lines and on open blocks. If the next line is blank, the editor places the cursor in the column corresponding to the first nonblank character of the previous line. If neither of the above is true, the editor places the cursor on the first nonblank character of the line.
	Meta Newline	Moves the cursor to column 1 of the next line.
Paste (SHIFT+INS)	*Paste*	Inserts the contents of the Clipboard prior to the current line if the contents were placed there in a line-oriented way, such as with *linearg* or *numarg*. Otherwise the contents of the Clipboard are inserted at the current cursor position.
	Arg Paste	Inserts the text from the initial cursor position to the end of the line at the initial cursor position.
	Arg streamarg Paste *Arg textarg Paste*	Place the specified text into the Clipboard and insert that text at the initial cursor position.
	Arg Arg streamarg Paste *Arg Arg textarg Paste*	Interpret *textarg* or *streamarg* as a file name and insert the contents of that file into the current file above the current line.
	Arg Arg !streamarg Paste *Arg Arg !textarg Paste*	Treat the text as a DOS command and insert its output to **stdout** into the current file at the initial cursor position. The exclamation mark must be entered as shown.

Table A.3 (*continued*)

Function (and Default Keystrokes)	Syntax	Description
Pbal (CTRL+[)	*Pbal*	Scans backward through the file, balancing parentheses and brackets. The first unbalanced one is highlighted when found. If it is found and is not visible, the editor displays the matching line on the dialog line, with the highlighted matching character. The corresponding character is placed into the file at the current cursor position. Note that the search does not include the current cursor position and that the scan only looks for more left brackets or parentheses than right, not just an unequal amount.
	Arg Pbal	Performs similarly to *Pbal*, except that it scans forward in the file and looks for more right brackets or parentheses than left.
	Meta Pbal	Performs similarly to *Pbal*, except that the file is not updated.
	Arg Meta Pbal	Performs similarly to *Arg Pbal*, except that the file is not updated.
Plines (CTRL+Z)	*Plines*	Adjusts the window forward by the number of lines specified by the **vscroll** switch or less if in a small window.
	Arg Plines	Moves the window downward so the line that the cursor is on is at the top of the window.
	Arg numarg Plines	Moves the window forward the specified number of lines.
Ppage (PGDN or CTRL+C)	*Ppage*	Moves the window forward in the file by one window's worth of lines.
	Arg Ppage	Moves the window to the end of the file.
	Arg numarg Ppage	Moves the window the specified number of windows forward in the file.

Table A.3 (*continued*)

Function (and Default Keystrokes)	Syntax	Description
Ppara (CTRL+PGDN)	*Ppara*	Moves the cursor forward one paragraph and places the cursor on the first line of the new paragraph.
	Meta Ppara	Moves the cursor to the first blank line following the current paragraph.
Psearch (F3)	*Psearch*	Searches forward for the previously defined string or pattern. If the string or pattern is found, the window is moved to display it and the matched string or pattern is highlighted. If it is not found, no cursor movement takes place and a message is displayed.
	Arg Psearch	Searches forward in the file for the string defined as the characters from the initial cursor position to the first blank character.
	Arg streamarg Psearch *Arg textarg Psearch*	Search forward for the specified text.
	Arg Arg Psearch	Searches forward in the file for the regular expression defined as the characters from the initial cursor position to the first blank character.
	Arg Arg streamarg Psearch *Arg Arg textarg Psearch*	Search forward for a regular expression as defined by *streamarg* or *textarg*.
Pword (CTRL+RIGHT or CTRL+F)	*Pword*	Moves the cursor forward one word and places the cursor on the beginning of the new word.
	Meta Pword	Moves cursor to immediate right of current word, or if not in a word to the right of the next word.

