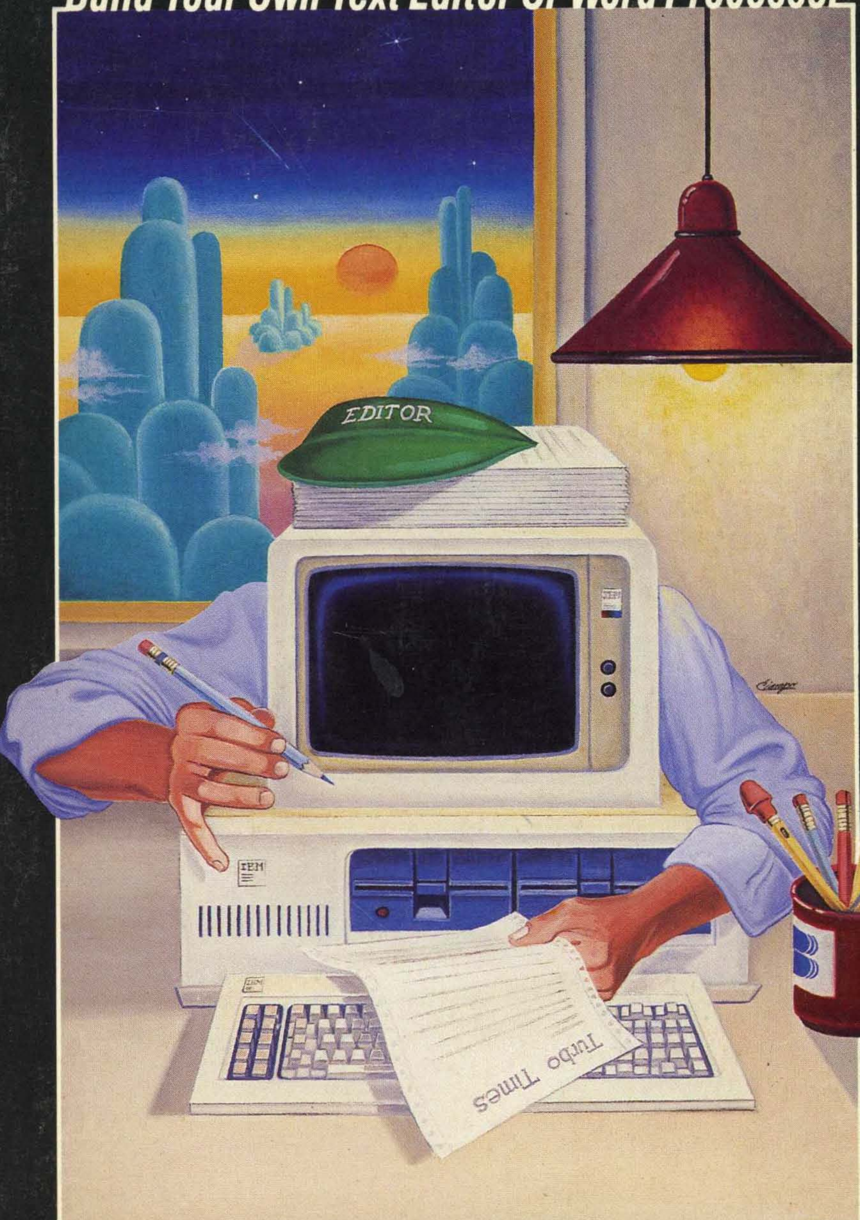
TURBO EDITOR TOOLBOX

Build Your Own Text Editor Or Word Processor



Turbo Editor Toolbox

version 1.0

Owner's Handbook

BORLAND INTERNATIONAL Inc.
4585 Scotts Valley Drive
Scotts Valley, CA 95066

TABLE OF CONTENTS

LIST OF FIGURES

INTRODUCTION

Welcome to Turbo Editor Toolbox. This software package will enable you to explore the interesting world of editing systems using Turbo Pascal. With the aid of the Editor Toolbox, you can develop text editors and applications for the IBM PC family (and true compatibles) that use text-editing functions.

This manual makes extensive use of Turbo Pascal programming examples, and a good working knowledge of Turbo Pascal is assumed. If you need to brush up on the Pascal language, refer to the *Turbo Pascal Reference Manual* and/or the *Turbo Tutor*.

What Can You Do With the Editor Toolbox?

The Turbo Editor Toolbox is designed to help you build both simple and complex text editing applications. With the Toolbox, you can write a complete, full-featured text editor, or add specific text editing functions to your Turbo applications. We provide high- and low-level procedures which allow you to manipulate textual information as:

- Single characters
- Groups of characters: words and lines
- Groups of lines: text streams
- Windows of text streams
- Blocks of lines
- Files of lines
- Screen displays

Required Hardware

As shipped, the Editor Toolbox will run with no alterations on an IBM PC, PCjr, XT, or AT (or a 100% compatible computer). Any of the standard display adapters may be used, including the IBM Monochrome display adapter, the IBM Color/Graphics adapter, and the Enhanced Color/Graphics adapter.

If your computer is *not* 100% IBM PC-compatible, but uses a "memory-mapped" video display, it may be possible to adapt the Editor Toolbox display routines to your machine. This conversion requires an intimate knowledge of the display hardware used in your computer, and possibly some assembly language skills.

Required Software

To compile the Editor Toolbox routines, you will need Turbo Pascal (preferably Version 3.0 or greater), as well as MS-DOS or PC-DOS (Version 2.0 or greater).

Structure of this Manual

This manual is divided into four parts:

- Part I, *A Text Editing Primer*, is an overview of the terminology, philosophy, user interfaces, and data structures of modern-day text editing programs. This section provides the basic information you need to know to make good use of the Turbo Editor Toolbox.
- Section II, *Building an Editor*, guides you through the construction of your first editor. The code for a very simple editor (which does not use the Toolbox) is presented and explained. Then, the two sample Toolbox editors, FIRST-ED and MicroStar, are described—along with the details of their internal structure.
- Section III, *Harnessing the Full Power of the Turbo Editor Toolbox*, takes you behind the scenes and explains the low-level structure of the Toolbox itself. This is the information that you need to modify the Turbo Editor Toolbox routines to suit your specific needs.
- Section IV, the *Turbo Editor Toolbox Technical Reference*, lists all of the constants, types, variables, procedures and functions of the Turbo Editor Toolbox.

Typography

The body of this manual is printed in normal typeface. Special characters are used for the following special purposes:

`Alternate`	`Alternate characters are used in program examples and procedures and function declarations.`
Italics	*Italics are used to emphasize certain concepts and terminology, such as predefined standard identifiers, parameters, and other syntax elements.*
Boldface	**Boldface type is used to mark reserved words, in the text as well as in program examples.**

Refer to the *Turbo Pascal Reference Manual* for a complete description of syntax, special characters, and overall appearance of the Turbo Pascal language.

The Distribution Diskettes

The Turbo Editor Toolbox is distributed on two diskettes. Disk #1 contains:

- The files README.COM and READ.ME, which describe in detail the most current version of the Toolbox.
- The source code for the Toolbox routines (.ED files).
- The file EDITERR.MSG, which contains the error messages displayed by the Toolbox routines.
- A simple editor, FIRST-ED.PAS, which uses the Turbo Editor Toolbox. This file should be compiled to a .COM file before being run.

Disk #2 contains the following files:

- MicroStar (MS.PAS), a sophisticated editor that uses the Turbo Editor Toolbox. This file should be compiled to a .COM file before being run.
- Include files (.ED, .OVL, .INC) necessary to compile MicroStar (some of these are duplicates of files found on Disk #1).
- Another copy of the error message file, EDITERR.MSG.

For a complete listing of the files included in the Turbo Editor Toolbox package, insert Disk #1 into your computer and type README. Last-minute modifications will also be described here.

Your distribution diskettes are your master copy of the Turbo Editor Toolbox files. Immediately after receiving the Toolbox, you should complete and mail the License Agreement at the front of this manual. You should then copy your distribution diskettes and put them away in a safe place. *Never use your distribution diskettes as working diskettes. There is a charge for replacement copies.*

Acknowledgments

In this manual, references are made to several products:

Turbo Editor Toolbox, MicroStar, Turbo Pascal, Turbo Tutor, SideKick, and SuperKey are trademarks of Borland International.

IBM is a registered trademark and PC, PCjr, XT, AT, and PC-DOS are trademarks of International Business Machines Corporation.

MacWrite is a trademark of Apple Computer Corp.

MS-DOS and Microsoft Word are trademarks of Microsoft Corporation.

MultiMate is a trademark of MultiMate International Corporation.

PFS:Write is a trademark of Software Publishing.

Volkswriter is a trademark of Lifetree Software Inc.

WordPerfect is a trademark of Satellite Software International.

WordStar is a trademark of MicroPro International Corporation.

Section I

A TEXT EDITING PRIMER

Chapter 1
SOME TEXT-EDITING TERMS

Before working with the Editor Toolbox, you will need to understand the text-editing terminology used throughout this manual. Some of the most important terms and concepts are defined below.

Editor

An *editor* is a program that allows its user to create, update, or modify information, usually stored in logical chunks called files. Some editors are used to edit graphic information, such as topographic maps, or schematic diagrams. Others edit textual information, encoded in a language that is intended for a computer or a human to read. Editors come in many flavors and styles, to suit different needs and purposes. Here we will discuss several types of editors.

Our focus is on *text editors*, which edit files containing textual encoded information. Text editors allow the user to create, update, or modify text at many levels of organization—from single characters or groups of them, to entire blocks or files of text.

A text editor may be very simple, such as the line editor supplied with the operating system, or as sophisticated as MicroStar, the demonstration editor supplied with the Turbo Editor Toolbox. You are probably already familiar with the text editor that is integrated into Turbo Pascal. With the Editor Toolbox, you can write your own applications with built-in editors.

Text

The term *text* refers to a sequence of characters and/or lines that are being edited. Text may be a program, a document, or any sequence of one or more characters.

The characters of a piece of text are usually represented using the widely-accepted ASCII (American Standard Code for Information Interchange) code. This standard assigns a numeric value to every character in the English alphabet, distinguishing upper from lower case. It also includes the numeric characters '0' through '9', some special symbols (e.g., !@#$%^&*()—+={ } [ ] :;"'< >?,./), and finally, a set of control characters.

Besides assigning a unique numeric value to each character, the ASCII character code also establishes an *order* for the characters in the character set. Programs can depend on, and take advantage of, the fact that no matter what numeric value the letter "B" is assigned to, that value is always one more than the numeric value for "A".

In most natural languages like English, and in nearly every computer programming language like Pascal, characters are combined in groups, separated by "white space" such as blanks, tabs, line feeds, or carriage returns. Most editors allow the user to manipulate text not only on a character basis, but also on several other organizational levels, including words, lines, and contiguous sequences of lines.

An editor can distinguish these units of information from one another by looking for special unseen characters in electronic text, called control characters. Although not usually considered to be a part of the message of a piece of text, they are always embedded in the text, and take up the same room as other characters do. The following control characters are used by nearly every text editor on most computer systems, whether they're microcomputers or supercomputers:

Space, or <SP> (ASCII code 32)	- Separates words
Tab, or <HT> (ASCII code 9)	- Separates words, indicates spacing
Return, or <CR> (ASCII code 13) and Line feed, or <LF> (ASCII code 10)	Together, these characters - mark the end of a line of text
Form feed, or <FF> (ASCII code 12)	- Marks the end of a page

Sometimes, more than one kind of textual *delimiter*, or separating character, may be acceptable to mark a particular division in a piece of text. For example, programs like compilers often allow words to be separated by either spaces or tabs, and lines by either line feeds carriage returns, or both. All of the characters listed above are traditionally referred to as "white space", since they don't print anything on paper when output to a printer. However, they do affect the location where neighboring visible characters will be printed.

Some editors have the capability to show these characters graphically on the display device. Turbo Editor Toolbox allows you to make editors which graphically display most control characters, but space, tab, line feed, and return are always recognized as text delimiters.

Line

A *line* is a sequence of characters displayed on a single row of the screen. In many text editors, it is possible for there to be more characters in a line than there are columns on the screen; in this case, a means is usually provided to shift the displayed characters left and right so that the entire line may be seen.

Text Stream

A *text stream* is a group of one or more lines of text. A text editor manipulates one or more text streams.

Window

A *window* is a rectangular region of the computer's screen used to display information and keep it separate from other information that may be present on the screen at the same time. In a text editor, a window usually "looks" into, and displays a portion of, the text stream being edited. If the editor is capable of displaying more than one window, different parts of the same text stream (or parts of two different text streams) may be displayed simultaneously. In the Editor Toolbox, windows always span the full width of the screen.

Cursor

The *cursor* is a small block or line on the screen (sometimes blinking) that marks the place where changes are being made to the text. In a screen editor, the cursor is usually within a text window. In editors built with the Toolbox, each window can have a different cursor position, although the cursor itself will only be displayed in one window at a time.

Block

A *block* is a contiguous sequence of characters within a larger body of text. By "marking" a block (that is, indicating its beginning and end), the user of a text editor enables operations to be performed on that entire piece of text as a unit.

In editors built with the Turbo Editor Toolbox, as in most screen-oriented editors, the currently marked block of text is displayed in a different color from the rest of the text (or as highlighted text on a monochrome display). To improve the speed and efficiency of the editor, the Toolbox routines require a block to consist of whole lines of text, rather than parts of lines.

File

Depending on the context, the term *file* may refer to a disk file or to the text of a disk file which is being edited. In a window-oriented editor, such as can be built with the Toolbox, files may be read and written to and from windows.

Command

A *command* is a keystroke or sequence of keystrokes given to an editor that invokes an editor operation. A command may modify text in some way, or it may manipulate a block, file, or window. The Toolbox contains many predefined routines which process commands. In addition, you may write your own commands to supplement the Toolbox, or to replace existing commands.

Command Dispatcher

A *command dispatcher* is a procedure in an editor (or other interactive program) that interprets characters typed as commands, and calls one or more *command processors* to execute those commands.

Command Processor

A *command processor* is a procedure that does the actual work associated with the performance of a command. The Turbo Editor Toolbox contains a complete set of command processors for the manipulation of text.

Document Mode, Nondocument Mode

The desired behavior of a text editor often depends on the type of text being edited. When editing a letter, for instance, it is often convenient to have the editor adjust the format of the text so that the lines are full and even. This feature (called *word wrapping*) is very useful for correspondence, but is *not* helpful to a user who is composing computer programs (where the vertical alignment of the text must not be disturbed). For this reason, text editors usually edit text files in one of two modes: either "document" or "nondocument" mode.

In *document* mode, the editor reformats text as it is entered to make paragraphs look pleasing. To speed this reformatting process (and also to distinguish characters which were added for aesthetic purposes from those which the user specifically typed), the editor may place a "1" in the high bits of certain character codes in the text. Such a file will appear garbled when typed with the DOS "type" command, but no information has been lost; in fact, additional information about how the text has been formatted is present in the file.

In *nondocument* mode, an editor performs no automatic reformatting; the user must position every character manually. This is useful when writing computer programs, in which the exact position of every word of text should be controlled to provide high legibility. Because this mode does not use the high bits of character codes to contain editor information, these bits can be used to perform other functions. For instance, on the IBM PC, the character codes with the high bit set cause special graphic and foreign-language characters to be displayed. Non-document mode gives the user the ability to intersperse these characters with text. (Such characters might confuse, or be misinterpreted by, an editor operating in document mode.)

Chapter 2
TYPES OF EDITORS

Line Editors

The earliest text editors for microcomputers were *line editors*—editors that allow the user to display and edit text only one line at a time. Every computer with MS-DOS or IBM PC-DOS comes with a line editor on the operating system disk; it is called EDLIN, and is designed to work with any MS-DOS machine.

EDLIN does not know what kind of terminal the computer uses, and thus can make very few assumptions about how it may move the cursor or manipulate the image you see on the screen. In fact, it cannot assume that there is a video display at all—and so restricts itself to actions which can be done on a printing terminal, such as a teletype.

In EDLIN, the user enters single-letter commands to manipulate lines of the file. For instance, the EDLIN command to enter new text into a file is "I", for "insert," followed by a carriage return:

```
I <RETURN>
```

EDLIN responds by asking for a line of text. If the file is new, EDLIN will ask for line 1 by responding:

```
1:_
```

and waiting for the user to input a line. When the user types a carriage return at the end of the line, EDLIN responds by prompting for another, and another, until <Ctrl-Z> is pressed:

```
1: This is some text
2: to test the line editor.
3: <Ctrl-Z>
```

The user can use other single-letter commands to perform basic editing operations, such as "L" to list lines, "D" to delete lines, "R" to replace one string with another, and "W" to write the edited file to the disk.

While EDLIN is a powerful editor, and can be used in some situations where an editor like WordStar cannot, it is certainly not convenient for everyday use. The requirement to type "L" just to see what the text looks like is annoying, and since every line must be referred to by number, the user expends much unnecessary effort just trying to find the line number associated with the text to be worked on.

If you are fortunate enough never to have had to use a line editor, you may want to find EDLIN on your DOS disk and experiment with it. It is by far the best way to appreciate how far text editors have come since the early days of microcomputers!

WYSIWYG Editors

The acronym WYSIWYG (pronounced "whizzy-wig") stands for the phrase "What You See Is What You Get"—a characteristic of most modern text editors. When using a WYSIWYG editor, the user sees what the finished text will look like (or something close to it) as it is manipulated. Text is entered by simply typing it in; a cursor indicates where the new text will go. At all times, the context surrounding the place where the editing is being performed is visible.

Virtually all WYSIWYG editors also have commands to allow the user to mark blocks of text "by eye" (rather than specifying them by line number). These blocks can then be copied, moved, or deleted.

Some of the more advanced WYSIWYG editors (such as the editors that can be built with this Toolbox) have the ability to display more than one window at a time. This feature allows two texts to be compared or combined with great ease and efficiency.

Most modern text editors are WYSIWYG editors. One of the oldest, and still the most popular, is called WordStar, which can be had for almost any computer, regardless of size or type. The command structure of WordStar is so well-known to so many computer users that we at Borland elected to use it in the editors for Turbo Pascal and SideKick.

Other popular WYSIWYG editors include Microsoft Word, Volkswriter, PFS:Write, WordPerfect, and MacWrite. Some of these editors will even allow pictures to be included in the text, and will show different fonts by manipulating individual pixels of the display screen.

RAM-Based Editors

A RAM-based editor is an editor that loads an entire disk file into memory at once, and keeps it there while it is being edited. Because all of the text in the file is available to the program directly through data structures in RAM, searching through the file is simply a matter of examining these data structures, and new text is added by creating new ones. Most RAM-based editors are extremely small, fast, and simple; the Turbo Pascal, SideKick, and SuperKey editors are all RAM-based for this reason (as are the editors you can build with this Toolbox).

"Swapping" or "Virtual" Editors

Of course, using system memory to store the text limits the amount of text that can be edited at any one time to the amount of memory in the system. This is acceptable for many applications, such as programming, where it is good practice to break programs up into small, easily understandable modules. However, for some applications (or for computer systems with limited memory) it may be desirable to be able to edit a file of any size. A "swapping" or "virtual" editor allows the user to do this.

In a "virtual" editor, parts of the text that are not currently being edited or displayed are written out to a disk file (called a "swap" file), and replaced with the portion of the text that is being worked on at the time. The text being edited can thus be as large as the storage device that holds the swap file. However, because disk accesses are much slower than memory accesses, operations involving data in the swap file are slowed down dramatically.

This is an important trade-off in editor design: fast with limited capacity, or slow with unlimited capacity. The Toolbox is primarily designed to operate as a memory-resident editor, but may be altered to allow a "swapping" mode as well.

Chapter 3
REPRESENTATION OF TEXT

Character Data

Most computer systems represent character data as 8-bit bytes. The ASCII standard defines the numbers 0 through 127 as valid numeric codes that stand for commonly used characters. Because it takes seven bits to represent 128 different values, an extra bit in each byte usually remains unused. By convention, this is the "high bit" of the byte, or bit number seven. The following is a graphical representation of a byte, and an ASCII character in that byte.

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
High bit	←		ASCII Character				→

Figure 3-1. Representing an ASCII Character in an 8-bit Byte

Because a byte is just a unit of storage in a digital computer, it can be treated as a number, and compared with other bytes, just like integers can be compared. Compilers and other computer programs use this notion to decode the user's program text when compiling a program.

