Microsoft® Editor

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

User's Guide

Microsoft Corporation

Document No. 614510001-100-R00-0288
Part No. 01694
9 8 7 6 5 4 3 2 1

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):

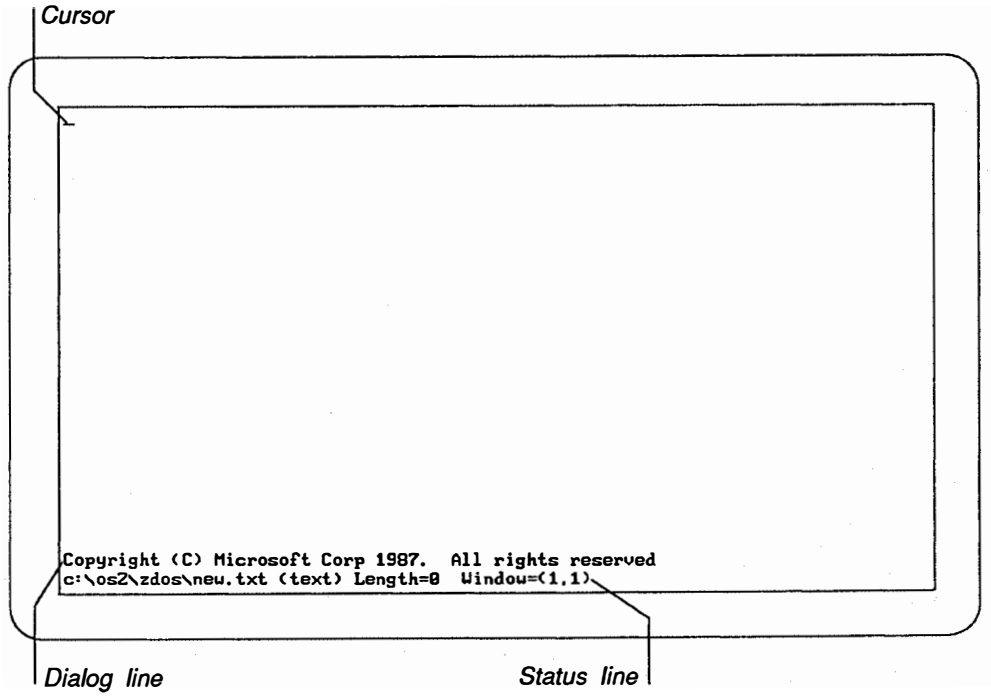


Figure 2.1 Microsoft Editor's Screen

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

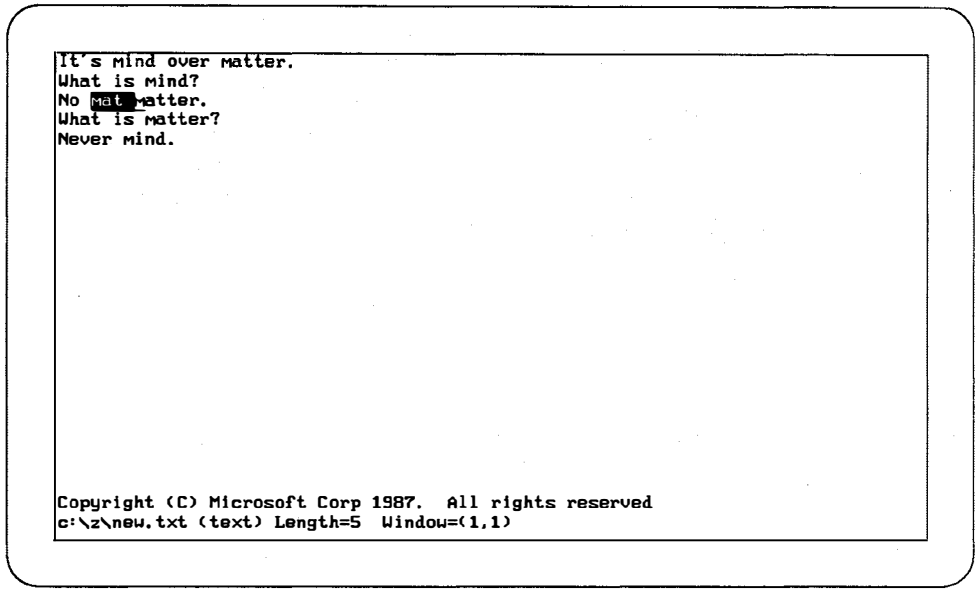
Section 2.3.3, "Introducing the Arg Function," examines this command sequence in detail.

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.