Table A.3 (*continued*)

Function (and Default Keystrokes)	Syntax	Description
Qreplace (CTRL+\)	*Qreplace*	Performs a simple search-and-replace operation, prompting you for the search and replacement strings, and prompting at each occurrence for confirmation. The search begins at the cursor position and continues through the end of the file.
	Arg boxarg Qreplace *Arg linearg Qreplace* *Arg streamarg Qreplace*	Perform the search-and-replace operation over the specified text, prompting at each occurrence for confirmation.
	Arg markarg Qreplace	Performs the search-and-replace operation between the initial cursor position and the specified file marker, prompting at each occurrence for confirmation.
	Arg numarg Qreplace	Performs the search-and-replace operation over the specified number of lines, starting with the current line, prompting at each occurrence for confirmation.
	Arg Arg Qreplace *Arg Arg boxarg Qreplace* *Arg Arg linearg Qreplace* *Arg Arg markarg Qreplace* *Arg Arg numarg Qreplace* *Arg Arg streamarg Qreplace*	Perform the same as their respective counterparts above, except that the search pattern is a regular expression and the replacement pattern can select special tagged sections of the search for selective replacement. See Chapter 5, "Regular Expressions," for more information.
Quote (CTRL+P)	*Quote*	Reads one keystroke from the keyboard and treats it literally. This is useful for inserting text into a file that happens to be assigned to an editor function.

Table A.3 (*continued*)

Function (and Default Keystrokes)	Syntax	Description
Refresh (SHIFT+F7)	*Refresh*	Asks for confirmation and then rereads the file from disk, discarding all edits since the file was last saved.
	Arg Refresh	Asks for confirmation and then discards the file from memory, loading the last file edited in its place.
Replace (CTRL+L)	*Replace*	Performs a simple search-and-replace operation without confirmation, prompting you for the search string and replacement string. The search begins at the cursor position and continues through the end of the file.
	Arg boxarg Replace *Arg linearg Replace* *Arg streamarg Replace*	Perform the search-and-replace operation over the specified text.
	Arg markarg Replace	Performs the search-and-replace operation between the cursor and the specified file marker.
	Arg numarg Replace	Performs the search-and-replace operation over the specified number of lines, starting with the current line.
	Arg Arg Replace *Arg Arg boxarg Replace* *Arg Arg linearg Replace* *Arg Arg markarg Replace* *Arg Arg numarg Replace* *Arg Arg streamarg Replace*	Perform the same as their respective counterparts above, except that the search pattern is a regular expression and the replacement pattern can select special tagged sections of the search for selective replacement. See Chapter 5, "Regular Expressions," for more information.
Restcur	*Restcur*	Restores the cursor position saved with *Savecur*.

Table A.3 (*continued*)

Function (and Default Keystrokes)	Syntax	Description
Right (RIGHT or CTRL+D)	*Right*	Moves the cursor one character to the right. If this would result in the cursor moving out of the window, then the window is adjusted to the right the number of columns specified by the **hscroll** switch or less if in a small window.
	Meta Right	Moves the cursor to the rightmost position in the window.
Savecur	*Savecur*	Saves the current cursor position to be restored with *Restcur*.
Sdelete (DEL)	*Sdelete*	Deletes the single character under the cursor, excluding line breaks. It does not place the deleted character into the Clipboard.
	Arg Sdelete	Deletes from the cursor through the end of line, joining the following line with the current line at the point of the cursor position. The text deleted (including the line break) is placed into the Clipboard.
	Arg boxarg Sdelete *Arg linearg Sdelete* *Arg streamarg Sdelete*	Delete the stream of text from the initial cursor position up to the current cursor position and place it into the Clipboard.
Setfile (F2)	*Setfile*	Switches to the most recently edited file, saving any changes made to the current file to disk.
	Arg Setfile	Switches to the file name that begins at the initial cursor position and ends with the first blank.
	Arg streamarg Setfile *Arg textarg Setfile*	Switch to the file specified by *streamarg* or *textarg*.