Some editors, such as the SideKick editor and the Turbo Pascal editor, treat characters strictly according to their ASCII codes. Other editors, such as MultiMate, use their own (nonstandard) modification of ASCII. WordStar has a "nondocument mode" that selectively sets the high bit of certain bytes. This high bit is very useful for storing information about how the text is formatted (for example, right justification, proportional spacing, fonts, and so on). However, there are no standards for the meaning of the high bit, so setting the high bit usually makes the file unintelligible to other programs and computers.

The Structure of the Editor's Text Buffer

One of the most important characteristics of the design of a text editor is the data structure it uses to store textual data in memory while it is being edited. There are a number of possible techniques, and each affects the performance of specific editing tasks. Here, we will consider three ways of managing the text buffer, in order of increasing complexity and elegance.

The "Array-of-Lines" Approach

The simplest and most obvious way to arrange the text is simply as an array of strings, each with a given maximum length. For instance, if you were writing an editor to edit a piece of text that was never more than 400 lines long, and each line was to be no more than 80 characters in length, you might declare your buffer as shown in Figure 3-2.

```
type
  Buffer = array [1..400] of string[80];

var
  Note : Buffer;
```

and perform your editing within that array.

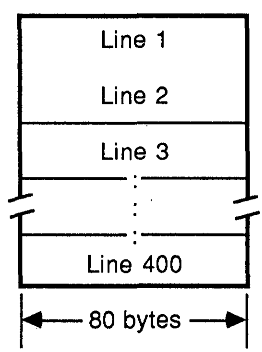


Figure 3-2. Array-of-Lines Buffer Structure

This "brute force" method, while extremely fast and simple, makes very poor use of space. Each line takes up the full 80 characters (plus an additional byte for the length) whether it is empty or full. Since a Turbo Pascal program can have, at most, 64K of global variables or stack space—of which this size array will take up nearly half—it must be allocated on the heap if there is to be room available for many additional variables. If there are to be several active buffers at one time, the space on the heap may be quickly consumed as well.

When this approach is used, adding characters to a line is very fast; the characters are merely appended to the appropriate string. Scanning for a pattern is reasonably fast when using the Turbo **Pos** function. However, adding a new line requires a block move, which is quite time-consuming.

The "Fixed Buffer" Approach

The next technique also allocates a fixed chunk of memory to store the text in, but does not waste as much space. The text is simply read into a large, contiguous block of memory, control characters and all. Viewed as a very big array of bytes, this organization has the advantage of being quick to scan for strings. It is also fairly quick to scan for end of lines when moving up or down in the file.

Another advantage to this approach is that the transfer of text to and from the computer's file system (usually disk files) is most efficient. Text can be read from disk directly into the text buffer with no need to scan it as it is read; similarly, it can be quickly output to files using Turbo Pascal's *BlockWrite* procedure.

This structure displays its greatest weakness when text is inserted into the buffer. Inserting characters into the buffer in the middle (or, worse, at the beginning) can be very slow if the most obvious method is used—that is, performing a block move to make space for each new character. Insertions can be speeded up somewhat by creating a large "hole" in the middle of the buffer in which to insert text, so that only one block move is done as the insertion begins. However, as the body of text gets large, opening the hole still causes an inordinately long delay.

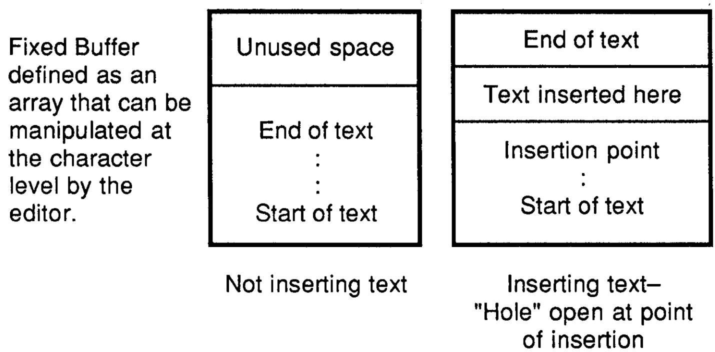


Figure 3-3. Layout of Memory in Fixed-Buffer Model

The "Linked-List" Approach

In this representation, the text consists of a linked list of Pascal records, each of which contains pointers to: 1) the previous line, 2) the following line, and 3) a string containing the line of text. This is the representation used in the Turbo Editor Toobox.

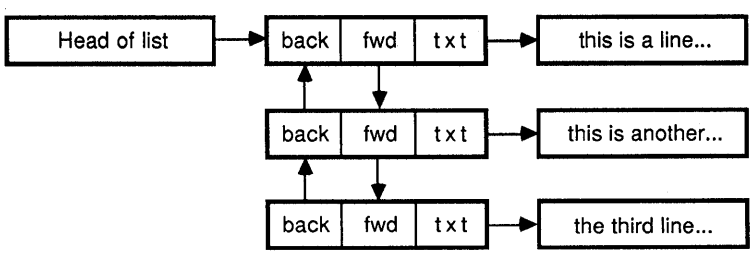


Figure 3-4. Layout of Memory in Linked-List Model

In this approach, insertion of new lines is relatively simple and fast. A new list record, called a *line descriptor*, is allocated on the heap and linked into the existing list of lines. Space for the text of the line is also allocated dynamically, and is pointed to by the *text* pointer of the new line descriptor.

This approach is more complicated to program than the previous two, but provides more even performance all around. Some of its advantages follow.

First, operations such as "cut" and "paste" are implemented as simple splicing of linked data structures. Implementing this kind of operation is very simple and quick.

Second, an editor designed around this technique can manage an unlimited number of files of arbitrary size (subject only to memory constraints), since memory is allocated dynamically.

This method is not without disadvantages. Its primary drawback is that files read into the structure from disk must be scanned while they are read to break them up into lines, and memory must be allocated for each line when a new line is read from the file. Writing to files from this structure involves a write to the disk for each line, using the line's text buffer as the write buffer as well.

If the impact of these problems on performance is minimized, then the linked-list model is superior to the fixed buffer model. In the case of the Turbo Editor Toolbox, a linked-list model is combined with high-speed disk I/O routines to provide the best overall performance.

Section II
BUILDING AN EDITOR

Chapter 4
AN (ALMOST) TRIVIAL EDITOR

In this chapter, we will present an extremely simple editor based on the array-of-lines data structure. This editor is so simple that it does not require the services of the toolbox at all; however, its limitations help to highlight the power and ease of use that an editor built with the toolbox can provide.

We start by defining an array of strings to serve as the buffer for our text. Since this is only a small sample program, we will settle for 50 lines of up to 80 characters each; thus, the buffer can be defined as:

```
const
  MaxLines = 50;    { We will allocate space for 50 lines      }


var
  Lines : array [1..MaxLines] of string[80]; { The text buffer }
```

We also need some housekeeping variables—one to keep track of the line we are currently working on, another to serve as a loop index, and another to hold the command input by the user, etc. The remaining declarations provide these variables.

We begin execution by clearing all of the strings in the array to the null string, and setting the current-line pointer equal to 1. We then enter an endless loop, in which we get a command and process it. The large case statement processes all of the commands; each is only a few lines of code long. The user can change any line by entering a new line to take its place; he or she can also show the current line, list all the lines, and write the text to a file.

Try all of the commands in this simple program (it is on your program disk) and use them to create some text. Despite its simplicity, this is a real text editor, similar to the editors that the pioneers of microcomputing used on their early machines!

```
program Line;

{$R+}

const
  MaxlLne = 50;    {We will allocate space for 50 lines        }

var
  Lines   : array [1..MaxLine] of string[80]; {The text buffer}
  CurLine : 1..MaxLine;                       {The current line}
  i       : 1..MaxLine; {Loop index, used in various commands }
  cmd     : char;                           {Command character}
  fn      : string[80];           {Name of file to read or write}
  f       : text;                   {File variable for work file}

begin
  for i := 1 to MaxLine do                     {Clear the buffer}
   Lines[i] := '';

CurLine := 1;     {Start with the current line at the beginning}

repeat
    Write('Line ', curline:2, '>');    {Prompt with line number}
    Readln(cmd);                                  {Get a command}

    case UpCase (cmd) of
      'U' : if curline > 1 then                         {Up line}
              curline := Pred (curline);

      'D' : if curline < maxline then                 {Down line}
              curline := Succ (curline);

      'C' : begin                                   {Change line}
              Writeln('New line:');
              Readln(lines[curline]);
            end;
```

```
      'S' : Writeln(lines[curline]);           {Show lines}

      'L' : for i := 1 to maxline do           {List buffer}
              Writeln(i:1, '> ', lines [i]);

      'W' : begin                              {Write file}
              Write ('Filename:');
              Readln (fn);
              Assign (f, fn);
              Rewrite (f);
              for i := 1 to maxline do
                writeln(f, lines [i]);
              Close (f);
            end;

      'Q' : Halt;                                     {Exit}

      else
        writeln ('Illegal command');

      end;                                            {Case}
  until false; {Loop until user quits}
end.
```

Because of its simplicity, this editor is far from bulletproof; for example, any file I/O error will crash the program immediately. While not a commercial-quality product, this program does illustrate the ease with which a very simple line editor can be created. Of course, most of us are spoiled nowadays—we want (and expect) an editor with the power of the Turbo Pascal editor or the Turbo Editor Toolbox.

Chapter 5
CREATING FIRST-ED—A FIRST EDITOR USING THE TOOLBOX

The task of designing a good text editor is a difficult one. How should the text appear on the screen? What should the commands be like? How should files be formatted? Should the editor support multiple text windows? Ultimately, the answers to all of these questions are a matter of personal preference. The Turbo Editor Toolbox is flexible enough to allow you to create editors with a wide variety of features.

With the Turbo Editor Toolbox, you do not need to start from scratch when designing your customized text editor. The Toolbox comes out of the package pre-configured as a simple, but powerful, editor with WordStar-like commands.

In this chapter, we will show how the Toolbox can be used unaltered to create and use FIRST-ED, the Toolbox *default* editor. In subsequent chapters, we will demonstrate how to customize FIRST-ED, and rearrange the parts of the Toolbox to create an editor of your own design.

Building FIRST-ED

FIRST-ED, the default Toolbox editor, demonstrates how to use the entire Toolbox as a "black box." To build FIRST-ED, it is only necessary to include all of the Toolbox files (those files with the extension .ED) in a program in the proper order, and define some empty procedures (these are "hooks" that we will use to customize the editor later). The Toolbox variables are initialized by invoking the procedure *EditInitialize,* and a built-in command loop is invoked with a call to the procedure *EditSystem.* Because the entire Toolbox is used here with no changes, the main program for FIRST-ED is only three lines long, including one line to simply clear the screen when the program is finished!

```
{$C-,I-}
program FirstEditor;
{
        Copyright (c) 1985 by Borland International, Inc.

1.      Program name.
        FIRST-ED - Simplest editor possible for Turbo Editor Toolbox.

2.      Functional description.
        This program demonstrates how easy it is to generate an editor with
        Turbo Editor Toolbox.  The following source includes all of the
        necessary modules and then calls EditSystem to execute the default
        editor loop.

        Last modified:  10/25/85

        System requirements:  IBM PC and true compatibles
                              TURBO PASCAL 3.0
                              DOS 2.0 or later
                              128 K-bytes system memory minimum

        List of data files:
          EDITERR.MSG   - Text file containing error messages.
}

{$I VARS.ED }    { Toolbox global variables and data structure definitions }

{
  The following procedures must always be defined, even if they are just
  null procedures (as they are here). Why? They're "hooks" into the Toolbox
  command processors, error processors, display routines, and scheduler.
  These routines will allow you to (1) redefine and add commands, (2) control
  the presentation of error messages to the user, (3) control the status line
  display, (4) Control the prompting mechanism for find & replace operations,
  and (5) provide tasks for the editor's multitasking scheduler to execute in
  the background.
}

procedure UserCommand(var ch : byte);
{ user command processor hook }
begin
end;

procedure UserError(var Msgno : byte);
{ user error handler hook }
begin
end;
```

```
procedure UserStatusline(var Wn : byte; Column, Line : integer);
{ user status line handler }
begin
end;

procedure UserReplace(var ch : byte);
{ user replace handler hook }
begin
end;

procedure UserTask;
{ user multi-tasking hook }
begin
end;

{$I USER.ED     } { Editor kernel and primitive level helper routines }
{$I SCREEN.ED   } { Screen updating routines            }

{$I INIT.ED     } { initialization code      }
{$I KCMD.ED     } { Ctrl-K routines          }
{$I OCMD.ED     } { Ctrl-O routines          }
{$I QCMD.ED     } { Ctrl-Q routines          }
{$I CMD.ED      } { general editing commands }

{$I K.ED        } { Ctrl-K dispatcher and interface             }
{$I O.ED        } { Ctrl-O dispatcher and interface             }
{$I Q.ED        } { Ctrl-Q dispatcher and interface             }
{$I DISP.ED     } { General command dispatcher                  }
{$I TASK.ED     } { Scheduling subsystem and central dispatcher }
{$I INPUT.ED    } { Input routines                              }

begin { program body }
  EditInitialize;  { Initialize dynamic structures }
  EditSystem;      { Use the default main loop     }
  ClrScr;
end.
```

While this program may be deceptively simple to build, it is actually an extremely powerful editor with multiple text windows, a WordStar-like command interface, an Undo command, and a full complement of useful text-editing features. We suggest that you compile and run FIRST-ED (it's in the file FIRST-ED.PAS on your Toolbox disk) and explore the functions of this editor. Remember that all of these features, and others that you create, will be available to you in any program you write with Toolbox!

A Brief Introduction to FIRST-ED

To compile FIRST-ED, start Turbo Pascal and specify FIRST-ED as the **M**ain file by typing the letter *M* followed by the file name FIRST-ED.PAS. You will want to compile FIRST-ED to a .COM file, so that it will have the maximum possible memory; to do this, type the sequence OCQC (**O**ptions, **C**OM file, **Q**uit options menu, **C**ompile). After Turbo Pascal finishes compiling FIRST-ED, type Q (for **Q**uit) to exit Turbo and FIRST-ED to run the editor.

When you start FIRST-ED, you will see a screen with a single status line. The status line gives the name of the file being edited, along with the line and column numbers of the cursor within the file. When first loaded, the status line should show that you are editing file NONAME in Window 1.

If you type some text into the window and experiment with it a bit, you will find that the command interface is much like WordStar. The WordStar cursor diamond works as expected, as do many of the other control keys and prefix/command combinations.

Certain commands, however, are missing, while others are new and unique to the Toolbox. For instance, Ctrl-OO, which does nothing in WordStar, creates a whole new editing window in FIRST-ED! Escape, the Undo command, is also new. Try deleting a few lines and restoring them using this feature—it is an excellent way to do a quick block move without marking the text.

We will leave you here with a quick command reference that will allow you to explore all of the features of FIRST-ED. We recommend that you pay special attention to some of the novel features of the Toolbox, such as the windowing commands. Remember that to exit FIRST-ED, you must use Ctrl-KX, *not* Ctrl-KD or Ctrl-KQ. Enjoy!

Here is a list of all of the commands for FIRST-ED; these are the default commands for any editor written with the Toolbox.

Table 5-1. FIRST-ED Quick Command Reference

Command	Description	Command	Description
^A	Left Word	^Q^C	To end of file
^S	Left Character	^Q^R	To top of file
^D	Right Character	^Q^I	Toggle Autoindent Mode
^F	Right Word	^Q^B	To Beginning of Block
^E	Up Line	^Q^K	To End of Block
^X	Down Line	^Q^J	Jump to Marker
^C	Down Page	^Q^A	Find and replace
^W	Scroll Up	^Q^F	Find pattern
^Z	Scroll Down	^Q^D	To right on line
^P	Insert Character by ASCII	^Q^S	To left on line
^J	Jump to Beg/End of Line	^Q^Y	Delete line right
RETURN	New Line	^Q 1	Jump to marker 1
^N	Insert Line	^Q 2	Jump to marker 2
^G, DEL	Delete Char	....	....
^I	Tab	^Q 9	Jump to marker 9
^R	Up Page		
^T	Delete Words	^K^B	Begin Block
^Y	Delete Line	^K^K	End Block
^B	Reformat Paragraph	^K^C	Copy Block
^V	Toggle Insert Mode	^K^V	Move Block
^L	Find Next Occurrence	^K^Y	Delete Block
^R	Up Page	^K^H	Hide Block
ESC	Undo	^K^R	Read File
		^K^W	Write File
^O^X	Down Window	^K^S	Save File
^O^G	Goto Window	^K^T	Set Tab Width
^O^E	Up Window	^K^X	Exit
^O^J	Link Window	^K^M	Set Marker
^O^Y	Delete Window	^K 1	Set marker 1
^O^O	Create Window	^K 2	Set marker 2
^O^W	Toggle Wordwrap Mode	....	....
^O^C	Center Line	^K 9	Set marker 9
^O^I	Jump to Column		
^O^N	Jump to Line		
^O^K	Change Case of Current Character		
^O^L	Set Left Margin		
^O^R	Set Right Margin		
^O^S	Set UNDO Limit		
^O 1	Jump to window 1		
^O 2	Jump to window 2		
....	....		
^O 9	Jump to window 9		

Chapter 6
CUSTOMIZING FIRST-ED

By now, we hope you have experimented with FIRST-ED and discovered some of its capabilities and limitations. You may want to re-map commands or install the function keys. You may also decide that you don't like FIRST-ED's screen layout, or the way it handles error messages, or even the colors in which it displays text! With the Turbo Editor Toolbox, all of the properties of the editor—the command interface, the screen display, even the representation of the text in memory or on the disk—may be changed to suit your whim and fancy. There are even hooks to add macro capabilities—and possibly a command language—if you so choose.

The Rules of the Game: Required Files, Procedures and Data Structures

To make an editor with the Editor Toolbox, you must merge the Toolbox source code with a simple main program. We suggest that you begin with the source for FIRST-ED (shown in Chapter 5), and build from there.

Your program should use the {$I} compiler directive to include each Toolbox module you are using. (Of course, you should only include the modules containing routines that you will use in your final version. For illustrative purposes, FIRST-ED uses them all.)

The first editor module contains the declarations of the most important variables and types that Toolbox uses internally, which must be included in any editor built with the Toolbox. It is:

VARS.ED	Global constants, types, and variables

Next, it is necessary to declare some required procedures, as follows:

```
procedure UserCommand    (var ch : byte);   {user cmd processor hook}
procedure UserError    (var Msgno : byte);   {user error handler hook}
procedure UserStatusline (var Wn : byte;   {user status line handler}
                    Column, Line : Integer);
procedure UserReplace    (var ch : byte); {user replace handler hook}
procedure UserTask;
```

As mentioned in Chapter 5, these procedures will allow you to add features to the editor without having to modify the Toolbox routines directly. They can be defined as null procedures (as they are in FIRST-ED) if not used. We will discuss how to use each of these procedures in subsequent sections.

Next, the editor *kernel* routines—important procedures that are called from many places throughout the code—must be included. These are contained in the following module:

USER.ED	Editor kernel routines

Following the kernel routines is the module containing the Toolbox screen routines. This module is called:

SCREEN.ED

After the screen routines, include the editor initialization code and the command processors:

INIT.ED	Initialization code
KCMD.ED	All overlaid Ctrl-K routines
OCMD.ED	All overlaid Ctrl-O routines
QCMD.ED	All overlaid Ctrl-Q routines
CMD.ED	Overlaid single-keystroke editing commands

Finally, the remaining modules should be included in the following order:

K.ED	Ctrl-K dispatcher and interface
O.ED	Ctrl-O dispatcher and interface
Q.ED	Ctrl-Q dispatcher and interface
DISP.ED	General command dispatcher
TASK.ED	Scheduling system and central dispatcher
INPUT.ED	Input routines

This ordering need not be followed exactly; however, deviating from it to any great extent may require **forward** declarations to be added to your program. In general, any procedure that is defined in the modules you include may be called from anywhere in your program, subject to the scope rules of Pascal.

If you want to make very major changes to FIRST-ED, you may decide to replace an entire Toolbox module with your own code. (In MicroStar, our sophisticated Toolbox editor, we replaced the command dispatchers EditK, EditO, and EditQ with new user interface procedures and a pulldown menu system.)

The Main Program

The main program of your editor should consist of:

- A procedure call to EditInitialize, to initialize the editor data structures
- A call to the Toolbox's predefined command-processing loop (EditSystem)
- Code to "clean up" the screen, and so on, after the editor is finished.

In some cases, you may wish to provide your own command-processing loop instead of *EditSystem*, the Toolbox default. This will rarely be necessary, however, since the Toolbox provides easy access to the command dispatching routines.