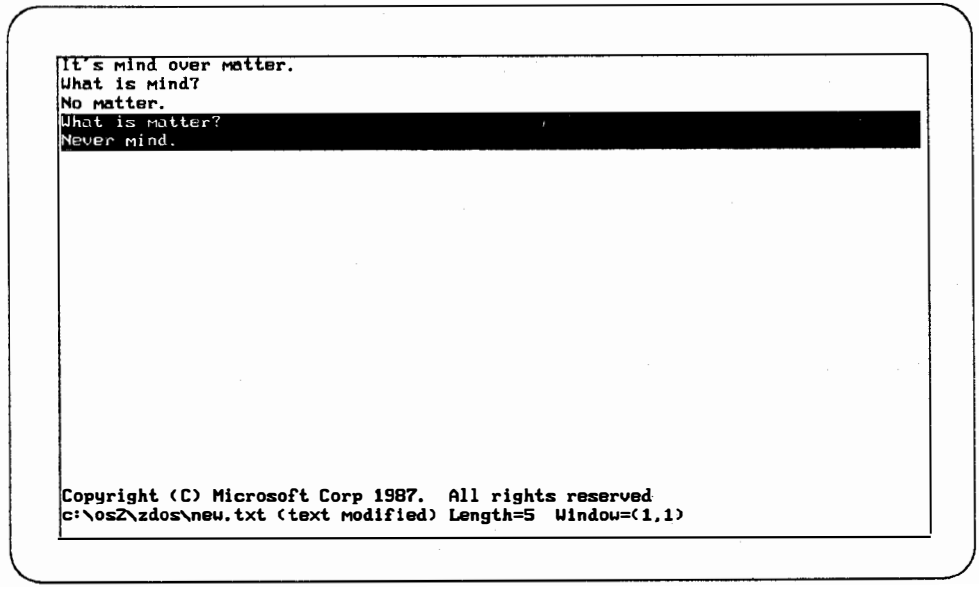
2.3.5 Using Ldelete to Move Text

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:

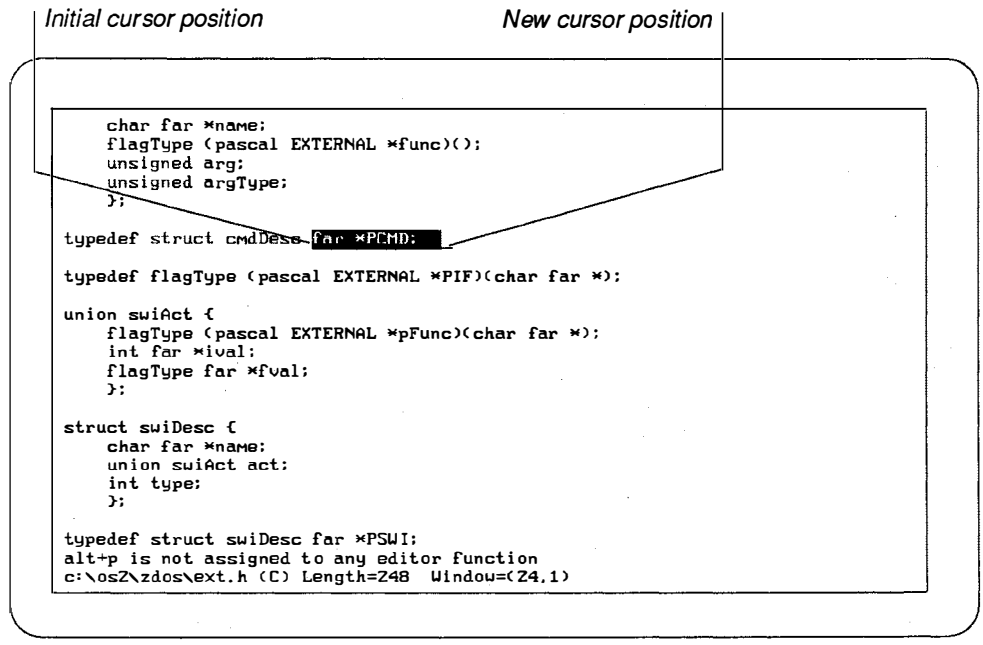


Figure 3.1 Sample streamarg

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:

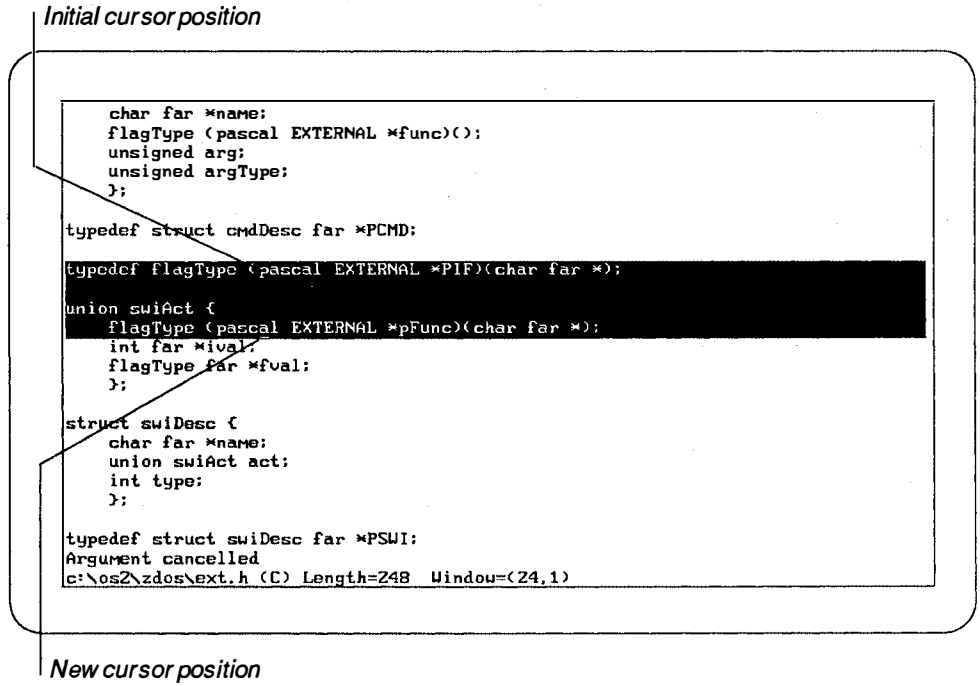


Figure 3.2 Sample linearg

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:

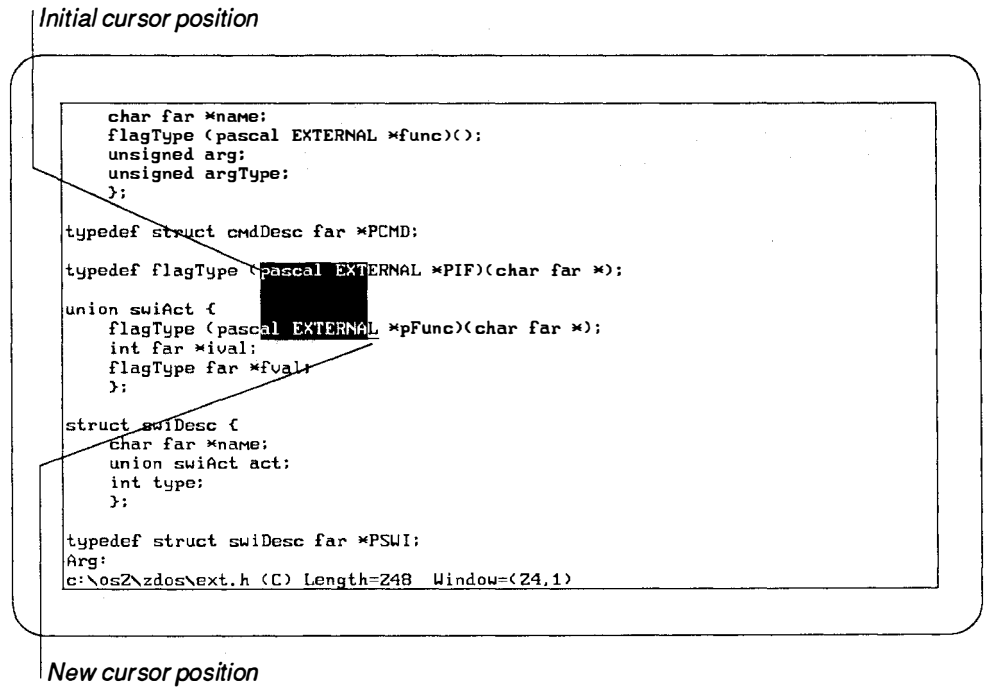


Figure 3.3 Sample boxarg

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`.
	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, **Mxxx.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	ALT+D	CTRL+Z
Ppage	PGDN or CTRL+C	PGDN or CTRL+C	PDGN	PDGN or CTRL+V
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	F9	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 or CTRL+W
Setfile	F2	F2	ALT+N	F2
Setwindow	CTRL+]	CTRL+]	F2	CTRL+]
Shell	SHIFT+F9	SHIFT+F9	ALT+Z	ALT+Z
Sinsert	CTRL+J	CTRL+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