Table A.3 (*continued*)

Function (and Default Keystrokes)	Syntax	Description
	Meta Setfile *Arg Meta Setfile* *Arg streamarg Meta Setfile* *Arg textarg Meta Setfile*	Perform similarly to their counterparts above, but disable the saving of changes for the current file.
	Arg Arg streamarg Setfile *Arg Arg textarg Setfile*	Save the current file under the name specified by *streamarg* or *textarg*.
	Arg Arg Setfile	Saves the current file.
Setwindow (CTRL+])	*Setwindow*	Redisplays the entire screen.
	Meta Setwindow	Redisplays the current line.
	Arg Setwindow	Adjusts the window so that the initial cursor position becomes the home position (upper-left corner).
Shell (SHIFT+F9)	*Shell*	Saves the current file and runs the command shell.
	Meta Shell	Runs the command shell without saving the current file.
	Arg Shell	Uses the text on the screen from the cursor up to the end of line as a command to the shell.
	Arg boxarg Shell *Arg linearg Shell*	Treat each line of either argument as a separate command to the shell
	Arg streamarg Shell *Arg textarg Shell*	Use *streamarg* or *textarg* as a command to the shell.
Sinsert (CTRL+J)	*Sinsert*	Inserts a single blank space at the current cursor position.
	Arg Sinsert	Inserts a carriage return at the initial cursor position, splitting the line.
	Arg boxarg Sinsert *Arg linearg Sinsert* *Arg streamarg Sinsert*	Insert a stream of blanks between the initial cursor position and the current cursor position.

Table A.3 (*continued*)

Function (and Default Keystrokes)	Syntax	Description
Tab (TAB)	*Tab*	Moves the cursor to the next tab stop. Tab stops are defined to be every *n*th character, where *n* is defined by the **tabstops** switch.
Undo (ALT+BKSP)	*Undo*	Reverses the last editing change. The maximum number of times this can be performed is set by the **undocount** switch.
Up (UP or CTRL+E)	*Up*	Moves the cursor up one line. If this would result in the cursor moving out of the window, the window is adjusted upward by the number of lines specified by the **vscroll** switch or fewer if in a small window.
	Meta Up	Moves the cursor to the top of the window without changing the column position.
Window (F6)	*Window*	Moves the cursor to the next window to the right of or below the current window.
	Arg Window	Splits the current window horizontally at the initial cursor position. Note that all windows must be at least five lines high.
	Arg Arg Window	Splits the current window vertically at the initial cursor position. Note that all windows must be at least 10 columns wide.
	Meta Window	Closes the window.

Appendix B

Support Programs for the Microsoft Editor

This appendix discusses the following programs, which work in conjunction with the Microsoft Editor:

- **ECH.EXE**
- **MEGREP.EXE**
- **CALLTREE.EXE**
- **UNDEL.EXE, EXP.EXE,** and **RM.EXE**

The editor uses **ECH.EXE** in a way that is invisible to you; it is mentioned here only because it appears as a separate file on the disk. **MEGREP.EXE** searches through files for a string or regular expression. **CALLTREE.EXE** searches through program source files, locating function calls. The other three programs work with backup files. When a file is updated and the **backup** switch is set to **undel**, the old version of the file is copied to a hidden subdirectory called **deleted**. **UNDEL.EXE, EXP.EXE,** and **RM.EXE** manipulate the files in the **deleted** subdirectory.

B.1 MEGREP.EXE

Use this program to search through files for a simple string or regular expression. (See Chapter 5, "Regular Expressions," for more information on regular expressions.) The following is the command-line syntax for **MEGREP**:

megrep [[/C]] [[/c]] {/**f** *patternfile* | *pattern*} *files*

MEGREP.EXE searches through *files* for *pattern*, where *pattern* may be a string or regular expression. The **/C** option makes case insignificant in the search. The **/c** option lists the number of matches that are made. The **/f** option specifies that *pattern* to search for is located in *patternfile* rather than on the command line.

Note

MEGREP.EXE can be used separately or from within the Microsoft Editor using the *Arg Arg textarg Compile* command, where *textarg* uses the syntax described above.