Customizing the Command Interface: The *UserCommand* Procedure

The procedure *UserCommand* is the simplest hook into the Toolbox command dispatchers. This procedure takes and returns a single byte as a **var** parameter, and is called by Toolbox whenever a new command character is entered from the keyboard.

UserCommand is given the opportunity to act on, and/or change, each control character before it gets to the main command dispatcher. Because the character is passed as a **var** parameter, *UserCommand* can act as a filter that transforms one keystroke, or sequence of keystrokes, into another. It can also dispatch or process commands itself.

After control has returned from *UserCommand,* the command keystroke is passed on to procedure *EditPrccmd,* the Toolbox default command dispatcher. If *UserCommand* has done all the processing that needs to be done for a keystroke, it can change the **var** parameter to the number 255 before returning a signal that the default command dispatcher is to do nothing. If *UserCommand* does not alter the keystroke, or changes it to a value other than 255, *EditPrcCmd* will process that character just as if it were typed on the keyboard.

A Simpler Editor—Using *UserCommand* to Disable FIRST-ED Commands

The following is an example of a *UserCommand* procedure that converts FIRST-ED into a very simple editor indeed—one that does not have any commands that are prefixed by Ctrl-K, Ctrl-O, or Ctrl-D.

```
procedure UserCommand (var Ch : byte);
begin
  if Ch = Ctrlk then                        { have Ctrl-K read a file }
    begin
      EditCprfw;                  { the predefined read file processor }
      Ch := 255;                                   { indicate it's done }
    end
  else if Ch = Ctrlq then                    { have Ctrl-Q write file }
    begin
      EditCpwfw;                 { the predefined write file processor }
      Ch := 255;                                   { indicate it's done }
    end
  else if Ch = Ctrlo then                        { disable this prefix }
    Ch := 255;
end; {UserCommand}             { all single-key commands remain the same }
```

Of course, disabling so many of the Toolbox commands would vastly diminish the power of FIRST-ED. However, if one were to actually make this change, it would be possible to delete the procedures which handle commands prefixed with Ctrl-K, Ctrl-O, and Ctrl-Q—EditK, EditO, and EditQ, respectively—as well as the code that processed these commands. Such surgery would result in a significantly smaller final program.

A Matter of Personal Preference—Using *UserCommand* to Filter Commands

UserCommand can also be used to configure certain editor keys the way you like them. For instance, in FIRST-ED, the Backspace key is destructive; that is, it deletes the character to the left of the cursor, as in the Turbo, SideKick, and SuperKey editors. However, if you are used to the default configuration of the program WordStar, you may want the backspace key simply to move the cursor to the left—a change that can be easily implemented in *UserCommand.* Just change the Ctrl-H character generated by the backspace key to a Ctrl-S (cursor left):

```
procedure UserCommand (var Ch : byte);
begin
  if Ch = Ctrlh then           { Backspace becomes cursor left }
     Ch := CtrlS;
end;  {UserCommand}          { all other commands remain the same }
```

A New Command

Suppose you want to write a new command not included in FIRST-ED.PAS. Imagine, for a moment, that FIRST-ED had no command to delete all text on the current line to the right of the cursor. Making one, while it requires some knowledge of the editor's internal data structures, is not difficult. Here is the way this command is implemented in the Toolbox:

```
procedure EditDeleteTextRight;

var
  i: integer;

begin                          { EditDeleteTextRight }
  with Curwin^ do
  begin
    if Colno <= Curline^.BuffLen then
    begin
      for i := Colno to Curline^.BuffLen do
        Curline^.Txt^. [i] := ' ' ;
      if not EditSizeline (Curline, Colno) then
```

```
        begin
          EditErrormsg (35);          { No more memory }
          exit;
        end
      end
    end
end;                               { EditDeleteTextRight }
```

In this example, the **with** statement indicates that all operations are to be performed within the current window, whose descriptor record is pointed to by the global pointer CurWin. If the cursor is already past all the text on the line, nothing happens. Otherwise, the **for** loop puts spaces in all columns on the current line (Curline), starting at the cursor column, Colno. The procedure then sets the line length to the column number, and calls EditSizeline to free any extra space.

Integrating a New Single-Keystroke Command

The above procedure could be called from anywhere in an editor or a program you have integrated with an editor. But—since you have intended it to be a command invoked by the user—you need to make it available from the keyboard. The best way to do this depends on the key sequence you choose to trigger the command.

If you want to invoke the command with a single keystroke, the procedure *UserCommand* would be the best way to add it to the user interface. Here's how to change the editor to make Ctrl-J delete all the text to the right of the cursor and leave all other commands the same.

```
procedure UserCommand (var Ch : byte);

begin {UserCommand}
  if Ch = Ctrlj then              { make Ctrl-J a special command }
    begin
      EditDeleteTextRight;              { call our own procedure }
      Ch := 255;                  { indicate the command's done }
    end
end; {UserCommand}
```

Note that this change would override the standard command associated with the Ctrl-J key (go to beginning/end of line). The old command would become inaccessible as a result of the change.

Integrating a New Prefixed Command

In the Editor Toolbox, *prefixed* commands (commands that consist of Ctrl-K, Ctrl-O, or Ctrl-Q followed by another keystroke) are handled in two steps. When the prefix is typed, control is passed to a special dispatching procedure that handles commands with that prefix (EditK, EditO, or EditQ, respectively). This procedure then prompts for the additional character, passing control to the appropriate command processing routine. Figure 6-1 shows the flow of control within the Toolbox for both prefixed and non-prefixed commands:

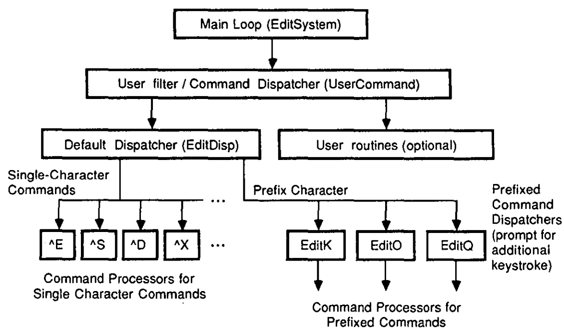


Figure 6-1. Editor Toolbox Command Dispatching: Flow of Control

Thus, while a single-keystroke command can be added by modifying *UserCommand,* creating or changing a prefixed command must be done by modifying the dispatcher for that prefix. When we wrote the Toolbox, we added our Ctrl-QY command by modifying the Ctrl-Q command dispatcher procedure *EditQ* (found in the file Q.ED) as shown below:

```
procedure EditQ;
{
1. Proc name.
   EditQ-perform window and searching command processing.

2. Functional description.
   This routine processes the Ctrl-Q command processing for the
   editor.  The Ctrl-Q command is simply a submenu of commands to
   which window and searching commands may be added.  You can remove
   commands from the editor by removing their references in the case
   statement in this procedure.  If all of the commands in a module
   are to be deleted (say, all window commands), then you can simply
   omit that include and comment out the references in this
   procedure.
}
```

```
var
  ch: byte;

begin                             { EditQ }
  EditAppcmdnam ('<Ctrl-Q>');
  ch := EditCtrlChar;
  case ch of                          { COMMAND DESCRIPTION }
    Ctrla : EditCpreplace;               { Find and Replace }
    Ctrlb : EditTopBlock;           { Cursor to beg. of block }
    Ctrlc : EditWindowBottomFile;        { Bottom of window }
    Ctrld : EditEndLine;              { End of current line }
    Ctrlf : EditCpfind;                      { Find pattern }
    Ctrli : EditToggleAutoindent ;     { Toggle autoindent mode }
    Ctrlj : EditCpjmpmrk;          { Jump to marker (prompt) }
    Ctrlk : EditBottomBlock;         { Cursor to end of block }
    Ctrlr : EditWindowTopFile;                { Top of window}
    Ctrls : EditBeginningLine;         { Beg. of current line }
    Ctrly : EditDeleteTextRight;          { Delete text to eol }
   One..Nine : EditJumpMarker(ch - Zero); {Jump to numbered marker }
  end
end;                                               { EditQ }
```

If you want to redefine many commands, the best course of action is usually to completely replace the default *EditQ*, *EditK*, and *EditO* procedures with your own.

A More Complex Editor

You can make very complex editors with the Toolbox by writing your own command and prefix command dispatchers, as well as additional command procedures. MicroStar, the powerful demonstration editor to be presented in the next chapter, is such a program. The command dispatchers have been extensively modified (MS.PAS includes its own EditK, EditO, and EditQ routines), and some commands have been written from scratch. We have implemented procedures to rename, copy, and delete files, a directory display similar to SideKick's, and a complete set of WordStar-compatible commands. There is a pull-down menu system (invoked by pressing the F10 key), and a background printing routine! All of these features were built on top of the basic Toolbox editor, using the "hooks" described earlier.

Chapter 7
MICROSTAR—A SOPHISTICATED EDITOR

To demonstrate the power and versatility of the Turbo Editor Toolbox, we have developed a "super-editor" using the Toolbox routines. We call it MicroStar, and it's in the file MS.PAS on your Toolbox disk.

MicroStar, a professional-quality editor in its own right, was designed to use—and show off—every feature of the Turbo Editor Toolbox. Some of the goodies you will find include:

- Wordstar command emulation
- An Undo facility (thanks to the Toolbox)
- Fast, multitasking background printing
- Sidekick-like directory display
- A complete pulldown menu system

...all with complete source code!

Getting to Know MicroStar

To compile MicroStar, start Turbo Pascal and specify MS as the **M**ain file by typing the letter M followed by the file name MS. You will want to compile MS to a .COM file, so that it will have the maximum possible memory; to do this, type the sequence OCQC (**O**ptions, **C**OM file, **Q**uit options menu, **C**ompile). After Turbo Pascal finishes compiling MS, type Q (**Q**uit) to exit Turbo and MS to run the editor.

Building Microstar

Compiling the file MS.PAS on your Toolbox disk will automatically pull in all of the necessary Toolbox routines to build MicroStar. All of these files are replaced by code in the body of MicroStar.

As with FIRST-ED, the first thing you should do is run MicroStar to get a feel for its user interface and text-handling capabilities.

MicroStar will display a pop-up window that asks for the name of a file to edit. You can enter a filename here, or simply hit the ESCAPE key. If you do the latter, then MicroStar will restrict your movements to the pulldown menus at the top of the screen; you may not type text until you have named the file you are going to create or edit.

Using the Pulldown Menu System

The arrow keys (or the WordStar cursor commands) will allow you to move along the menu bar. To "open" a pulldown menu, press the RETURN key, or type the letter corresponding to the capitalized letter in the menu name. The menu will open, and you will be able to select an item either by letter again, or by moving to your selection and pressing RETURN.

You can make a pulldown menu disappear by pressing the Escape key. If no pulldown menu is exposed, the next press of the Escape key will cause you to exit the pulldown menu system (assuming that you have a file open) and return to the text window.

The MicroStar Command Set

The MicroStar command set is similar to that of FIRST-ED, with some enhancements. Ctrl-KD and Ctrl-KQ work as expected. Background printing from a file is possible, and the Escape key will Undo your last deletion. Table 7-1 shows a complete list of the MicroStar commands:

Table 7-1. Micro-Star Quick Command Reference

Command	Description	Command	Description
^A	Left Word	^Q^S	To left on line
^S	Left Character	^Q^D	To right on line
^D	Right Character	^Q^J	Jump to marker
^F	Right Word	^Q^I	Toggle Autoindent Mode
^E	Up Line	^Q^R	To top of File
^X	Down Line	^Q^C	To end of File
^R	Up Page	^Q^B	To begin of Block
^C	Down Page	^Q^K	To end of Block
^W	Scroll Up	^Q 1	Jump to Marker 1
^Z	Scroll Down	^Q 2	Jump to Marker 2
RETURN	New Line	....	....
^N	Insert Line	^Q 9	Jump to Marker 9
^G, DEL	Delete Character	^Q^Y	Delete Line Right
BKSP	Delete Left Character	^Q^F	Find Pattern
^I	Tab	^Q^A	Find and Replace
^T	Delete Word		
^Y	Delete Line	^K^S	Save File & Resume
^B	Reformat Paragraph	^K^D	Save & Open file
^V	Toggle Insert Mode	^K^X	Exit to DOS
^L	Find Next Occurrence	^K^Q	Abandon & open file
^J	Beginning/End of line	^K 1	Set Marker 1
^P	Insert Char by ASCII	^K 2	Set Marker 2
		....	....
^O^O	Open new window	^K 9	Set Marker 9
^O^L	Set Left Margin	^K^B	Begin Block
^O^R	Set Right Margin	^K^K	End Block
^O^C	Center Line	^K^H	Hide Block
^O^K	Change Case	^K^C	Copy Block
^O^S	Set UNDO Limit	^K^V	Move Block
^O^W	Toggle Wordwrap Mode	^K^Y	Delete Block
^O^G	Goto other window	^K^W	Write Block
^O^I	Goto Column	^K^R	Read block
^O^N	Goto Line	^K^M	Set marker
^O^Y	Destroy window	^K^T	Define tab width
ESC	Undo last change	^K^L	Lightning spell check
F10	Activate Pulldown Menus		

Chapter 8
INSIDE MICROSTAR

In this chapter, we will share the secrets of the MicroStar editor, so that you, too, can create equally powerful text-editing systems using the Turbo Editor Toolbox. We will cover:

- The command dispatchers
- The pulldown menu system
- The pop-up window routines
- The background print spooler
- Customizing error handling
- Customizing the status display
- The "dirty" bit—detecting changes in the text

The Command Dispatchers

In Chapter 6, we showed graphically the flow of control within the Turbo Toolbox command dispatching routines. In MicroStar, two levels of the original hierarchy are replaced: the procedure *EditPrccmd,* which processes the characters filtered through *User-Command,* and *EditK, EditO,* and *EditQ,* which process the prefixed commands.

The command dispatchers in MicroStar underwent only minor changes from those in the Toolbox. Because of the different display requirements of MicroStar, we added the procedure *SecondChar* to get the second character of the command, rather than using *EditAp-pcmdnam.* We also eliminated the calls to *EditZapCmdnam* for the same reason. The *EditO* command dispatcher actually became simpler than the corresponding default Toolbox routine. Because MicroStar allows, at most, two windows to appear on the screen, the commands involving numbered windows were eliminated.

This command dispatcher changes only a few single-character commands from the original. The most important difference between this version and the standard Toolbox dispatcher is that the call to *Edit-ZapCmdname* has been removed; this prevents the Toolbox from blanking a portion of the screen that is no longer used for status or commands. If desired, however, any or all of the editor commands could have been changed by modifying this procedure.

The *EditK, EditO,* and *EditQ* procedures underwent more drastic modifications.

The new *EditK* supports the enhanced layout of the MicroStar screen display by replacing the call to *EditAppcmdnam* (which prompts for the second character for FIRST-ED's prefixed commands) with a new procedure, *SecondChar.* It also replaces the default "Block Move" combination, Ctrl-KM, with the more familiar Ctrl-KV. It implements the WordStar Ctrl-KD (Done) and Ctrl-KQ (Quit) commands, and allows the Ctrl-KX command to save the file before terminating the editor.

```
procedure EditK;
{ This routine is the utility supercommand processor for the
   editor.  The Ctrl-K command is simply a submenu of commands
   to which file I/O and block commands may be added. You can remove
   commands from the editor by removing their references in the case
   statement in this procedure.  If all of the commands in a module
   are to be deleted (say, all block commands), then you can simply
   omit that include and comment out the references in this procedure.
}
var
  ch :  byte;

begin                                                   { EditK }
  EditAppcmdnam ('<Ctrl-K>');
  ch := EditCtrlChar;
  case ch of                       { COMMAND DESCRIPTION         }
    Ctrlb : EditBlockBegin;        { begin block                 }
    Ctrlc : EditBlockCopy;         { copy block                  }
    Ctrlh : EditBlockHide;         { hide/display toggle block   }
    Ctrlk : EditBlockEnd;          { end block                   }
    Ctrlm : EditCpsetmrk;          { set marker (prompt)         }
    Ctrlr : EditCprfw;             { read file into window       }
    Ctrls : EditCpfilesave;        { save file from window       }
    Ctrlt : EditCptabdef;          { define tab width            }
    Ctrlv : EditBlockMove;         { move block                  }
    Ctrlw : EditCpwfw;             { write file from window      }
    Ctrlx : EditCpexit;            { exit editor                 }
    Ctrly : EditBlockDelete;       { delete block                }
 One..Nine: EditSetMarker(ch - Zero); { set marker #             }
  end
end;                                                      {EditK}
```

The new *EditO* also integrates the display routine *SecondChar,* and removes two of the original Toolbox commands Ctrl-OE (a nonstandard "jump to end of block" command) and Ctrl-OA (toggle "autoindent" mode—this was replaced by the Turbo-compatible Ctrl-QI).

```
procedure Edit0;
{  This routine processes the Ctrl-O command processing for the editor.  The Ctrl-O command
   is simply a submenu of commands to which text manipulation commands may be added. You can
   remove commands from the editor by removing their references in the case statement in this
   procedure.  If all of the commands in a module are to be deleted, then you can simply omit
   that include and comment out the references in this procedure.
}
var
   ch : byte;

begin                                              { Edit0 }
  EditAppcmdnam ('<Ctrl-O>');
  ch := EditCtrlChar;
  case ch of                              { COMMAND DESCRIPTION     }
    Ctrlc : EditCenterLine;               { center text             }
    Ctrle : EditWindowUp;                 { up window               }
    Ctrlg : EditCpgotowin;                { goto window (prompt)    }
    Ctrli : EditCpgotocl;                 { goto column i           }
    Ctrlj : EditCplnkwin;                 { link (join) window      }
    Ctrlk : EditChangeCase;               { change case             }
    Ctrll : EditCpsetlm;                  { set left margin         }
    Ctrln : EditCpgotoln;                 { goto line n             }
    Ctrlo : EditCpcrewin;                 { open new window         }
    Ctrlr : EditCpsetrm;                  { set right margin        }
    Ctrls : EditCpundlim;                 { set undo limit          }
    Ctrlw : EditToggleWordwrap;           { toggle word wrap mode   }
    Ctrlx : EditWindowDown;               { down window             }
    Ctrly : EditCpdelwin;                 { destroy window          }
 One..Nine: EditWindowGoto(ch - Zero);    { jump to window #        }
   end
end;                                                {Edit0}
```

The new *EditQ* integrates *Secondchar,* and eliminates the Toolbox windowing commands Ctrl-QL (Link Window), Ctrl-QE (Up Window), and Ctrl-QX (Down Window). It adds confirmation to the Ctrl-QY command (Delete Text in Window), so that the user who is accustomed to the WordStar command Ctrl-QY (Delete to End of Line) does not lose all of the text in the window without warning.

```
procedure EditQ;
{  This routine processes the Ctrl-Q command processing for the editor.  The Ctrl-Q command
   is simply a submenu of commands to which window and searching commands may be added. You
   can remove commands from the editor by removing their references in the case statement in
   this procedure.  If all of the commands in a module are to be deleted (say, all window
   commands), then you can simply omit that include and comment out the references in this
   procedure.
}

var
   ch: byte;

begin                                                { EditQ }
  EditAppcmdnam ('<Ctrl-Q>');
  ch := EditCtrlChar;
  case ch of                                { COMMAND DESCRIPTION     }
      Ctrla : EditCpreplace;                { Find and replace        }
      Ctrlb : EditTopBlock;                 { Cursor to beg. of block }
      Ctrlc : EditWindowBottomFile;         { Bottom of window        }
      Ctrld : EditEndLine;                  { End of current line     }
```

```
    Ctrlf : EditCpfind;                 { Find pattern           }
    Ctrli : EditToggleAutoindent;       { Toggle autoindent mode }
    Ctrlj : EditCpjmpmrk;               { Jump to marker (prompt) }
    Ctrlk : EditBottomBlock;            { Cursor to end of block }
    Ctrlr : EditWindowTopFile;          { Top of window          }
    Ctrls : EditBeginningLine;          { Beg of current line    }
    Ctrly : EditDeleteTextRight;        { Delete text to eol     }
  One..Nine: EditJumpMarker(ch - Zero); { Jump to numbered marker }
 end
end;                                                   { EditQ }
```

The Pulldown Menu System

The pulldown menu system included with MicroStar is a useful general-purpose utility that can be used in other programs as well.

The data structures giving the text strings for the pulldown menu system are defined as typed constants at the very beginning of the file MS.PAS. The name of the first constant string must be "st01"; the lengths and positions of the remaining items are calculated by the menu initialization routines.

The first group of strings gives the headlines to be placed in the "menu bar." Of course, there must be exactly as many strings in this group as there will be menus. The last headline is followed by a constant string set to null that indicates the end of the list, as shown below:

{The following strings are the headline of the pull-down menu. Insert only spaces before the section. If you need more than one space after the whole line, change the value of "NoSpaces." The length must match the actual length of the string.}

```
st01:string[06]='  Block';
st02:string[08]='  Search';
st03:string[07]='  Go to';
st04:string[13]='  Text format';
st05:string[08]='  Window';
st06:string[06]='  File';
st07:string[01]=''; {The headline and each submenu are terminated by an empty string with length of "1"}
```

The list of headlines is followed by an equal number of lists of menu entries, each also terminated by a null string. Note that the length of the first entry determines the width of the menu; it must be at least as wide as the other entries to prevent them from being truncated.

{Here comes the 1st submenu. The first entry has to be filled with spaces as it represents the maximum length of a submenu}

```
st11 : string[12] = 'Begin       ';
st12 : string[03] = 'End';
st13 : string[04] = 'Copy';
st14 : string[04] = 'Move';
st15 : string[04] = 'Read';
st16 : string[05] = 'Write';
st17 : string[06] = 'Delete';
st18 : string[09] = 'Hide/show';
st19 : string[12] = 'Spell check ';
st1x : string[01] = '';
```

Similar groups of strings are used to define the remaining pulldowns.

The menu initialization routine, *InitMainMenu* is called at the beginning of the program to display the menu bar. From this point on, the procedure *PulldownMenu* activates pulldown menu operation.

The items in the menus are linked to command processor routines through a large case statement in procedure *PulldownMenu*, as follows:

```
if ch = #13 then                         { a selection has been made }
case CurrSubmenu of
  1:case CurrSelection of
      1:EditBlockBegin;
      2:EditBlockEnd;
      3:EditBlockcopy;
      4:EditBlockmove;
      5:ReadBlock;
      6:if WriteFile(true) then;
      7:EditBlockDelete;
      8:EditBlockHide;
      9:SpellingCheck;
    end;
  2:case CurrSelection of
      1:FindRep(false,true);
      2:FindRep(true,true);
      3:if Replast then
          FindRep(true,false)
        else
          FindRep(false,false);
    end;
```

A menu item may be made to invoke any routine in the Toolbox through modifications to this case statement.

The *UserCommand* procedure is used to trigger MicroStar's pull-down menu routines when F10 is pressed. In the Toolbox, all of the special keys on the keyboard that do not have assigned functions generate a null character followed by the "scan code" of the key. The scan codes for some of the unused special keys are as follows:

F1-F10:	59-68
Shift + F1-F10:	84-93
Control + F1-F10:	94-103
Alt + F1-F10:	104-113
Ctrl + <—:	115
Ctrl + —>:	116
Ctrl + End:	117
Ctrl + PgDn:	118
Ctrl + Home:	119
Alt + 1-10:	120-129
Ctrl + PgUp:	132

UserCommand traps the F10 key and activates the menus as follows:

```
procedure UserCommand(var ch : byte);
{ This is the user command processor hook }
begin                              { procedure UserCommand }
  if ch = 0 then                          { Function key }
    begin
    ch := EditGetInput;
    if ch = 68 then                 { F10 pulls down menu }
      PulldownMenu;
    ch := 255;                     { Always absorb keycode }
    end;
end;                               { procedure UserCommand }
```

This is a very convenient way to integrate a menu system, since the command generated by the menu can be returned in the **var** parameter of *UserCommand* and processed immediately by *EditPrcCmd*. Multi-character commands can be returned by *UserCommand* by using the procedure *EditUserPush*; the pushed characters are automatically processed so that they do *not* pass through *UserCommand* again. The menu routines can also call command processor routines directly. In MicroStar, we used this approach because many of the menu items invoked low-level command processors directly.

The Pop-Up Window Routines

MicroStar uses pop-up windows to prompt for yes/no responses and other information from the user. Each window saves the information that was on the screen underneath it, and restores it when through. The routines:

procedure MakeWindow(col,line,nc,nl : byte);
and
procedure RestoreWindow(col,line,nc,nl : byte);

draw the windows and save and restore the screen contents.

The Background Print Routines

The Toolbox task scheduler, which is part of the loop invoked by the procedure *EditSystem,* allows the user to run background tasks while the editor is waiting for keyboard input. MicroStar specifies that the printing routines are to be run in the background by placing calls to them in the procedure *UserTask*:

```
procedure UserTask;
begin
  PrintNext;
end;
```

The procedure *PrintNext* is a simple state routine whose first line is: **if not** printing **then** exit;.

This provides a speedy return and prevents any noticeable system degradation when not printing. When a file is being printed, *PrintNext* will read characters from a buffer and send them to a printer until it detects input at the keyboard; it then returns control to the main editor loop.

Customizing Error Handling

The Turbo Editor Toolbox allows a user routine to take control of error processing in much the same way that it allows filtering of keyboard commands. When the toolbox detects an error, it passes control to the procedure *UserError,* with a **var** parameter indicating the type of error that occurred.

When *UserError* returns, the **var** parameter *Msgno* is passed to the standard Toolbox routine *EditErrormsg*. If *Msgno* is set to any value other than zero, *EditErrormsg* will produce an error message corresponding to that value, and display it in the upper left-hand corner of the screen. If the value is zero, the Toolbox assumes that the error has already been handled appropriately.

Because the standard Toolbox error-handling routines would place the error message in an unpleasing position on the MicroStar display, MicroStar's *UserError* routine unconditionally handles all error messages through MicroStar's own procedure, *ErrorCheck*:

```
procedure UserError {(var Msgno : byte)};  {user error handler hook}
begin
  if ErrorCheck(Msgno) then;
  Msgno := 0;
end;
```

ErrorCheck, in turn, calls the Toolbox function *EditMessage*, which retrieves the error message corresponding to a particular error number from an error message file. It then displays the error message in an aesthetically pleasing way on the screen.

The format of the Toolbox error message file, EDITERR.MSG, is designed to allow easy modification. Each line with a valid error message must begin with a three-digit error number, followed by a space and the message string, as follows:

```
*** This is a comment, and will be ignored

001  File does not exist
002  File is not open for input
003  File is not open for output
004  File is not open
005  File unreadable
006  File unwritable
016  Invalid numeric format
```

The error messages defined by the Toolbox include the Turbo Pascal file I/O errors—which are, in fact, numbered exactly the same in the Toolbox as they are in Turbo. You may use any of the unused numbers to define up to 255 total messages. MicroStar defines a number of messages that are pertinent to its own operations; these are displayed through direct calls to the procedure *ErrorCheck* when needed.

Customizing the Status Display

The Turbo Editor Toolbox also allows the user to redefine the way that the editor status (current line, column, window, etc.) is displayed on the screen. The "hook" for this operation is

procedure UserStatusline (**var** TWindow: byte; Column,line: integer);

Before displaying the current editor status, the Toolbox calls *UserStatusLine* with the pertinent information. If *UserStatusLine* returns with the window number TWindow greater than zero, the default Toolbox status display is used. If TWindow is returned as zero, then the Toolbox assumes that the user has displayed the information and does nothing.

MicroStar uses a customized *UserStatusLine* routine to display information for up to two windows in the upper right-hand corner of the user screen. TWindow is unconditionally set to zero to disable the default Toolbox routines.

The "Dirty" Bit—Detecting Changes in the Text

Most editors have a number of commands which, implicitly or explicitly, destroy the text stream currently in memory (such as Ctrl-K). Before executing such commands, it is desirable to prompt the user and ask whether it is OK to abandon any changes that might have been made before proceeding with the operation.

The Turbo Editor Toolbox provides a mechanism to do this through a global boolean variable called EditChangeFlag. This flag is set by all of the Toolbox command processors when they modify the text stream; the Toolbox itself performs no other operations on this flag. Your routines must set the bit in command processors that you define, and must clear it when a file has been successfully saved.

MicroStar manipulates EditChangeFlag in many routines throughout the file MS.PAS. Any editor written with the Toolbox should maintain the flag in a similar fashion to ensure that the user is warned appropriately of the consequences of his or her actions.

The background-printing routine in MicroStar does not process WordStar-style "dot" commands; however, it does ignore lines that begin with a dot so that WordStar files print in an attractive way. Because the printing routines go directly to the IBM PC BIOS when checking for keyboard input, they must be modified if the Toolbox is ported to a non-compatible computer.

Section III

HARNESSING THE FULL POWER OF THE TURBO EDITOR TOOLBOX

Chapter 9
TEXT AND WINDOW DATA STRUCTURES

This section of the manual describes the internals of the Turbo Editor Toolbox. With this information in hand, you will be able to tailor the architecture and features of the Toolbox to your needs.

How Text Data is Stored

In order to modify Toolbox, you must first understand how the editor stores and keeps track of the text stream(s) being edited. As mentioned earlier, streams of text are represented in Toolbox routines as linked lists of lines, each with a forward pointer, a backward pointer, a dynamically allocated string, a length word, and some flags. Represented pictorially, the structure as shown in Figure 9-1.

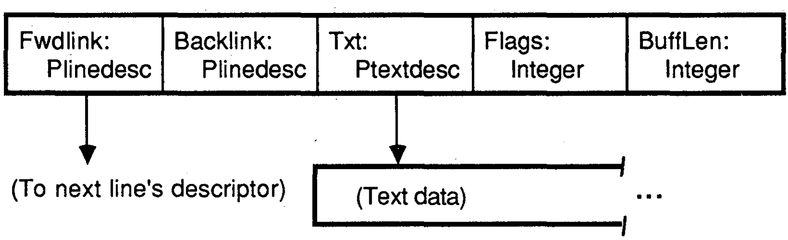


Figure 9-1. How the Editor Stores and Tracks Text Data

The types *Plinedesc* and *Ptextline* are pointers to objects of types *Linedesc* and *Textline*, respectively. The type *Textline* is defined in the module VARS.ED as a maximum-length Turbo Pascal string (255 characters).

The fields of the line descriptor record include the following:

Fwdlink. A pointer to the line immediately following. If the line in question is the last line in the text, this field is nil. To insert a line into a linked list of lines, use a section of code similar to this:

```
{assume line p is to be inserted above}
{line 9                              }
p^.Fwdlink := q^.Fwdlink;
q^.Fwdlink^.Backlink := p;
p^.Backlink := q;
p^.Backlink^.Fwdlink := p;
```

Backlink. A pointer to the line immediately preceding. If the line in question is the first line in the text, this field is nil. To insert a line into a linked list of lines, use a section of code similar to the previous example.

Txt. A pointer to a dynamically allocated string containing the text on the line. To get at the character in a particular column on a line, use an expression of the form *Line^.Txt^ [Column].*

Note that the objects pointed to by Txt are actually blocks of space that are allocated on the heap using Turbo Pascal's *Getmem* procedure, not variables of type *Textline.* If they *were* variables of type *Textline* every line would need 256 bytes reserved for it (255 characters plus a length byte), and no line could be over 255 characters long. This would waste space, and would also prevent the editor from being able to handle text files with very long lines. To sidestep these problems, the Toolbox takes advantage of the fact that all Turbo pointer types are compatible by allocating a block of memory for each line that is as long as (or slightly longer than) needed. It then points the Txt pointer to that area, and uses it to hold the string.

Flags. An integer containing various bits indicating whether the line is wordwrapped, in the block, or should be displayed in the special color *Usercolor.* To test for one of these properties, the display routines logically AND this field with the predefined constants *Inblock, Wrapped,* or *Colored.* Since only three of the 16 bits of this word are used, the remaining bits are available for your use.

BuffLen. An integer indicating the length of the text on the line. Because it is possible for the line to be longer than 255 characters, the first byte of the area pointed to by *Txt* may not be able to hold the length of the line. *BuffLen* is therefore used for this purpose.

How Windows are Managed

The Turbo Editor Toolbox maintains a linked list of records called *window descriptors* that specify the data relevant to each window. The window descriptor records are shown in Figure 9-2.

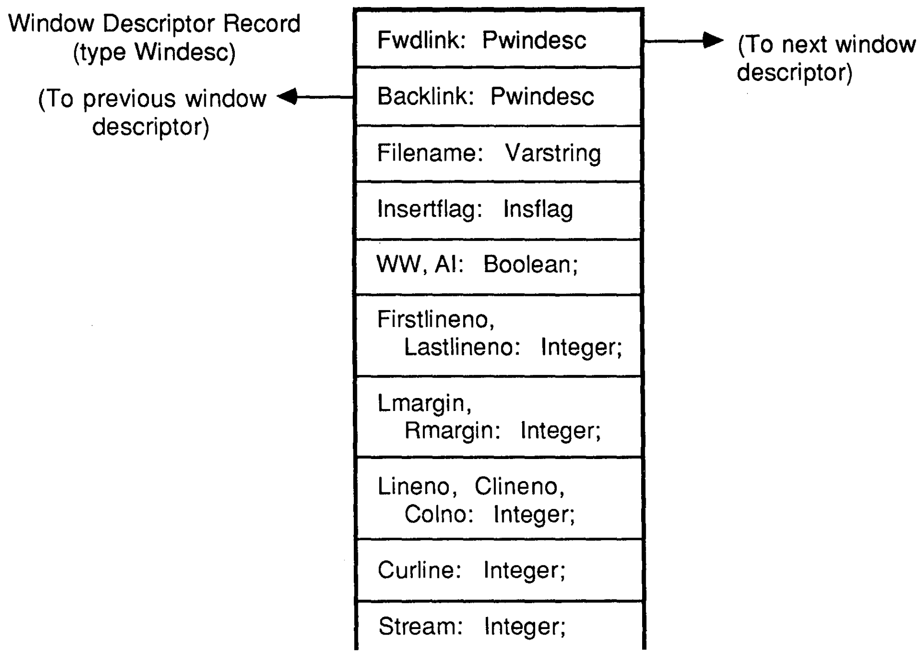


Figure 9-2. Window Descriptor Records

The type *Pwindesc* is a pointer to a window descriptor record, and *Varstring* is a Turbo string of length 80. The type *Insflag* is an enumerated type with the values (*Insert, Typeover*).

The fields of the window-descriptor record are detailed below.

Fwdlink. A pointer to the descriptor for the window below this one. Because the displayed windows are in a circular linked list, the *Fwdlink* field of the bottom window points to the top window. To insert a window into a linked list of windows, use a section of code similar to the following.

```
{ assume p is the window to be inserted, and q is the }
{window after which you wish p to be inserted        }

p^.Fwdlink := q^.Fwdlink;
q^.Fwdlink^.Backlink := p;
p^.Backlink := q;
p^.Backlink^.Fwdlink := p;
```

Backlink. A pointer to the descriptor for the window above this one. The displayed windows are in a circular linked list, so the *Backlink* field of the top window points to the bottom window.

Filename. A string which is set by *EditFileRead.* It contains the name of the file from which the text stream in the window was read.

Insertflag. A variable that indicates whether the window is in Insert or Overtype mode. When a window is in Insert mode, inserting new text pushes other text on the current line to the right; in Typeover mode, new text overwrites text that is already present.

WW. A boolean that indicates whether the window is in Wordwrap mode or not. When a window is in Wordwrap mode, any text typed past the currently defined right margin will wrap down to the next line and start at the currently defined left margin.

AI. A boolean that indicates whether the window is in Autoindent mode or not. When a window is in Autoindent mode, pressing the Return key positions the cursor under the first nonblank character on the previous line after inserting the new line. This mode is also present in the Turbo editor.

Firstlineno. An integer that indicates the top screen line the window owns; this is therefore the physical screen row that the status line will be put on. To change the displayed size of a window, the program must change the Firstlineno of one window and the Lastlineno of an adjacent window.

Lastlineno. An integer that indicates the last screen line the window "owns." If there is a window below this window, the lower window's default status line will be on the next line.

Lmargin, Rmargin. Integers that indicate the current margins. This information is used by *EditPrctxt* and *EditReformat* for wordwrapping.

Lineno. An integer that indicates the physical screen line of cursor.

Clineno. An integer that indicates the line in the text stream where the cursor is located, starting from the beginning of the text stream.

Colno. An integer that indicates the cursor screen column. This information is used to update the character under the cursor.

Topline. A pointer to a descriptor for the top text line in the window (used for vertical scrolling).

Curline. A pointer to the descriptor for the text line the cursor is on. The expression Curwin^.Curline^.Txt^ [Colno] references the character at the current cursor position.

Stream. A unique integer that identifies the window's text stream. When two windows are "linked," their stream fields are equated and their Curline fields point to lines in the same text stream.

Leftedge. An integer that indicates the column in the text currently displayed in physical screen column 1 (used for horizontal scrolling).

Two global variables of type *Pwindesc* are available in the editor: Window1 and Curwin. Window1 is the window displayed at the top of the screen, just below the command line. Curwin is the window the cursor is currently in; if the cursor is on the command line (asking for input, responding to an error message, and so forth), Curwin is the window the cursor was most recently in. All of the window and text data structures can be reached by traversing the lists referenced by these pointers.

Chapter 10
THE EDITOR KERNEL

This section describes the operation of the editor kernel: how keystrokes and commands are interpreted, how the screen is updated, and how multitasking is performed. This information will be useful if you need to modify the operation of the low-level editor routines, or if you want to alter the flow of control through the Toolbox.

Keyboard Input

All of the commands are taken from the keyboard. The routines that handle keyboard input directly are in the module *INPUT.ED.*

The *EditKeypressed* function checks DOS's typeahead buffer to see if the user has typed anything. It then calls *Pokechr* to push the character into the editor's typeahead buffer. At this point, the editor checks for the abort character Ctrl-U. If this occurs, the editor calls *EditAbort* to clear the typeahead buffer and display an error message. If *EditKeypressed* did receive input, it returns the value TRUE.

EditBreathe is a procedure that merely calls *EditKeypressed* and discards the return value. This is enough to get characters from the DOS typeahead buffer and place them in the editor typeahead buffer. In an operation that takes a long time or that may be interrupted with Ctrl-U (such as reading a file), it is desirable to call *EditBreathe* at least once in the main loop of the routine.

The Scheduler

The scheduler, contained in the procedure *EditSystem,* is the heart of the Toolbox editor. It determines when there is input in the editor typeahead buffer to be processed; it also calls the screen updating routines and background task(s) when there is nothing else to do. If it sees input in the typeahead buffer, it will take it and call *EditClsinp* to classify it. *EditClsinp* immediately calls *UserCommand* with the character. If *UserCommand* returns 255, *EditClsinp* does nothing more; otherwise, it classifies what *UserCommand* returned. If the character is a control character (ASCII 0-31) or delete character (ASCII 127), it calls *EditPrccmd* to process the character as a command. Otherwise, *EditClsinp* calls *EditPrctxt* to process the character as text.

The Primary Command Dispatchers

EditPrccmd is the general command dispatcher. It calls a specific command processor (or a prefixed command dispatcher) depending on the input it receives. This routine may be changed to modify the command mapping (as it is in MicroStar).

EditPrctxt is the text processor. When it receives a character, it inserts it into the current line (if in Insert mode) or replaces the current character with the parameter (if in Overwrite mode). It then moves the cursor right one character. If wordwrap mode is on and the text just added extends beyond the right margin, the text will be wrapped to the next line.

In most editing applications, it is useful to have a facility to insert control characters directly into the text. To do this, we use a command processor that asks the user for a character, converts that input to a control character, and then calls *EditPrctxt* directly—sidestepping the classification facility. The Toolbox default command for this function is Ctrl-P.

Prefixed Command Dispatchers

As mentioned in earlier chapters, the procedure *EditPrccmd* can invoke procedures that implement prefixed commands. In the default Toolbox editor, these commands use the prefixes Ctrl-K, Ctrl-O, and Ctrl-Q. The dispatchers that come with the Toolbox (*EditK*, *EditO*, and *EditQ*) can be rewritten to implement the prefixed commands differently (as done in MicroStar), or can be eliminated completely.

The *UserTask* "Hook"

The scheduler allows for command multitasking by repeatedly calling the procedure *UserTask* whenever the editor is waiting for input. Since the multitasking is not pre-emptive, the scheduler relies on the background task to return control whenever keyboard input is present, or when it has finished some incremental portion of its job.

The *UserTask* procedure can implement almost any background task. Some useful jobs that can be run in the background might be:

- Printing a file
- Redialing a remote computer
- Uploading or downloading files by modem
- Maintaining a log of system activity
- Performing automatic saves of a file being edited after predetermined intervals
- Backing up a hard disk

The only requirements for the background task code are: 1) It must be able to return control to the system quickly when editor input is detected, and 2) It must maintain its state (either in local typed constants or in global variables) so that it can resume operation the next time *UserTask* is called.