B.2 CALLTREE.EXE

Use this program to create any of the following output files using C or assembly-language source files:

- Calltree listing file
- Called-by listing file
- Warning listing file
- Marker file for the Microsoft Editor

The following is the command-line syntax for **CALLTREE**:

calltree [[*options*]] *source-filename...*

Table B.1 gives the *options* you can use with **CALLTREE.EXE.**

Table B.1
CALLTREE.EXE Options

Option	Meaning
-a	Causes the argument lists to be shown with procedure definitions and references in the calltree listing file. It also causes an entry in the warning listing file if there is a discrepancy in an argument list.
-v	Causes a complete (verbose) listing in the calltree listing file. A previously viewed path is listed again, instead of being displayed as an ellipsis (...).
-i	Causes case insensitivity during name comparisons.
-q	Prevents output from going to the screen (quiet mode).
-s *symbol*	Specifies to search only for *symbol* in the source files.
-m *filename*	Uses the symbols listed in *filename* for calltree information.
-c *filename*	Specifies the name of the calltree listing file.
-b *filename*	Specifies the name of the called-by listing file.
-w *filename*	Specifies the name of the warning listing file.
-z *filename*	Specifies the name of the marker file that is created for use with the Microsoft Editor.
source-filename...	Specifies the names of the source files to use. The use of wildcards is permitted.

The calltree listing file produces an indented listing showing the procedure names at the left margin. Calls are shown indented four spaces per level. If a path has already been viewed, it is shown as an ellipsis (...). A recursive call is shown as an asterisk (*). If a call for an undefined procedure is made, a question mark (?) appears.

The called-by listing file produces a tabled listing of defined procedures and all references to them. The procedure names are sorted alphabetically.

The warning listing file lists duplicate procedure names and argument-list discrepancies if the **-a** and **-b** options are used.

The Microsoft Editor marker file lists the name, the file it was found in, and the line and column numbers for each function. This allows you to move quickly to any function, using the *Arg markarg Mark* command, by entering the function name as *markarg*. Use the **markfile** switch to provide the Microsoft Editor with the name of this file.

B.3 UNDEL.EXE

Use this program to move a file from the **DELETED** subdirectory to the parent directory. Its command-line syntax is as follows:

undel [[*filename*]]

If *filename* is not given, the contents of the **DELETED** subdirectory are listed. If there is more than one version of the file, you are given a list to choose from. If the file already exists in the parent directory, the two files are swapped.

B.4 EXP.EXE

Use this program to remove all of the files in the specified directory's hidden **DELETED** subdirectory. Use the following command-line syntax:

exp [[/r]] [[/qD]] [[*directory*]]

If no directory is specified, then the current directory's **DELETED** subdirectory is used. If the **/r** option is given, **EXP.EXE** recursively operates on all subdirectories. The **/q** option specifies quiet mode; the deleted file names are not displayed on the screen.

B.5 RM.EXE

Use this program to move one or more files from its current directory into the **DELETED** subdirectory. The following is the command-line syntax for **RM**:

rm [[/i]] [[/r]] [[/f]] *filename...*

The **/i** option prompts you for confirmation for each file it is about to delete. The **/r** option causes **RM.EXE** to recursively operate on all subdirectories. The **/f** option forces read-only files to be deleted without prompting.

Glossary

This glossary defines terms that this manual uses in a technical or unique way.

Arg

A function modifier that introduces an argument or an editing function. The argument may be of any type and is passed to the next function as input. For example, the command *Arg textarg Copy* passes the argument *textarg* to the function *Copy*.

argument

An input to a function. The Microsoft Editor uses two classes of arguments: cursor-movement arguments and text arguments. Cursor-movement arguments (*boxarg*, *linearg*, and *streamarg*) specify a range of characters on the screen. Text arguments (*markarg*, *numarg*, and *textarg*) allow you to enter information to be used by a function. Arguments are introduced by using the *Arg* function.

assignment

See "function assignment."

boxarg

A rectangular area on the screen, defined by the two opposite corners: the initial cursor position and the current cursor position. The two cursor positions must be on separate rows and separate columns. A *boxarg* is generated by invoking the *Arg* function and then moving the cursor to a new location.

buffer

An area in memory in which a copy of the file is kept and changed as you edit. This buffer is copied to disk when you do a save operation.