Chapter 11
THE EDITOR SCREEN ROUTINES

Screen Manipulation

The module *SCREEN.ED* contains all the routines that display text and other information on the screen. The key display routine is *EditUpdphyscr,* which updates all information on the screen. Other, more specific routines are used to update a single screen line, the status line of a window, or an entire window. The command line (the region in the upper left-hand corner of the screen) is also handled as a special case.

Default Screen Format

The default editor, FIRST-ED, is designed to display multiple text windows, using the top line to display commands and error messages. *EditAppcmdnam* and *EditZapcmdnam* are the two routines that affect this line; they both use a special string variable, *Cmdlinest,* to update the line's contents. The remainder of the screen is partitioned into one or more windows, each splitting the screen horizontally.

The top line of each window is called the *window status line.* This line displays information like the file name, cursor position, window number, and so on. Since there are 25 physical screen lines on an IBM display, the maximum number of displayed text lines is 23 (25 less one command line and one window status line). Further partitioning decreases display space but increases the number of different streams of text available at once.

All columns of the screen are used. The extreme right-most column (80) is used for special flag characters. The global variable *Logscrcols* controls this column. A left angle bracket indicates that the line was terminated with a carriage return. A period indicates that the line is past the end of the file. A blank indicates that the line was word-wrapped. If *Logscrcols = Physcrcols* (the physical number of columns on the screen), then the flag characters will not be displayed. If *Logscrcols* < *Physcrcols,* then flag characters will be displayed. Setting *Logscrcols* to less than *Pred* (*Physcrcols*) will not generate additional flag characters, but will starve the editor of display space. Because *Logscrcols* is a variable and not a constant, you may change it multiple times in an editor to turn the flags on and off. In MicroStar, the flags are left off at all times.

The Screen Updating Routines

Several levels of screen updating routines are provided. The highest level, *EditUpdphyscr*, updates the entire screen using lower-level routines. The first line it updates is the cursor line; then it updates the command line. If any input is ready, the routine will abort so that the command may be processed. Otherwise, the windows will be updated one by one starting with the current window. The updating process is interrupted by keyboard input, so that the user may perform scrolling commands (such as Ctrl-R and Ctrl-C) in rapid succession without waiting for the editor to "paint" each screen.

EditUpdwindow, an intermediate-level routine, is called by *EditUpdphyscr* to update individual windows on the screen. It uses the lowest-level routines to update each line in the window, and then calls *EditUpdwinsl* to update the status line. This routine can be used to update a single window by passing it a pointer to the window's window-descriptor record (described in Chapter 9).

EditUpdwinsl updates the status line of a single window. It is passed a *Pwindesc* pointing to the window's descriptor. It moves the file name, line and column numbers, window number, and labels for the flags into the status line. The various labels, like "Window:", are defined as constants in *VARS.ED*. You may change these constants to change the look of the status line. *EditUpdwinsl* is also called by *EditIncline* and *EditDecline*, which increment or decrement the line number and then display it immediately. You can also change *EditUpdwinsl* to remove certain information, add more information, or change the positions of labels.

If your display format differs greatly from that provided by the default Toolbox routines, you may wish to use the *UserStatusLine* procedure (described in Chapter 8) to display status information. *EditUpdwinsl* calls *UserStatusLine* with a window number and status information about the window. If the window number (a **var** parameter) is set to zero by *UserStatusLine, EditUpdwinsl* performs no further actions.

The lowest level screen routines are *EditWrline* and *EditUpdrowasm*. *EditWrline* is passed a string, a row number, and a color. It compares the string with the text on the physical screen on the row indicated. If the two are the same, it returns FALSE, indicating that the row does not need to be updated. If the strings are different, or if the color is different, *EditWrline* will change the Screen array to match the string and color, and then returns TRUE, indicating that the screen needs to be updated.

EditUpdrowasm is passed a row number to update. It copies the slice of the Screen array corresponding to that row directly into video memory, making screen updates very fast. If the global variable *Retracemode* is set, the routine will wait until the display card indicates that the display is performing a horizontal or vertical retrace operation so that the memory can be accessed without causing "snow" on the display. *Retracemode* need only be set to TRUE if the machine running the editor has an IBM (or equivalent) Color/Graphics adapter. The IBM Monochrome and Enhanced Color/Graphics cards do not require waiting for retrace.

The usual calling sequence is to call *EditWrline* to see if the screen array needs updating and, if so, update it; if *EditWrline* returns TRUE, call *EditUpdrowasm* to update the display.

Colors

The editor has five different attributes or combinations of color and intensity that are used for displaying ordinary text, blocks, window status lines, the command line, and special text such as menus. These color sets are variables, so they may be changed during an editing session. A simple command may be written to change the colors. The five colors are:

- Txtcolor ordinary text
- Bordcolor window status lines
- Blockcolor marked text
- Cmdcolor the command line
- Usercolor special text

Text may be marked *Colored* (displayed in *Usercolor*) by using the procedures *EditColorLine* and *EditColorFile*. *EditColorLine* sets the *Colored* flag for the current line; *EditColorFile* sets the *Colored* flag for every text line in the current window. This is useful if a menu or help file is to be displayed in a window, or to produce a strong contrast between windows.

Chapter 12
THE TOOLBOX COMMAND PROCESSORS

A *command processor* is a procedure that implements a Toolbox editing command. This chapter describes the different kinds of command processors available in the Toolbox. These routines can be called from a command dispatcher, from another command processor, or anywhere in a user program that has access to the global variables defined in *VARS.ED*. In general, the command processors that begin with *EditCp* are procedures that ask for input and then pass it to another procedure with a similar name (for instance, *EditCpcrewin* asks for input and calls *EditWindowCreate* to perform the operation). Command processors without a *Cp* in their names are usually passed the information they require in parameters or global variables.

Cursor Movement Commands

The movement commands do not change text; they simply move the cursor. Predefined commands can move the cursor left or right a single character or word, up or down a single line or screenful, or scroll the current window a single line. There are also predefined commands to move to the top or bottom of the text. Most movement commands only affect the current window. Many commands can be simulated by repetitive calling of these procedures; for instance, you can create a command to move the cursor left one word, which will move left one character repeatedly until the current character is a space.

Summary of movement command procedures:

EditLeftChar	*EditRightChar*
EditUpLine	*EditDownLine*
EditScrollUp	*EditScrollDown*
EditUpPage	*EditDownPage*
EditLeftWord	*EditRightWord*
EditBeginning Line	*EditEndLine*
EditWindowTopFile	*EditWindowBottomFile*
EditTopBlock	*EditBottomBlock*
EditCpgotoln	*EditGotoLine*
EditCpgotocl	*EditGotoCol*
EditBeginningEndLine	*EditGotoColumn*

An example of a movement command is the predefined *EditRightChar* command procedure. This procedure is called by the general command dispatcher.

```
procedure EditRightChar;
begin                          {EditRightChar}
  with Curwin^ do
    if Colno < Pred (Maxint) then
      Colno := Succ (Colno)
end;                           {EditRightChar}
```

Text Deletion Commands

Commands are defined in the Toolbox to remove text from a text stream on a character-by-character or line-by-line basis. Commands exist to delete a single character, a single word, a single line, or all the text in a window. Like movement commands, these commands may be repeated to simulate other operations. For example, *EditDeleteRightChar*, which deletes the character under the cursor, can be simulated by calling *EditRightChar* to move past the character, and then calling *EditDeleteLeftChar* to delete it. Most of these commands affect only the current window.

Summary of text deletion command processors:

EditDeleteRightChar	*EditDeleteLeftChar*
EditDeleteLine	*EditWindowDeleteText*
EditBlockDelete	EditDeleteTextRight
EditDeleteRightWord	

An example of a text manipulation command is the predefined *EditDeleteRightChar* procedure:

```
procedure EditDeleteRightChar;
{ This routine deletes the character underneath the cursor. }
var  i, j : integer;

begin                               { EditDeleteRightChar }
  EditChangeFlag := true;
  with Curwin^ do
    begin
      if Colno >= Curline^.BuffLen then
        begin
          if not EditSizeline (Curline, Succ (Colno)) then
            begin
              EditErrormsg (35);
              exit
            end
        end;

      with Curline^ do
        begin
          i := BuffLen;
          while (i > 1) and (Txt^ [i] = ' ') do i := Pred (i);
          for j := Colno to Pred (BuffLen) do
            Txt^ [j] := Txt^ [Succ (j)];
          Txt^ [BuffLen] := ' ';
          if (Colno > i) or (i = 1) then
            if EditJoinline then;          { try to join lines }
        end
    end
end;                                { EditDeleteRightChar }
```

Word Processing Commands

The next group of command processors supports general word processing functions.

Some routines affect only a single character, such as the change case command. Others affect only the current line, such as the center line command. The wordwrapping commands affect a paragraph. The text markers provided may be set anywhere in any window, and may be jumped to and from different windows or files. There are routines to jump to specific places, such as to the beginning or end of a block, or to a specific column or line. Also included is an Undo facility, with which you can undo line or block deletions anywhere, or decide how many lines the editor should retain for this facility.

Any of these routines may be called from a command dispatcher or any procedure in the editor. As mentioned earlier, those commands prefixed with *Cp* are procedures that ask the user for input. Their counterparts take that information as a parameter.

Summary of text processing command procedures:

EditReplace
EditChangeCase
EditCpReplace
EditSetLeftMargin
EditSetRightMargin
EditToggleAutoindent
EditReformat
EditUndo
EditCpsetlm
EditCpsetrm
ToggleInsert
EditDefineTab
EditCptabdef
EditToggleInsert
EditCenterLine
EditFind
EditJumpMarker
EditSetMarker
EditSetUndoLimit
EditToggleWordwrap
EditInsertCtrlChar
EditCpjmpmrk
EditCpsetmrk
EditCpundlim
EditInsertLine
EditTab
EditCpFind

An example of a text processing command procedure is the predefined *EditChangeCase* procedure.

```
procedure EditChangeCase;
{ This routine checks the character at the current cursor position.
  If it is an alphabetic character, it changes the case from upper
  to lower case or vice versa.  If the character is not alphabetic,
  no action is performed.
}
var
  Ch : char;

begin                                         { EditChangeCase }
  EditChangeFlag := true;
  with Curwin^ do
    begin
      Ch := Curline^.Txt^ [Colno];
      if (Ch >= 'A') AND (Ch <= 'Z') then
        Ch := Chr (ord (Ch) + 32)
      else if (Ch >= 'a') and (Ch <= 'z') then
        Ch := Chr (ord (Ch) - 32);
      Curline^.Txt^ [Colno] := Ch
    end
end;                                          { EditChangeCase }
```

Multiple Windows and Text Buffers

Sometimes it is useful to view several files at the same time, or even edit them at the same time. For example, suppose you have a Turbo Pascal program that calls a routine in this Toolbox but you'd like to see the source for MicroStar (MS.PAS) to see how to call the routine. Instead of leaving in the middle of your work by saving it, and opening a new file, it would be nice to be able to partition the screen to see part of the program you're working on in one window, and the source to MS.PAS in the other.

This is made possible with the Editor Toolbox window management routines. The Toolbox keeps track of a window on the screen through a window descriptor record. As we mentioned in Chapter 9, this record supplies the contents and position of the window, the filename of the text being edited, the starting row number of the window, and how many lines it uses on the display. It also contains mode flags that indicate whether wordwrapping, autoindenting, etc., is to be performed while editing in that window.

In the Toolbox, it is possible to create as many windows as can be contained in memory, and even have them share text streams. This is called *window linking*. The toolbox has routines to manage window linking automatically, so that operations performed in one window cause the displays in linked windows to be updated if necessary.

Windows are normally created by calling the default Toolbox window routines. Of course, you can define windows yourself without calling the Toolbox routines to do it, and create a text stream for those windows. However, these windows will not be displayed on the screen unless they are linked into the system's list of windows.

Windows of text that are not displayed can be used to store text in memory for later use. A simple procedure can be written that makes the window visible on the screen, and another that removes it from the screen; this can be helpful if the user is editing many small files and does not want (or need) to see every one at once.

Window Commands

The Toolbox has high-level and low-level routines to manipulate windows. *EditCpcrewin* and *EditCpdelwin* are procedures that ask the user for data concerning a window to be created or deleted, and then perform the operation. *EditWindowCreate* and *EditWindowDelete* are procedures that are passed this information as parameters. The lowest level routine, *EditCrewindow*, does not reorganize the display as it creates a new window; it merely initializes a record. An editor can use any combination of these routines to modify windows. A command dispatcher might have one command call, *EditCpcrewin*, to ask the user about creating a window, and another command call, *EditWindowCreate*, with a fixed value. These different levels give you maximum flexibility.

Summary of window manipulation command processors:

EditWindowCreate	*EditWindowDelete*
EditWindowTopFile	*EditWindowBottomFile*
EditWindowDeleteText	*EditWindowLink*
EditWindowUp	*EditWindowDown*
EditWindowGoto	*EditCpgotowin*
EditCpcrewin	*EditCpdelwin*
EditCplnkwin	

An example of a window manipulation command is the predefined *EditWindowCreate* procedure.

```
procedure EditWindowCreate (Size : byte; Win : byte);
{  This routine creates a window of the specified size, taking
   the lines from the window specified by Win.
}
var  i       : integer;
     p, q    : Pwindesc;

begin                                        { EditWindowCreate }
  if Size >= 3 then
    begin
      if Win < 1 then exit;
      p := Window1;
      i := 1;
      while i < Win do
        begin
          p := p^.Fwdlink;
          i := Succ (i)
        end;

      { Make a new window structure }

      q := EditCrewindow (Succ (p^.Lastlineno-Size),
                          Size,
                          Nofile,
                          Line1,
                          Col1);
      if q = nil then
        begin                               { No memory for window }
          EditErrormsg (35);
          exit
        end;

      if p^.Lastlineno-p^.Firstlineno - Size >= 1 then
        begin                        { If window can be compressed }
          p^.Lastlineno := p^.Lastlineno - Size;

           { we may be positioned outside the window's area now }

          while p^.Lineno > (p^.Lastlineno - p^.Firstlineno) do
            begin                            { Fix up the pointers }
              p^.Lineno := Pred (p^.Lineno);
              p^.Curline := p^.Curline^.Backlink
            end;
```

```
        q^.Backlink := p;
        q^.Fwdlink := p^.Fwdlink;
        p^.Fwdlink^.Backlink := q;
        p^.Fwdlink := q
      end
    else
      EditErrormsg (22);
  end
end;                                      { EditWindowCreate }
```

Block Commands

A block is a contiguous series of lines in a piece of text. There can only be one block defined at any one time, but it may be accessed from another window or another file. A block is defined by marking its first line and last line, using the predefined procedures *EditBlockBegin* and *EditBlockEnd.* These can be called from a command dispatcher or directly from a procedure. Once the block is marked, all lines between the two markers are labeled with the *Inblock* flag, which tells the screen updating routines that the lines should be displayed in *Blockcolor.*

There are three basic operations that can be performed on blocks. A block may be moved; that is, its contents deleted from their current location and inserted at a new location. A block may be copied so that the block text is inserted at a new location but the original copy is not changed. A block may also be deleted from the text. All block operations will operate from another window or file. These manipulation procedures can be called from a command dispatcher or another procedure.

Summary of block manipulation command processors:

EditBlockBegin	*EditBlockEnd*
EditBlockCopy	*EditBlockMove*
EditBlockDelete	*EditBlockHide*

An example of a block manipulation command is the predefined *EditBlockBegin* procedure.

```
procedure EditBlockBegin;
var
  p : Plinedesc;
```

```
begin                                          { EditBlockBegin }
  EditOffblock;                    { Turn off all lines in blocks }
  with Curwin^ do
    Blockfrom := Curline;    { Repoint the beginning of the block }


            { If end is not nil, then turn on the block display,
              provided we are still in the same text stream. Scan
              now for stream contiguity                           }


  if Blockto = nil then
    begin                              { Only beginning defined }
      Blockhide := TRUE;
      exit
    end;
  p := Blockfrom;
  while p <> nil do
    begin                                                { Scan }
      if p = Blockto then
        begin
          Blockhide := false;          { Turns on block display }
          exit
        end
      else
        p := p^.Fwdlink                    { Scan next linedesc }
    end;
  Blockto := nil;                   { Markblk won't mark anymore }
  Blockhide := true                     { Different streams now }
end;                                           { EditBlockBegin }
```

An example of a command processor that could duplicate the current line might look like this.

```
procedure DupLine;

begin
  EditBlockBegin;          { mark the block at the current line }
  EditBlockEnd;
  EditBlockCopy;                           { duplicate the line }
  EditBlockHide;                 { turn off block highlighting }
end;
```

File Commands

These commands let you read and write files. A file may be read into a window, or the text in a window may be written to a file. There are two levels of file access: *EditCprfw* and *EditCpwfw,* which ask the user for input and then operate on that input, and *EditFileRead* and *EditFileWrite,* which are passed a filename as a parameter to operate on. Thus, a command dispatcher might call *EditCprfw* to read a file, and an initializtion routine might call *EditFileRead* to read in a message or help file. Both levels of access may be used interchangeably within an editor.

The basic file I/O routine is EditReatxtfil. It opens a file, and uses block I/O (unrelated to editor blocks) to read its text. It interprets a carriage return from the upper ASCII set (with the high bit set) as a wordwrapped line. The file writing commands check for the Wrapped flag as they write, and terminate wordwrapped lines with this same character. You can modify these routines to do I/O the way you want, or to do automatic buffering of text to disk.

Summary of file manipulation command processors:

EditFileRead	*EditFileWrite*
EditFileSave	*EditCprfw*
EditCpwfw	*EditReatxtfil*
EditCpFilesave	

An example of a file manipulation command is the predefined *EditReatxtfil* procedure.

```
procedure EditReatxtfil (Fn : Varstring);
{ This routine opens the file specified and copies it line by
  line into the current window's text stream.
}
const
 Bufsize = 512;

var
 Error      : boolean;
 Nulln      : boolean;
 Endoffile  : boolean;                    { Set by read routine }
 Endofline  : boolean;
 Ch         : char;
 Nrecsread  : integer;                  { Bytes read by read routine }
 Infile     : file;
 Colnosave  : integer;
 Linenosave : integer;
 i          : integer;
 Pointer    : integer;                 { Next byte to read in buffer }
 Topsave    : Plinedesc;
 Textsave   : Plinedesc;
 Filnam     : String80;
 Buffer     : array [1..Bufsize] of char;
 x          : real;                        { Number of bytes to read }

begin                                                     { Reatxtfil }
 Assign (Infile, Fn);
 Reset (Infile,1);
 x := longfilesize(InFile);            { Get number of bytes to read }
 if EditFileerror then exit;
 with Curwin^ do
 begin
   if Filename = Nofile then
     Filename := Fn;
   Nulln := true;
   for i := 1 to Curline^.BuffLen do
     if Curline^.Txt^ [i] <> ' ' then Nulln := false;
   if (Curline^.Backlink = nil) and
      (Curline^.Fwdlink = nil)  and Nulln then
   begin
   end
```

```
  else if not EditInsbuf (1) then                       { Inserts line }
  begin
    EditErrormsg (40);
    Close (Infile);
    exit
  end;
  Topsave := Topline;                               { Save cursor position }
  Textsave := Curline;
  Colnosave := Colno;
  Linenosave := Lineno
end;
Error := false;
Pointer := Succ (Bufsize);                   { Force read on the first time }
Nrecsread := 0;
Endoffile := false;
repeat
  with Curwin^ do
  begin
    Error := Error or Abortcmd;
    Endofline := false;
    repeat                                         { Get next char in line }
      if Pointer > Nrecsread then
      begin                                { Need to load another buffer full }
        if x > BufSize then
        begin
          Blockread (Infile, Buffer, Bufsize, Nrecsread);
          x := x - Nrecsread;                   { Number of bytes left to read }
        end
        else
          Blockread (Infile, Buffer, trunc(x), Nrecsread);
        Error := Error or EditFileerror;
        Pointer := 1
      end;
      if Nrecsread = 0 then
        Ch := chr (Ctrlz)
      else
```

```
begin
  Ch := Buffer [Pointer];
  Pointer := Succ (Pointer)
end;
case ord (Ch) of
  Ctrlm : begin end;
  141   : Curline^.Flags := Curline^.Flags or Wrapped;
  Ctrlj : begin
            if not EditInsbuf (1) then
            begin
              EditErrormsg (40);
              Error := true
            end
            else
              Colno := 1
          end;
  Ctrlz : begin
            Endofline := true;
            Endoffile := true
          end;
else
begin
  if not EditSizeline (Curline, Succ (Colno)) then
  begin
    EditErrormsg (40);
    Error := true
  end
  else
    begin
      Curline^.Txt^ [Colno] := Ch;
      Colno := Succ (Colno)
    end
end
```

```
    end
    until Error or Endofline;
    if (not Endoffile) and (not Error) then {Don't add line if not needed}
      if not EditInsbuf (1) then
      begin                          { Out of memory in middle of file }
        EditErrormsg (40);
        Error := true
      end
  end                                                          { with }
  until Endoffile or Error;
  Close (Infile);
  with Curwin^ do                   { Restore original cursor position }
  begin
    Topline := Topsave;
    Curline := Textsave;
    Colno := Colnosave;
    Lineno := Linenosave
  end;
end;                                                   { EditReatxtfil }
```

Exit Commands

These command processors allow a graceful exit from the editor.

Summary of exit command processors:

EditExit	*EditCpExit*

Chapter 13
OVERLAYING YOUR EDITOR

Because the Turbo Editor Toolbox has so many useful features, it is possible to exceed Turbo Pascal's 64K code size limit when building a sophiticated editor such as MicroStar. Such occurrences are even more likely if the Toolbox editor is combined with another application, such as a database or a spreadsheet.

The Turbo Pascal overlay system, if used correctly, can allow you to create programs much larger than 64K with very little degradation in performance. This system works by allowing any number of separate procedures or functions in your program to share the same area of memory, with the required procedures brought in as needed.

Creating Overlay Groups

An *overlay group* is a group of routines that share the same space in memory. To create an overlay group, you simply precede each of a series of successive procedure and/or function declarations with the reserved word **overlay**, as follows:

```
overlay procedure a;
  begin
    .
    .
  end;

overlay procedure b;
  begin
    .
    .
  end;

overlay procedure c;
  begin
    .
    .
  end;
```

In this example, the three procedures just defined would all share the same space in memory. When compiling this program, Turbo would create an *overlay file* with a three-digit extension to hold the code for the procedures in the overlay group. For instance, if these procedures comprised the first overlay group of a program called TEST.COM, these procedures would be compiled into the file TEST.000.

Any procedure or function declaration that does *not* contain the word **overlay** ends an overlay group. Thus, if the above procedures were followed by

```
procedure e;
  begin
   .
   .
  end;

overlay procedure f;
  begin
   .
   .
  end;

overlay procedure g;
  begin
   .
   .
  end;
```

procedure f would *not* share memory with procedures a, b, and c—but only with procedure g. These two procedures would generate a separate overlay file, TEST.001.

Minimizing Thrashing

When a number of procedures in the same overlay group are called in rapid succession, many reads of the program disk may be necessary. This phenomenon is called *thrashing*, and results in greatly reduced performance (especially on floppy-based systems). Avoiding this problem requires careful consideration when designing the overlay structure for a program.

Thrashing can be avoided by putting seldom-used code (such as initialization routines) or sections of code in which the user will likely remain for some period of time (such as a find-and-replace procedure) in overlays. In an editor, for instance, it would be poor practice to overlay the cursor movement routines—but help facilities, directory listing programs, and file handling procedures could be placed in overlays without significantly affecting the utility of the program.

The Toolbox Overlay Structure

The first sample editor, FIRST-ED, does not use overlays. In MicroStar, there is one overlay group that consists of many of the commands invoked by the Ctrl-O, Ctrl-K, and Ctrl-Q prefixes (and also a few single-keystroke commands, such as Reformat Text). Because these commands are not as frequently used as the nonprefixed commands (and because many of them access the disk anyway), the performance of the Toolbox does not suffer noticeably as a result.

To add your own procedures to this overlay group, insert your overlay procedures and functions immediately before or after the overlay group indicated in the source of MS.PAS.

Caveats Regarding Overlays

There are reasons, however, why you may *not* want to place your procedures in the same overlay group as the Toolbox routines. The most important reason to avoid this practice is that *procedures in the same overlay group cannot call one another.* If one of your procedures is placed in the Toolbox overlay group, and attempts to call one of the other routines there, your program will crash without warning!

Two other restrictions also apply to overlay procedures and functions. First, they cannot be declared **forward** (though you *may* declare another **forward** procedure which, in turn, calls the overlay procedure). Second, they cannot be recursive (this counts as a call to a procedure in the same overlay group).

If these practices are avoided, overlay procedures will prove to be a useful and efficient way to add more power to your Turbo programs. For additional information on overlays and how they work, consult the *Turbo Pascal Reference Manual.*

Chapter 14
INCLUDING AN EDITOR IN YOUR PROGRAM

Because the Turbo Editor Toolbox comes with complete source code, you can recompile the Toolbox modules as part of a larger program that you design. This chapter discusses the different strategies you can use to accomplish a smooth integration.

Including the Toolbox Directly in Your Code

The simplest and most straightforward way to integrate the Toolbox with your program is to simply include all the modules in the same order that they are included in FIRST-ED.PAS or MS.PAS. This technique is best when your program and the Toolbox routines that you use can both fit in memory at the same time.

If this technique is used, be sure that the Toolbox modules are included in the same order that was described in Chapter 6, and that *EditInitialize* is called before invoking the editor's scheduler for the first time.

Overlaying the Editor with Your Program

A slightly more complex way to merge the Toolbox with your program is to overlay an editor that you create with the rest of your code. Since Turbo Pascal allows overlay procedures to be nested, it is possible to make a Toolbox editor, such as FIRST- ED, into a procedure that shares memory with your program. Data can be passed back to your program through the heap, through global variables, or through text files. This method allows better use of memory than including the Toolbox routines in your program—but, like that method, entails long compile times when debugging and/or testing new features.

Making the Editor a Chain File

It is also possible to compile a Toolbox editor as a Turbo Pascal *chain file*—a special file with the extension .CHN that can be loaded and run by your main program. Because a program run from a chain file can share global variables with the main program, data is easily passed back and forth between the editor and other parts of the program.

A big advantage of chain files that developers will appreciate is that they are compiled separately from the main program. Since the Toolbox is a substantial piece of code, this may save you many minutes of waiting as Turbo recompiles everything.

If you *do* use the editor as a chain file, be sure that you carefully read the documentation in the *Turbo Pascal Reference Manual* regarding the requirements for chain files. In particular, note that you must manually set the sizes of the code and data areas for the main program to be greater than or equal to that of the chained program—this is done by entering the largest value of each into the Turbo Pascal Options menu at compile time. Also, make sure that all global variables are declared the same way and in the same order at the beginning of each program (an include file is useful for this purpose), so that data can be shared and the programs do not overwrite one another's variables.

Invoking the Editor with the Execute Procedure

Another method, which also allows your program to be compiled separately from the editor modules, is to invoke the editor (compiled as a normal .COM file) as a separate program using Turbo's *Execute* procedure. This method does not allow variables to be shared, but may save space if not much sharing is needed between the programs. The programs can pass data back and forth via the file system, and can also send brief messages to one another through an *absolute* variable at the address Cseg:$80. The size constraints for programs invoked with the *Execute* procedure are the same as for chained programs. For more information on using the *Execute* procedure, see the *Turbo Pascal Reference Manual.*

Section IV

Turbo Editor Toolbox Technical Reference

Chapter 15
TURBO EDITOR TOOLBOX FILES

This part of the manual provides detailed technical information about all the routines contained in the Turbo Editor Toolbox. The first section lists all of the modular files that you'll need to include in your application programs; the following section describes the constants, types, and variables used by the Toolbox. Finally, there is a complete listing of the procedures and functions contained in the package.

The Turbo Editor Toolbox Distribution Diskettes

The Turbo Editor Toolbox is provided on two diskettes. Disk #1 contains the Toolbox itself, the FIRST-ED demonstration program, and the README files. Disk #2 contains the MicroStar editor, complete with a series of .INC and .OVL files that overlay the Toolbox command processor routines for MicroStar.

Files Included on Disk #1

README.COM	A program which lists the READ.ME file. You should run this program immediately upon receiving your distribution diskettes.
READ.ME	This file contains a brief list of all the files on the Turbo Editor Toolbox distribution diskettes, and details of any last-minute changes or additions to the Toolbox.
VARS.ED	A module containing constant, type, and variable declarations for the Toolbox routines.
USER.ED	A module containing low-level routines used throughout the Toolbox.
SCREEN.ED	A module containing the Toolbox routines that manipulate the screen during editing.
INIT.ED	A module containing the initialization routine for the Toolbox.

KCMD.ED	A module containing the command processor routines for commands prefixed by ^K in the default Toolbox command set. These commands deal generally (but not exclusively) with files and blocks of text.
OCMD.ED	A module containing the command processor routines for commands prefixed by ^O in the default Toolbox command set. These commands deal generally (but not exclusively) with the way text appears on the screen.
QCMD.ED	A module containing the command processor routines for commands prefixed by ^Q in the default Toolbox command set. These commands deal generally (but not exclusively) with quick movements through the text.
CMD.ED	A module containing the command processor routines for commands invoked by a single keystroke in the default Toolbox command set.
K.ED	A module containing the command dispatcher routines for the command processors in KCMD.ED. Also included here are the command "preprocessor" routines (with names including the letters "Cp") which ask for input prior to invoking the command processors in KCMD.ED.
O.ED	A module containing the command dispatcher routines for the command processors in OCMD.ED. Also included here are the command "preprocessor" routines (with names including the letters "Cp") which ask for input prior to invoking the command processors in OCMD.ED.
Q.ED	A module containing the command dispatcher routines for the command processors in QCMD.ED. Also included here are the command "preprocessor" routines (with names including the letters "Cp") which ask for input prior to invoking the command processors in QCMD.ED.

DISP.ED	A module containing the main command dispatcher for the Editor Toolbox commands.
TASK.ED	A module containing the task scheduler and main command loop for the Toolbox editors.
INPUT.ED	A module containing the keyboard input routines for the Toolbox.
EDITERR.MSG	A file containing the error messages for the Toolbox editors.
FIRST-ED.PAS	The source code for the default Toolbox editor, FIRST-ED.
MS.COM	Compiled code for the sample editor, MicroStar.
MS.000	Overlay file for MicroStar.
FIRST-ED.COM	Compiled code for FIRST-ED.

Files Included on Disk #2

MS.PAS	The source code for MicroStar, a sophisticated editor built with the Toolbox.
VARS.MS	A module containing constant, type, and variable declarations for the Toolbox routines.
USER.MS	A module containing low-level routines used throughout the Toolbox.
SCREEN.MS	A module containing the Toolbox routines that manipulate the screen during editing.
FASTCMD.MS	A module containing the command processor routines which are *not* overlaid in the MicroStar editor. These include both prefixed and non-prefixed commands.
INIT.MS	A module containing the initilization routine for the Toolbox as an overlay procedure.

KCMD.MS	A module containing the overlaid command processor routines for commands prefixed by ^K in the MicroStar command set.
OCMD.MS	A module containing the overlaid command processor routines for commands prefixed by ^O in the MicroStar command set.
QCMD.MS	A module containing the overlaid command processor routines for commands prefixed by ^Q in the MicroStar command set.
CMD.MS	A module containing the overlaid command processor routines for commands invoked by a single keystroke in MicroStar.
TASK.MS	A module containing the task scheduler and main command loop for the Toolbox editors.
INPUT.MS	A module containing the keyboard input routines for the Toolbox.
EDITERR.MSG	A file containing the error messages for the Toolbox editors. (This file is identical to the file with the same name on Disk #1.)
PRINT.MS	Print routines for MicroStar.
MSCMD.MS	MicroStar-specific commands.
SPELL.MS	MicroStar/Lightning spell checker.
PULLDOWN.MS	Pulldown menu routines.
K.MS	^K command dispatcher.
O.MS	^O command dispatcher.
Q.MS	^Q command dispatcher.
DISP.MS	MicroStar main command dispatcher.

Chapter 16
TURBO EDITOR TOOLBOX CONSTANTS

This section describes, in alphabetical order, the constants used in the Turbo Editor Toolbox routines.

Col1

Declaration `const Col1 = 1;`

Purpose Logical column number 1.

Colored

Declaration `const Colored = 4;`

Purpose Flag in Linedesc record indicating that the line should be displayed in Usercolor instead of Txtcolor or Blockcolor.

Ctrla-Ctrlz

Declaration `const Ctrla = 1; Ctrlb = 2; ...; Ctrlz = 26;`

Purpose ASCII codes of the control characters ^A-^Z.

Defhelplen

Declaration `const Defhelplen = 9;`

Purpose Default number of lines in help window.

Defnocols

Declaration `const Defnocols = 80;`

Purpose Number of columns on the physical screen.

Defnorows

Declaration `const Defnorows = 25;`

Purpose Number of rows on the physical screen.

Deftypahd

Declaration `const Deftypahd = 500;`

Purpose Default size of the typeahead buffer.

Del

Declaration `const Del = 127;`

Purpose ASCII code of the Delete key (Ctrl-Backspace).

Errorfile

Declaration `const Errorfile = 'EDITERR.MSG';`

Purpose Name of the Toolbox error message file.

Escape

Declaration `const Escape = 27;`

Purpose ASCII code of the escape key.

Inblock

Declaration `const Inblock = 1;`

Purpose Flag in Linedesc record indicating that the line is inside the block.

Lex1

Declaration `const Lexl = 1;`

Purpose Lexical level 1.

Line1

Declaration `const Linel = 1;`

Purpose Logical line number 1.

Maxlinelength

Declaration `const Maxlinelength = 255;`

Purpose Mamimum number of characters in a textline string.

Maxmarker

Declaration `const Maxmarker = 20;`

Purpose Maximum number of text markers.

Nofile

Declaration `const Nofile = 'NONAME';`

Purpose Filename for a window that has not had a file read into it yet.

Notavailable

Declaration `const Notavailable = 255;`

Purpose Value returned by input routines indicating no input was present.

Nul

Declaration `const Nul = 0;`

Purpose ASCII code of the null character.

Screenadr

Declaration `const Screenadr : integer = $B800;`

Purpose Physical memory address of the color screen.

Stai

Declaration `const Stai : string [2] = 'AI';`

Purpose Label for use on window status line indicating that Autoindent mode is active.

Stcol

Declaration `const Stcol : string [4] = 'Col:';`

Purpose Label for use on window status line indicating absolute cursor column in text stream.

Stfile

Declaration `const Stfile : string [5] = 'File:';`

Purpose Label for use on window status line indicating the current pathname.

Stins

Declaration `const Stins : string [3] = 'INS';`

Purpose Label for use on window status line indicating that Insert mode is active.

Stline

Declaration `const Stline : string [5] = 'Line:';`

Purpose Label for use on window status line indicating absolute cursor line in text stream.

Stte

Declaration `const Stte : string [2] = 'TE';`

Purpose Label for use on window status line indicating that tab expansion mode is active.

Stwindow

Declaration `const Stwindow : string [7] = 'Window:';`

Purpose Label for use on window status line indicating the window number.

Stww

Declaration `const Stww : string [2] = 'WW';`

Purpose Label for use on window status line indicating that Wordwrap mode is active.

Wrapped

Declaration `const Wrapped = 2;`

Purpose Flag in Linedesc record indicating that the line was created by wordwrapping.

Zero-Nine

Declaration `const Zero = 48; One  = 49; ...; Nine = 57;`

Purpose ASCII codes of the digits 0-9.

Chapter 17
TURBO EDITOR TOOLBOX DATA TYPES

This section describes, in alphabetical order, the data types used in the Turbo Editor Toolbox routines.

Character

Declaration

```
type Character = record
                   Ch : char;
                   Color : byte
                 end;
```

Purpose

Used to store a display character including color/attribute.

Insflag

Declaration

```
type Insflag = (Insert, Typeover);
```

Purpose

Indicator for Insert mode in window data structure.

Linedesc

Declaration

```
type Linedesc = record
                  Fwdlink  : Plinedesc;
                  Backlink : Plinedesc;
                  Txt      : Ptextline;
                  Flags    : integer;
                  Buflen   : integer
                end;
```

Purpose

A complete description of a line in a text stream, including pointers to its neighbors, the text on the line, flags indicating that it is inside a block, wordwrapped, etc., and the length of the text on the line.

Plinedesc

Declaration `type Plinedesc = ^ Linedesc;`

Purpose Pointer to a complete line descriptor including links to neighboring line descriptors.

Ptextline

Declaration `type Ptextline = ^ Textline;`

Purpose Pointer to the string of text on a line in a text stream.

Pwindesc

Declaration `type Pwindesc = ^ Windesc;`

Purpose Pointer to a window structure.

St6

Declaration `type St6 = string [6];`

Purpose String for use in input translation.

String80

Declaration `type String80 = string [80];`

Purpose String for use in file utilities.

Strvartype

Declaration `type Strvartype = string [Maxlinelength];`

Purpose String for use in user interface.

Textline

Declaration `type Textline = string [Maxlinelength];`

Purpose String used to hold text on a line in a text stream.

Varstring

Declaration `type Varstring = string [Defnocols];`

Purpose General-use variable length string.

Windesc

Declaration

```
type Windesc = record
                Fwdlink     : Pwindesc;
                Backlink    : Pwindesc;
                Filename    : Varstring;
                Insertflag  : Insflag;
                WW          : boolean;
                AI          : boolean;
                Firstlineno : integer;
                Lastlineno  : integer;
                Lmargin     : integer;
                Rmargin     : integer;
                Lineno      : integer;
                Colno       : integer;
                Clineno     : integer;
                Topline     : Plinedesc;
                Curline     : Plinedesc;
                Stream      : integer;
                Leftedge    : integer
              end;
```

Purpose A complete description of a window including pointers to its neighbors, its current filename, the current operation modes, where it is located on the screen, its margins, the absolute line number, pointers to the current line and top line, a unique stream identifier, and the leftmost displayed column.

Chapter 18
TURBO EDITOR TOOLBOX VARIABLES

This section describes, in alphabetical order, the global variables used in the Turbo Editor Toolbox routines.

Abortcmd

Declaration `var Abortcmd : boolean;`

Purpose Set to indicate that user has aborted with ^U.

Aborting

Declaration `var Aborting : boolean;`

Purpose Set when inside *EditAbort* to prevent recursion.

Asking

Declaration `var Asking : boolean;`

Purpose Set if *EditAskfor* is getting input from the command line.

Blockcolor

Declaration `var Blockcolor : integer;`

Purpose Color/attribute for text inside a block.

Blockfrom

Declaration `var Blockfrom : Plinedesc;`

Purpose Pointer to top line of the currently defined block.

Blockhide

Declaration `var Blockhide : boolean;`

Purpose Set if block is not displayed.

Blockto

Declaration `var Blockto : Plinedesc;`

Purpose Pointer to bottom line of the currently defined block.

Bordcolor

Declaration `var Bordcolor : integer;`

Purpose Color/attribute for window status lines.

Circbuf

Declaration `var Cirbuf : array [0..Deftypahd] of char;`

Purpose The circular typeahead buffer.

Circin

Declaration `var Circin : integer;`

Purpose Pointer into the typeahead buffer where *Pokechr* should insert characters.

Circout

Declaration `var Circout : integer;`

Purpose Pointer into the typeahead buffer where *EditPushtbf* should insert characters and where *EditGetinput* should remove them.

Cmdcol

Declaration `var Cmdcol : integer;`

Purpose Column at which the next command line operation should start.

Cmdcolor

Declaration `var Cmdcolor : integer;`

Purpose Color/attribute for command line.

Cmdlinest

Declaration `var Cmdlinest : Textline;`

Purpose Image of the command line for screen updating.

Curwin

Declaration `var Curwin : Pwindesc;`

Purpose Pointer to the current window's descriptor.

EditChangeflag

Declaration `var EditChangeFlag: boolean;`

Purpose Set if editor should exit.

EditUsercommandInput

Declaration `var EditUsercommandInput : integer;`

Purpose Count of characters pushed into the typeahead buffer by *UserCommand.*

Interactive

Declaration `var Interactive : boolean;`

Purpose Set if *EditAskfor* is getting input directly from the keyboard instead of the typeahead buffer.

Intrflag

Declaration `var Intrflag : (Nointrpt, Intrpt);`

Purpose Nointrpt indicates that the routine is not to be aborted if input is discovered in process. Intrpt indicates that the routine may abort if input is encountered.

Linelength

Declaration `var Linelength : integer;`

Purpose Number of characters in a text line.

Logscrcols

Declaration `var Logscrcols : integer;`

Purpose Number of columns in a logical line.