C extension

A C-language module that defines new editing functions and switches.

See Chapter 8, "Programming C Extensions."

Clipboard

A section of memory that holds text that has been deleted with the *Copy*, *Ldelete*, or *Sdelete* functions. You can use the *Paste* function to insert text from the Clipboard into a file.

configuration

A description of the specific assignments of functions to keys. For example, a BRIEF configuration implies that the Microsoft Editor uses keys similar to those that the BRIEF editor uses to invoke similar functions.

default

A setting that is assumed by the editor until you specify otherwise. The Microsoft Editor uses two categories of default settings: function assignments and switches.

emacs

A popular type of editor, from which the functions *Emacscdel* and *Emacsnewl* were taken.

function assignment

A method of assigning an editor function to a specific keystroke so that pressing the keystroke invokes the function. Use the *Arg textarg Assign* command to make an assignment for a single editing session, or you can enter the assignment in the **TOOLS.INI** file so that it may be used during any editing session.

See Chapter 6, "Function Assignments and Macros."

initial cursor position

The position the cursor is in when the *Arg* function is invoked.

insert mode

An input mode that inserts rather than replaces characters in the file as they are entered.

linearg

A range of complete lines, including all the lines from the initial cursor position to the current cursor position. You define a *linearg* by invoking the *Arg* function (pressing ALT+A), then moving the cursor to a different line but same column as the initial cursor position.

macro

A function that is made up of arguments and previously defined functions. For example, you can create a macro that contains a set of functions that you perform repeatedly and assign the macro to a keystroke. Those functions can now be carried out much more quickly and simply by invoking the macro.

See Chapter 6, "Function Assignments and Macros."

markarg

A special type of *textarg* that has been previously defined to be a marker, that is, it is associated with a particular position in the file.

marker

A name assigned to a cursor position in a file so that this position can be referred to within a command by using this name. For example, you could perform the command *Arg markarg Mark* to move to the marker specified by *markarg*. A marker is assigned using the *Arg Arg textarg Mark* command.

Meta

A function that modifies other functions so they perform differently, similar to the way CTRL or ALT modifies a key so that it performs differently.

numarg

A numerical value you enter on the dialog line, which is passed to a function. A *numarg* is introduced by the *Arg* function.

regular expression

A pattern for specifying a set of strings of characters to search for. It may be a simple string or a more complex arrangement of characters and special symbols that specify a variety of strings to be matched.

See Chapter 5, "Regular Expressions."

return value

A value returned by an editing function. The value may be true or false, depending on whether the function was successful. This value can be used to create complex macros that perform differently depending upon the results of individual functions within the macro.

See Chapter 6, "Function Assignments and Macros."

streamarg

A highlighted continuous string of characters on a single line. A *streamarg* is specified by invoking the *Arg* function and moving the cursor to any other position on the same line.

switch

A variable that modifies the way the editor performs. The Microsoft Editor uses three kinds of switches: Boolean switches, which turn a certain editor feature on or off; numeric switches, which specify numeric constants; and text switches, which specify a string of characters.

See Chapter 7, "Using the TOOLS.INI File."

textarg

A string of text that you enter on the dialog line, after invoking the *Arg* function (by pressing ALT+A). The text that you enter is passed as input to the next function.

TOOLS.INI

A file that contains initialization information for the Microsoft Editor and other programs. The file is divided into sections with the use of tags, and these sections can be loaded automatically when the editor is started or by command from within the editor.

See Chapter 7, "Using the TOOLS.INI File."

window

An area on the screen used to display part of a file. Unless a file is extremely small, it is impossible to see all of it on the screen at once. Therefore you see a portion of the file through the main editing window at any one time, and it is possible to see any part of the file by moving or scrolling this window. The Microsoft Editor allows you to open multiple windows on the screen, using the *Window* function, for viewing different parts of the same file or different files.

Index

Microsoft Corporation
16011 NE 36th Way
Box 97017
Redmond, WA 98073-9717

Microsoft®
Making it all make sense™

0388 Part No. 02949