Logscrrows

Declaration `var Logscrrows : integer;`

Purpose Number of lines on the logical screen.

Logtopscr

Declaration `var Logtopscr : integer;`

Purpose Physical line number for logical line 1.

Marker

Declaration `var Marker : array [1..Maxmarker] of Plinedesc;`

Purpose Array of text markers.

Nextstream

Declaration `var Nextstream : integer;`

Purpose Next stream identifier to be assigned to a stream.

Notfound

Declaration `var Notfound : boolean;`

Purpose Set if pattern not found by search or search/replace.

Optstr

Declaration `var Optstr : Varstring;`

Purpose String of find/replace options.

Physcrcols

Declaration `var Physcrcols : integer;`

Purpose Number of columns in a physical display line.

Physcrrows

Declaration `var Physcrrows : integer;`

Purpose Number of lines on the physical screen.

Physcrsig

Declaration
```
var Physcrsig : integer;
```

Purpose
Physical screen line for logical command line.

Replacestr

Declaration
```
var Replacestr : Varstring;
```

Purpose
Previous string used to replace search pattern.

Retracemode

Declaration
```
var Retracemode : boolean;
```

Purpose
Set if waiting for vertical retrace is necessary in screen updating.

Rundown

Declaration
```
var Rundown : boolean;
```

Purpose
Set if editor should exit.

Screen

Declaration
```
var Screen : array [1..Defnorows, 1..Defnocols]
                          of Character;
```

Purpose
Memory image of the screen used to determine whether to update the physical screen.

Searchstr

Declaration
```
var Searchstr : Varstring;
```

Purpose
Previous pattern searched for.

Tabsize

Declaration `var Tabsize : integer;`

Purpose Number of columns between tab stops.

Txtcolor

Declaration `var Txtcolor : integer;`

Purpose Color/attribute for normal text.

Typbufovl

Declaration `var Typbufovl : boolean;`

Purpose Set if typeahead buffer has overflowed.

Undocount

Declaration `var Undocount : integer;`

Purpose Number of lines currently on the UNDO stack.

Undoend

Declaration `var Undoend : Plinedesc;`

Purpose Pointer to end of Undo stack where very old lines are discarded.

Undolimit

Declaration `var Undolimit : integer;`

Purpose Maximum number of lines the Undo stack can hold.

Undostack

Declaration `var Undostack : Plinedesc;`

Purpose Pointer to the top of the Undo stack.

Updcurflag

Declaration `var Updcurflag : boolean;`

Purpose Set if the cursor needs to be updated.

Usercolor

Declaration `var Usercolor : integer;`

Purpose Auxiliary color for applications-like menus.

Window1

Declaration `var Window1 : Pwindesc;`

Purpose Pointer to the window displayed at the top of the screen.

Winstack

Declaration `var Winstack : Pwindesc;`

Purpose Pointer to the top of the list of free window structures.

Chapter 19
TURBO EDITOR TOOLBOX PROCEDURES AND FUNCTIONS

This section describes, in alphabetical order, the Turbo Editor Toolbox procedures and functions. The call-up for each procedure or function is given, followed by a detailed description of its function. Remarks and restrictions are given where appropriate, as well as cross-referencing to related procedures and functions. The Turbo Editor file that contains the procedure or function is given in brackets next to the name of the procedure or function.

Advance [CMD.ED]

Declaration	`procedure Advance;`
Usage	Advance;
Parameters	None
Function	This routine is local to *EditRightWord*, and is used to move the cursor rightward one character, moving to the beginning of the following line if necessary.
Restrictions	None

EditAbort [INPUT.ED]

Declaration `procedure EditAbort;`

Usage EditAbort;

Parameters None

Function This routine aborts the typeahead buffer. It sets the global variable Abortcmd, which should be checked by any procedure that does input. It then calls *Edit-Errormsg* to display a message on the command line and clear the typeahead buffer.

Remarks In the default editing system, this is the only command processor that is called from *Pokechr* instead of a menu processor. This ensures that it will be immediate and not buffered behind other commands.

Restrictions None

See Also EditErrormsg
Pokechr

EditAppchar [USER.ED]

Declaration `procedure EditAppchar (var s : Varstring; Ch : byte);`

Usage EditAppchar (s, Ch);

Parameters s : variable length string
Ch : character to add

Function This routine is used internally to append the specified character to the string, and return the result in that string.

Restrictions Length (s) must be less than Defnocols, otherwise an overflow will occur.

EditAppcmdnam [USER.ED]

Declaration `procedure EditAppcmdnam (s : Varstring);`

Usage EditAppcmdnam (s);

Parameters s : string to display

Function This routine displays the specified string on the command line, to the right of any other text present. It also updates Cmdcol so that *EditAskfor* will position the cursor correctly for input, and calls *EditCuradr* to position the cursor there.

Restrictions If the string given is longer than the space left on the command line, it will overwrite part of the top window's status line; thus, this must either be checked or avoided.

See Also EditAskfor
EditCuradr
EditZapcmdnam

EditAskfor [USER.ED]

Declaration	`procedure EditAskfor (var s : Varstring);`
Usage	EditAskfor (s);
Parameters	s : string to place user's input in
Function	This routine positions the cursor on the command line in Cmdcol, and gets input into the specified string until a carriage return is typed.
Remarks	If the global variable Interactive is set to FALSE, input is taken from the typeahead buffer; otherwise, input is taken directly from the keyboard without affecting the typeahead buffer. If control characters are typed, they appear as a caret followed by the representative character; i.e., Ctrl-A appears as ^A. Input is not accepted when there is no room to display it on the command line. The return is always a string; to get a number, use this routine followed by *EditCvts2i*. If a carriage return is typed immediately, a string of zero length is returned. Any previous contents of the string are lost unless Ctrl-U is typed during the input.
Restrictions	None
See Also	EditCvts2i EditZapcmdnam

EditBackground [TASK.ED]

Declaration	`procedure EditBackground;`
Usage	EditBackground;
Parameters	None
Function	This routine performs the real-time functions that must be executed whenever input is pending and the editor cannot execute a command.
Restrictions	None
See Also	EditSchedule EditSystem

EditBeginningEndLine [CMD.ED]

Declaration `procedure EditBeginningEndLine;`

Usage EditBeginningEndLine;

Parameters None

Function This routine jumps between the beginning and the end of the current line in the current window.

Remarks If the cursor is not in column one, it is moved there; if the cursor is in column one, it is positioned immediately after the last non-blank character on the current line.

Restrictions None

EditBeginningLine [QCMD.ED]

Declaration	`procedure EditBeginningLine;`
Usage	EditBeginningLine;
Parameters	None
Function	This routine positions the cursor to column 1 of the current line.
Restrictions	None
See Also	EditEndLine EditBeginningEndLine

EditBlockBegin [KCMD.ED]

Declaration `procedure EditBlockBegin;`

Usage EditBlockBegin;

Parameters None

Function This routine processes the begin block command. It repoints the Blockfrom pointer in the current window to the line on which the cursor is positioned (Curline).

Restrictions None

See Also EditBlockEnd
EditBlockHide

EditBlockCopy [KCMD.ED]

Declaration	`procedure EditBlockCopy;`
Usage	EditBlockCopy;
Parameters	None
Function	This routine processes the copy block command. It copies the text lines in the range of the block defined by the pointers Blockfrom and Blockto for the current window into the place where the cursor is positioned.
Restrictions	None
See Also	EditBlockMove EditBlockDelete

EditBlockDelete [KCMD.ED]

Declaration	`procedure EditBlockDelete;`
Usage	EditBlockDelete;
Parameters	None
Function	This routine processes the delete block command. It deletes the text lines in the range of the block defined by the pointers Blockfrom and Blockto for the current window.
Remarks	All deletes should be done with *Delline*, so that multiple windows pointing to the same text stream are handled. *Delline* also handles preparation for Undo.
Restrictions	None
See Also	EditBlockMove EditBlockCopy

EditBlockEnd [KCMD.ED]

Declaration	`procedure EditBlockEnd;`
Usage	EditBlockEnd;
Parameters	None
Function	This routine processes the begin block command. It repoints the Blockto pointer in the current window to the line on which the cursor is positioned (Curline).
Restrictions	None
See Also	EditBlockBegin EditBlockHide

EditBlockHide [KCMD.ED]

Declaration	`procedure EditBlockHide;`
Usage	EditBlockHide;
Parameters	None
Function	This routine processes the hide/display block toggle command. If *BlockHide* = false upon entry, then it is set to true. If otherwise, it is set to false only if both Blockfrom and Blockto are defined to be <> nil and they span the same text stream.
Restrictions	None
See Also	EditBlockBegin EditBlockEnd

EditBlockMove [KCMD.ED]

Declaration	`procedure EditBlockMove;`
Usage	EditBlockMove;
Parameters	None
Function	This routine processes the move block command. It moves the text lines in the range of the block defined by the pointers Blockfrom and Blockto into the place where the cursor is positioned. Before the move, it makes sure the cursor is not inside the block.
Restrictions	None
See Also	EditBlockCopy EditBlockDelete

EditBottomBlock [QCMD.ED]

Declaration	`procedure EditBottomBlock;`
Usage	EditBottomBlock;
Parameters	None
Function	This routine moves the cursor in the current window to the last line in the block.
Remarks	If the block is present in a different window, the cursor position in the current window is saved, and the cursor is moved to the window in which the block occurs. If the block is not defined, an error message is displayed.
Restrictions	None
See Also	EditTopBlock

EditBreathe [INPUT.ED]

Declaration	`procedure EditBreathe;`
Usage	EditBreathe;
Parameters	None
Function	This routine is called by an editor procedure to accept characters typed at the keyboard while some other operation is being performed, such as file I/O.
Remarks	*EditBreathe* calls *EditKeypressed,* which will check the keyboard and fill the editor's typeahead buffer, thus ensuring that the BIOS buffer will not overflow.
Restrictions	None
See Also	EditKeypressed

EditCenterLine [OCMD.ED]

Declaration	`procedure EditCenterLine;`
Usage	EditCenterLine;
Parameters	None
Function	This routine centers the text on the current line in the current window.
Remarks	If Wordwrap mode is set for the current window, the text is centered between the currently defined margins; otherwise it is centered between column 1 and the right margin. If the operation causes the text to be pushed beyond the current line's buffer length, the procedure will attempt to allocate more memory. If enough memory cannot be allocated, an error message is displayed, and the operation is aborted. The cursor position remains unchanged.
Restrictions	None
See Also	EditCpgotoln EditGotoLine

EditChangeCase [OCMD.ED]

Declaration	`procedure EditChangeCase;`
Usage	EditChangeCase;
Parameters	None
Function	This routine changes the case of the character at the current cursor position in the current window.
Remarks	If that character is uppercase, it is changed to lowercase; if it is lowercase, it is changed to uppercase. If the character is not alphabetic, no action is performed. The cursor position remains unchanged.
Restrictions	None

EditClsinp [TASK.ED]

Declaration	`procedure EditClsinp;`
Usage	EditClsinp;
Parameters	None
Function	This routine reads the next character from the typeahead buffer. If it was pushed by UserCommand, it will be processed only by the Toolbox routine *EditPrccmd*. If it wasn't pushed by UserCommand, then UserCommand gets a chance to process it before any other Toolbox routine. If UserCommand doesn't want the Toolbox command processors to see the input it saw, it sets its argument to a value of 255 before returning.
Restrictions	None
See Also	EditPrccmd EditPrctxt

EditColorFile [USER.ED]

Declaration	`procedure EditColorFile;`
Usage	EditColorFile;
Parameters	None
Function	This routine sets the Colored flag in every line in the current window's text stream.
Remarks	This function causes the screen updating routines to display the text in Usercolor instead of Blockcolor or Txtcolor. If any lines in the text are already colored, they will remain so.
Restrictions	None
See Also	EditColorLine

EditColorLine [USER.ED]

Declaration	`procedure EditColorLine;`
Usage	EditColorLine;
Parameters	None
Function	This routine sets the Colored flag for the current line in the current window.
Remarks	This function causes the screen updating routines to display the line in Usercolor rather than Blockcolor or Txtcolor. If the flag is already set for the current line, it is unchanged.
Restrictions	None
See Also	EditColorFile

EditCompressLine [CMD.ED]

Declaration	`procedure EditCompressLine (Lp : Plinedesc);`
Usage	EditCompressLine (Lp);
Parameters	Lp : pointer to line to compress
Function	This routine is local to *EditReformat*, and is used to compress multiple spaces to single spaces on the line referenced. It thus makes the line-splicing algorithms clearer.
Restrictions	None
See Also	EditReformat

EditCpcrewin [O.ED]

Declaration `procedure EditCpcrewin;`

Usage EditCpcrewin;

Parameters None

Function This routine creates a new window on the screen. It calls *EditAskfor* to get two strings from the user, *EditCvts2i* to convert them to a size and a window number to compress, and then *EditWindowCreate* to perform the operation. That routine handles all error conditions.

Restrictions None

See Also EditAskfor
EditCvts2i
EditWindowCreate

EditCpdelwin [O.ED]

Declaration `procedure EditCpdelwin;`

Usage EditCpdelwin;

Parameters None

Function This routine deletes a window from the screen. It calls *EditAskfor* to get a string from the user, *EditCvts2i* to convert it to a window number to delete, and then *EditWindowDelete* to perform the operation. That routine handles all error conditions.

Restrictions None

See Also EditAskfor
EditCvts2i
EditWindowDelete

EditCpexit [K.ED]

Declaration `procedure EditCpexit;`

Usage EditCpexit;

Parameters None

Function This routine is used to query the user before exiting the editor.

Remarks *EditCpexit* calls *EditAskfor* to make sure the user wants to exit the editor. If the string returned is "YES" (ignoring case), then *EditExit* is called to exit the editor. If any other string is specified, no action is performed.

Restrictions This routine does not save any window files before exiting (hence the query). This must either be done manually by the user or automatically by the calling routine.

See Also EditAskfor
EditExit

EditCpFileSave [K.ED]

Declaration	`procedure EditCpFileSave;`
Usage	EditCpFileSave;
Parameters	None
Function	This routine gets the filename from the command line and transfers the name to the *EditFileWrite* procedure
Restrictions	None
See Also	EditFileWrite

EditCpFind [Q.ED]

Declaration	`procedure EditCpFind;`
Usage	EditCpFind;
Parameters	None
Function	This routine asks for the find parameter and calls *EditFind.*
Restrictions	None
See Also	EditFind

EditCpgotocl [O.ED]

Declaration `procedure EditCpgotocl;`

Usage EditCpgotocl;

Parameters None

Function This routine moves the cursor to a column number.

Remarks *EditCpgotocl* calls *EditAskfor* to get a string from the user, *EditCvts2i* to convert it to a number, and then *EditGotoColumn* to perform the operation. *EditGotoColumn* does all error checking.

Restrictions None

See Also EditAskfor
EditCpgotoln
EditCpjmpmrk
EditCvts2i
EditGotoColumn

EditCpgotoln [O.ED]

Declaration `procedure EditCpgotoln;`

Usage EditCpgotoln;

Parameters None

Function This routine moves the cursor to a specific line in the current window.

Remarks *EditCpgotoln* calls *EditAskfor* to get a string from the user, *EditCvts2i* to convert it to a number, and then *EditGotoLine* to perform the operation. *EditGotoLine* performs all error checking.

Restrictions None

See Also EditAskfor
EditCpgotocl
EditCpjmpmrk
EditCvts2i
EditGotoLine

EditCpgotowin [O.ED]

Declaration `procedure EditCpgotowin;`

Usage EditCpgotowin;

Parameters None

Function This routine moves the cursor to a specific window. It calls *EditAskfor* to get string from the user, *EditCvts2i* to convert it to a destination window number, and then *EditWindowGoto* to perform the operation. *EditWindowGoto* handles all error conditions.

Restrictions None

See Also EditAskfor
EditCvts2i
EditWindowGoto

EditCpjmpmrk [Q.ED]

Declaration	`procedure EditCpjmpmrk;`
Usage	EditCpjmpmrk;
Parameters	None
Function	This routine moves the cursor to a text marker.
Remarks	*EditCpjmpmrk* calls *EditAskfor* to get a string from the user, *EditCvts2i* to convert it to a number, and then calls *EditJumpMarker* to perform the operation. *EditJumpMarker* does all error checking.
Restrictions	None
See Also	EditAskfor EditCpgotocl EditCpgotoln EditCvts2i EditJumpMarker

EditCplnkwin [O.ED]

Declaration	`procedure EditCplnkwin;`
Usage	EditCplnkwin;
Parameters	None
Function	This routine links two windows together. It calls *EditAskfor* to get two strings from the user, *EditCvts2i* to convert them to a destination window number and a source window number, and then *EditWindowLink* to perform the operation.
Remarks	*EditWindowLink* handles all error conditions.
Restrictions	None
See Also	EditAskfor EditCvts2i EditWindowLink

EditCpReplace [Q.ED]

Declaration	`procedure EditCpReplace;`
Usage	EditCpReplace;
Parameters	None
Function	This routine asks for the replace parameters and calls *EditReplace*.
Restrictions	None
See Also	EditReplace

EditCprfw [K.ED]

Declaration `procedure EditCprfw;`

Usage EditCprfw;

Parameters None

Function This routine queries the user for a filename to read into the current window. It then calls *EditFileRead* to perform the operation.

Restrictions The filename specified will be overwritten if it already exists; the calling procedure or the user should check before calling to make sure that valuable data will not be destroyed.

See Also EditFileRead

EditCpsetlm [O.ED]

Declaration

```
procedure EditCpsetlm;
```

Usage

EditCpsetlm;

Parameters

None

Function

This routine sets the left margin for the current window.

Remarks

EditCpsetlm calls *EditAskfor* to get a string from the user, *EditCvts2i* to convert it to a number, and then *EditSetLeftMargin* to perform the operation. *EditSetLeftMargin* performs all error checking.

Restrictions

None

See Also

EditAskfor
EditCpsetrm
EditCvts2i
EditSetLeftMargin

EditCpsetmrk [K.ED]

Declaration	`procedure EditCpsetmrk;`
Usage	EditCpsetmrk;
Parameters	None
Function	This routine sets a text marker to the current line in the current window.
Remarks	*EditCpsetmrk* calls *EditAskfor* to get a string from the user, *EditCvts2i* to convert it to a number, and then *EditSetMarker* to perform the operation. *EditSetMarker* performs all error checking.
Restrictions	None
See Also	EditAskfor EditCvts2i EditJumpMarker EditSetMarker

EditCpsetrm [O.ED]

Declaration `procedure EditCpsetrm;`

Usage EditCpsetrm;

Parameters None

Function This routine sets the right margin for the current window.

Remarks *EditCpsetrm* calls *EditAskfor* to get a string from the user, *EditCvts2i* to convert it to an integer, and then *EditSetRightMargin* to perform the operation. *EditSetRightMargin* performs all error checking.

Restrictions None

See Also EditAskfor
EditCpsetlm
EditCvts2i
EditSetRightMargin

EditCptabdef [K.ED]

Declaration `procedure EditCptabdef;`

Usage EditCptabdef;

Parameters None

Function This routine asks the user for a tab width to set.

Remarks *EditCptabdef* calls *EditAskfor* to get a string, then *EditCvts2i* to convert it to an integer. If the string returned is NULL, then *EditDefineTab* is called with a parameter one less than the current column number, so that a tab from column one would land directly in that column. If a number is given, *EditDefineTab* is called with that number. *EditDefineTab* checks for overflow and underflow.

Restrictions None

See Also EditAskfor
EditCvts2i
EditDefineTab

EditCpundlim [O.ED]

Declaration

```
procedure EditCpundlim;
```

Usage

EditCpundlim;

Function

This routine sets the *Undo* stack limit.

Remarks

EditCpundlim calls *EditAskfor* to get a string from the user, *EditCvts2i* to convert it to a number, and then *EditSetUndoLimit* to perform the operation. *EditSetUndoLimit* performs all error checking.

Restrictions

None

See Also

EditAskfor
EditCvts2i
EditSetUndoLimit

EditCpwfw [K.ED]

Declaration	`procedure EditCpwfw;`
Usage	EditCpwfw;
Parameters	None
Function	This routine queries the user for a filename to write the current window's text to. It then calls *EditFileWrite* to perform the operation.
Restrictions	Note that the filename specified will be overwritten if it already exists; the calling procedure or the user should check before calling to make sure that valuable data will not be destroyed.
See Also	EditFileWrite

EditCvts2i [USER.ED]

Declaration

```
procedure EditCvts2i (Src : Varstring;
                      var Result : integer);
```

Usage

EditCvts2i (Src, Result);

Parameters

Src : string to convert Result : variable to return the value in

Function

This routine converts the character string in *Src* to an integer using the Turbo procedure *Val*.

Remarks

If *Val* returns a non-zero error code, an error message is displayed indicating an error in the string; otherwise, the integer translation is returned in Result.

Restrictions

None

EditDecline [USER.ED]

Declaration	`procedure EditDecline;`
Usage	EditDecline;
Parameters	None
Function	This routine decrements the absolute line number for the current window and calls *EditUpdwinsl* to display it.
Remarks	*EditDecline* provides fast scrolling even in very large files.
Restrictions	None
See Also	EditUpdwinsl

EditDefineTab [KCMD.ED]

Declaration	`procedure EditDefineTab (Size : integer);`
Usage	EditDefineTab (Size);
Parameters	Size : distance in columns between tab stops.
Function	This routine sets the global variable Tabsize to the parameter passed. This variable is used by *EditTab* to determine where to place the cursor.
Remarks	If the number passed is less than 1 or greater than Maxint, no action is performed.
Restrictions	None
See Also	EditTab

EditDeleteLeftChar [CMD.ED]

Declaration	`procedure EditDeleteLeftChar;`
Usage	EditDeleteLeftChar;
Parameters	None
Function	This routine deletes the character to the left of the cursor in the current window. The cursor is moved left one column, and all text to the right of the cursor is shifted left one column.
Remarks	If the cursor is in column one, the current line is joined to the previous line, if that line exists. If the cursor is in column one on the top line of the text stream, no action is performed.
Restrictions	None
See Also	EditDeleteRightChar

EditDeleteLine [CMD.ED]

Declaration `procedure EditDeleteLine;`

Usage EditDeleteLine;

Parameters None

Function This routine deletes the current line in the current window from its text stream.

Remarks If there is only one line of text in the window, it is filled with spaces but not deleted. If the Undo facility is enabled, the line will be replicated and placed on the Undo stack. If the current line is marked by a marker, that marker will become undefined. If the current line is a block boundary, that block pointer will become undefined, and the block highlighting will be turned off.

Restrictions None

See Also EditDeleteLeftChar
EditDeleteRightChar
EditDeleteRightWord

EditDeleteRightChar [CMD.ED]

Declaration	`procedure EditDeleteRightChar;`
Usage	EditDeleteRightChar;
Parameters	None
Function	This routine deletes the character at the current cursor position in the current window. The text to the right of the cursor is shifted left one column.
Remarks	If the cursor is beyond the last non-blank character on the current line, the line below will be joined to it, if that line exists and memory can be allocated to hold the joined line.
Restrictions	None
See Also	EditDeleteLeftChar

EditDeleteRightWord [CMD.ED]

Declaration	`procedure EditDeleteRightWord;`
Usage	EditDeleteRightWord;
Parameters	None
Function	This routine deletes a word from the current line in the current window, starting at the cursor position. A word is defined as all characters in the same class, followed by spaces, if any. The three classes are alphanumeric characters, punctuation characters, and spaces.
Remarks	If the cursor is beyond the last non-blank character on the current line, the line below is joined to the current one. If that line does not exist, no action is performed.
Restrictions	None
See Also	EditDeleteLeftChar EditDeleteLeftWord EditDeleteLine EditDeleteRightChar

EditDeleteTextRight [QCMD.ED]

Declaration	`procedure EditDeleteTextRight;`
Usage	EditDeleteTextRight;
Parameters	None
Function	This routine deletes the character at the current cursor position, and all text to the right of it on the same line.
Restrictions	None

EditDelline [USER.ED]

Declaration `procedure EditDelline (p : Plinedesc);`

Usage EditDelline (p);

Parameters p : pointer to line descriptor to delete

Function This routine deletes a line descriptor from a text stream. The text stream is not spliced around the current line; this must be done manually before calling *EditDelline*. The routine checks to make sure that the line does not point to any window's Topline or Curline, Blockfrom, Blockto, or a text marker.

Remarks If the line is a block limit or a marker, the appropriate pointer will be reset to nil. If the line points to a window's Curline, the Curline pointer for that window will be moved to another line. If the line points to any window's Topline, the window will be scrolled to move Topline. If deleting the line shortens the window's visible span, *EditRealign* is called to fix up the display and the other windows. Finally the line is replicated for Undo, and *EditDestxtdes* is called to free the line.

Restrictions If the line in question is in a text stream, the *Fwdlink* and *Backlink* pointers of its neighbors must be spliced around the line before calling this routine.

See Also EditDestxtdes
EditRealign

EditDestxtdes [USER.ED]

Declaration	`procedure EditDestxtdes (Desc : Plinedesc);`
Usage	EditDestxtdes (Desc);
Parameters	Desc : pointer to line descriptor to release
Function	This routine is passed a pointer to a line descriptor. It first frees the pointer to the text line, then the descriptor itself. The line is not replicated for Undo.
Remarks	If this routine is passed a line that has not been removed from a text stream, unpredictable results can occur. The calling routine must check to make sure that it is not deleting the Topline or Curline of any window, a text marker, a block limit, the current line, etc. The text stream pointers must also be spliced around the line to be deleted. *EditDelline* makes all necessary checks and replicates the line for Undo before calling this routine.
Restrictions	The line specified must be completely removed from the text stream it was a part of, if any. The text stream must then be spliced around the line. The line must not point to the Topline or Curline of any window; it must not be marked as a block limit or a text marker.
See Also	EditDelline

EditDownLine [CMD.ED]

Declaration	`procedure EditDownLine;`
Usage	EditDownLine;
Parameters	None
Function	This routine moves the cursor in the current window down one line.
Remarks	If the cursor is on the last displayed line in the window, the window is scrolled down to keep the cursor within the window. If the cursor is on the last line in the text stream, no action is performed.
Restrictions	None
See Also	EditUpLine

EditDownPage [CMD.ED]

Declaration	`procedure EditDownPage;`
Usage	EditDownPage;
Parameters	None
Function	This routine slides the current window down one page over its text stream. The number of lines scrolled will be one less than the number of lines displayed in the window.
Remarks	If the last line of the text stream is displayed on the top line of the window, no action is performed.
Restrictions	None
See Also	EditUpPage

EditEndLine [QCMD.ED]

Declaration	`procedure EditEndLine;`
Usage	EditEndLine;
Parameters	None
Function	This routine positions the cursor to the right of the last nonblank character in the current line.
Restrictions	None
See Also	EditBeginningLine EditBeginningEndLine

EditErrormsg [USER.ED]

Declaration `procedure EditErrormsg (Msgno : byte);`

Usage EditErrormsg (Msgno);

Parameters Msgno : code of message to display

Function This routine is called when an error is encountered in the editor. The parameter passed is a code for the error message to display; for instance, code 1 indicates an out of memory error. It clears the typeahead buffer and waits for a key to be struck before clearing the command line. Before any message is displayed, *UserError* is called with the same message code.

Remarks If this routine resets the code to zero, *EditErrormsg* returns; otherwise, it continues and displays the message. If the error code is out of range, no message is displayed but the routine still clears the typeahead buffer and waits for input.

Restrictions None

See Also EditFileerror

EditExit [KCMD.ED]

Declaration	`procedure EditExit;`
Usage	EditExit;
Parameters	None
Function	This routine terminates the editor by setting the global variable Rundown to TRUE.
Restrictions	Note that this procedure does not save files; they must be saved manually or by the calling procedure.
See Also	EditAskfor EditCpExit

EditFileRead [KCMD.ED]

Declaration	`procedure EditFileRead (Fname : Varstring);`
Usage	EditFileRead (Fname);
Parameters	Fname : complete pathname of file to read.
Function	This routine appends the string "[Wait]" to the command line, updates the screen, and then calls *EditReatxtfil* to read the file named by Fname into the current window. *EditReatxtfil* checks for out of memory errors, file errors, and other problems.
Restrictions	None
See Also	EditReatxtfil

EditFileWrite [KCMD.ED]

Declaration	`procedure EditFileWrite (Fname : Varstring);`
Usage	EditFileWrite (Fname);
Parameters	Fname : complete pathname of file to write.
Function	This routine writes all text in the current window to the file named by *Fname*. If the file does not exist, it is created; if it does exist, it is overwritten. This routine checks for file errors (invalid name, etc.). Lines that have the Wrapped attribute are indicated by a carriage return with the high bit set (ASCII 141).
Restrictions	Note that the filename specified will be overwritten if it already exists; the calling procedure or the user should check before calling to make sure that valuable data will not be destroyed.

EditFind [QCMD.ED]

Declaration `procedure EditFind;`

Usage EditFind;

Parameters None

Function This routine uses *EditAskfor* to get a string to search for from the typeahead buffer.

Remarks If the string obtained is null, it will be assigned the previous string searched for. It then calls *EditScanpat* to search for the pattern on each line of the file, starting at the current cursor position. If the pattern is not found, an error message is displayed.

Restrictions None

See Also EditAskfor
EditScanpat

EditGenlineno [SCREEN.ED]

Declaration	`procedure EditGenlineno;`
Usage	EditGenlineno;
Parameters	None
Function	This routine generates the absolute line number for each window displayed by starting at the current line and stepping backward until the top of the stream is reached.
Remarks	If input is encountered at any time during this process, the routine is aborted because it is time consuming. The line number is not displayed on the status line by this routine, only calculated.
Restrictions	None

EditGotoColumn [OCMD.ED]

Declaration	`procedure EditGotoColumn (Cno : integer);`
Usage	EditGotoColumn (Cno);
Parameters	Cno : column number to go to.
Function	This routine positions the cursor to the specified column number.
Remarks	If the number passed is less than 1 or greater than Maxint, no operation is performed.
Restrictions	None
See Also	EditCpgotocl

EditGotoLine [OCMD.ED]

Declaration `procedure EditGotoLine (Lno : integer);`

Usage EditGotoLine (Lno);

Parameters Lno : line number to go to.

Function This routine positions the cursor to the line specified.

Remarks If it is less than 1, no operation is performed. If it is greater than the number of lines in the text stream, the cursor is placed on the last line in the text stream. The column position remains unchanged.

Restrictions None

See Also EditCenterline
EditCpgotoln
EditCpsetmrk

EditHscroll [SCREEN.ED]

Declaration	`procedure EditHscroll;`
Usage	EditHscroll;
Parameters	None
Function	This routine horizontally scrolls all windows if necessary to keep their cursors within the confines of the windows at all times.
Remarks	This is done by testing the current column number, and if it is out of bounds, the Leftedge field for that window is reset to bring the cursor within the window. If input is encountered at any time the routine is aborted.
Restrictions	None

EditIncline [USER.ED]

Declaration `procedure EditIncline;`

Usage EditIncline;

Parameters None

Function This routine increments the absolute line number for the current window and calls *EditUpdwinsl* to display it.

Remarks *EditIncline* provides fast scrolling even in very large files.

Restrictions None

See Also EditUpdwinsl

EditInitialize [LISTPROCS.ED]

Declaration `procedure EditInitialize;`

Usage EditInitialize;

Parameters None

Function This routine is called to initialize the entire editing environment. It sets the default display colors, tab size, search and replace strings, undo stack size, and block pointers. It also clears the screen and initializes the screen array. It then uses BIOS interrupt 10H to determine the screen type. It finally creates two windows on the screen and initalizes the text markers.

Restrictions This routine should always be the first routine called in any editor. Modifications to the initial setup may be made after this routine is called.

EditInsertCtrlChar [CMD.ED]

Declaration	`procedure EditInsertCtrlChar;`
Usage	EditInsertCtrlChar;
Parameters	None
Function	This routine accepts a single character from the typeahead buffer, and translates it to a control character. It then calls *EditPrctxt* with the ASCII value of this character to insert it into the text.
Restrictions	None
See Also	EditInsertLine EditPrctxt

EditInsertLine [CMD.ED]

Declaration	`procedure EditInsertLine;`
Usage	EditInsertLine;
Parameters	None
Function	This routine inserts a line at the current cursor position.
Remarks	If the cursor is in column one, a blank line is inserted above the current line. If the cursor is beyond the last non-blank character on the current line, a line is inserted below the current line. Otherwise, the current line will be split, and all text to the right of the cursor will be moved to the new line below the current one. If insufficient memory exists to create the new line, an error message is displayed.
Restrictions	None
See Also	EditInsertCtrlChar

EditJoinline [USER.ED]

Declaration `procedure ditJoinline;`

Usage EditJoinline;

Parameters None

Function This routine joins the line below the current line in the current window to the current line. The cursor must be positioned at the location where the line below should be appended.

Remarks If the resultant line is longer than the current line's buffer length, *EditSizeline* will be called to allocate a new text line. If *EditSizeline* reports insufficient memory, an error message is displayed and the text is not affected. Otherwise, the line below is appended and its line descriptor freed (and duplicated for Undo).

Restrictions The cursor must be positioned where the first character of the line below should be placed.

See Also EditSizeline

EditJumpMarker [QCMD.ED]

Declaration	`procedure EditJumpMarker (m : byte);`
Usage	EditJumpMarker (m : byte);
Parameters	m : marker number to jump t.
Function	This routine jumps to a text marker.
Remarks	If the number passed is less than 1 or greater than Maxmarker, an error message is displayed indicating the operation is not possible. Otherwise, the cursor is positioned to the marker. If the specified marker is in a different window, the current window's cursor position is saved and the cursor is moved to that window. The window's column number is not changed.
Restrictions	None
See Also	EditCpjmpmrk EditCpsetmrk EditSetMarker

EditK [K.ED]

Declaration `procedure EditK;`

Usage EditK;

Parameters None

Function This routine is the default utility menu processor for the editor. It displays the message "<Utility>" on the command line, waits for input, and then calls the appropriate toolbox routine to process the command.

Remarks If a different mapping of comamnds is desired, a new version of this procedure may be created using the default as a guide.

Restrictions None

See Also EditO
EditQ

EditLeftChar [CMD.ED]

Declaration	`procedure EditLeftChar;`
Usage	EditLeftChar;
Parameters	None
Function	This routine moves the cursor in the current window left one character.
Remarks	If the cursor is in column one, it is moved to the column immediately following the last non-blank character on the previous line. If the cursor is in column one on the first line in the text stream, no action is performed.
Restrictions	None
See Also	EditLeftWord

EditLeftWord [CMD.ED]

Declaration	`procedure EditLeftWord;`
Usage	EditLeftWord;
Parameters	None
Function	This routine moves the cursor in the current window left one word.
Remarks	If the cursor was positioned on or to the left of the first non-blank character on the current line, it is moved to the first column of the current line. If the cursor was positioned in the first column of the current line, it is moved to the column immediately following the last non-blank character on the previous line. If the cursor was positioned at the first character in the text stream, no action is performed. Otherwise, the cursor is positioned to the beginning of the previous word on the same line.
Restrictions	None
See Also	EditRightWord

EditLongLine [CMD.ED]

Declaration	`procedure EditLongline;`
Usage	EditLongline;
Parameters	None
Function	This routine is local to EditReformat, and is used to remove all text on the current line which extends beyond the current right margin.
Remarks	If there is a line below the current lne, the text is inserted at the beginning of that line. Otherwise, a new line is created below the current one, and the text is placed there. If the text extending beyond the right margin is a single word that also extends to the left of the left margin, an error message will indicate that there is a word too long.
Restrictions	None
See Also	EditNewLine EditReformat EditShortLine

EditMarkblock [SCREEN.ED]

Declaration	`procedure EditMarkblock;`
Usage	EditMarkblock;
Parameters	None
Function	This routine marks the currently defined block by stepping from the Blockfrom pointer to the Blockto pointer and setting the Inblock flag for each line in that range.
Remarks	If input is encountered at any time while marking, the operation is aborted. If the block highlighting is turned off, or if either limit pointer is undefined, no action is performed.
Restrictions	None

EditNewLine [CMD.ED]

Declaration	`procedure EditNewLine;`
Usage	EditNewLine;
Parameters	None
Function	This routine inserts a new line into the current window's text stream if Insert mode is set for the current window. Line splitting is identical to *EditInsertLine,* except that the cursor is moved with the text split.
Remarks	If Insert mode is not set, the cursor is simply moved down one line, unless the cursor is on the last line of the text stream, in which case a new line is created. In this case, the current line is not split, but the cursor is positioned on the new line. If Autoindent mode is set for the current window, the cursor is positioned under the first non-blank character on the previously current line. If that line is blank, or Autoindent mode is not set, the cursor is positioned in column one. This routine also resets the Wrapped flag for the previously current line, indicating the end of the paragraph to *EditReformat.*
Restrictions	None
See Also	EditLongLine EditReformat EditShortLine

EditO [O.ED]

Declaration	`procedure EditO;`
Usage	EditO;
Parameters	None
Function	This routine is the default text menu processor for the editor. It displays the message "<Text>" on the command line, waits for input, and then calls the appropriate toolbox routine to process the command.
Remarks	If a different mapping of comamnds is desired, a new version of this procedure may be created using the default as a guide.
Restrictions	None
See Also	EditK EditQ

EditOffblock [USER.ED]

Declaration	`procedure EditOffblock;`
Usage	EditOffblock;
Parameters	None
Function	This routine resets the Inblock flag in every text line in every text stream in the editing system.
Remarks	It is used by block routines when they determine that a block has been made discontiguous or corrupt. The block limit pointers are not disturbed.
Restrictions	None

EditPrccmd [DISP.ED]

Declaration	**procedure** EditPrccmd (Ch : byte);
Usage	EditPrccmd (Ch);
Parameters	Ch : byte to be interpreted as a command.
Function	This routine interprets the byte passed as a command, and calls the appropriate toolbox routine to process that command. If the byte passed is not mapped to a command, no action is performed.
Restrictions	None

EditPrctxt [USER.ED]

Declaration `procedure EditPrctxt (Ch : byte);`

Usage EditPrctxt (Ch);

Parameters Ch : ASCII code of character to insert into text.

Function This routine inserts the specified character into the current text stream at the current cursor position.

Remarks If Insert mode is set for the current window, text to the right of the cursor is pushed to the right to make room for the character. If Insert mode is reset, the character replaces the character at the current cursor position. If the length of the current line is increased beyond the next 16-byte boundary, *EditPrctxt* will attempt to allocate more space for the text. If it is unable to do this, it will display a message indicating so. If the character inserted pushes the cursor past the right edge of the screen, the window is horizontally scrolled. If Wordwrap mode is set for the current window and the character inserted pushes the cursor past the right margin, the word preceding the cursor is removed from the current line and inserted on a new line immediately below. The current line is always updated on the screen to reflect the change.

Restrictions None

See Also EditPrctxt

EditPushtbf [USER.ED]

Declaration	`procedure EditPushtbf (Ch : byte);`
Usage	EditPushtbf (Ch);
Parameters	Ch : ASCII code of character to be placed in buffer
Function	This routine pushes the specified character onto the front of the typeahead buffer so that it will be the next character read by *EditGetinput*.
Remarks	This is useful for implementing macros. If the typeahead buffer overflows, an error message is displayed and the buffer is cleared.
Restrictions	None
See Also	EditGetinput

EditQ [Q.ED]

Declaration	`procedure EditQ;`
Usage	EditQ;
Parameters	None
Function	This routine is the default window menu processor for the editor. It displays the message "<Window>" on the command line, waits for input, and then calls the appropriate toolbox routine to process the command.
Remarks	If a different mapping of commands is desired, a new version of this procedure may be created using the default as a guide.
Restrictions	None
See Also	EditK EditO

EditRealign [USER.ED]

Declaration	`procedure EditRealign;`
Usage	EditRealign;
Parameters	None
Function	This routine updates the relative line number, current line pointer, and top line pointer for every window in the system. This is necessary when lines are inserted into or deleted from text streams.
Remarks	It assumes that Curline and Topline are defined for each window; this is assured if *EditDelline* is used to delete lines from a text stream.
Restrictions	None
See Also	EditDelline

EditReatxtfil [USER.ED]

Declaration	`procedure EditReatxtfil (Fn : Varstring);`
Usage	EditReatxtfil (Fn);
Parameters	Fn : pathname of file to read
Function	This routine attempts to read the file named by *Fn*, and insert its text into the current window after the current line.
Remarks	If the file does not exist or some other I/O error occurs, an appropriate error message is displayed and the operation is aborted. If memory is insufficient to read the entire file it reads what it is able to and then displays an error message. The cursor position is not changed. The operation may be aborted at any time during the read with Ctrl-U.
Restrictions	None

EditReformat [CMD.ED]

Declaration `procedure EditReformat;`

Usage EditReformat;

Parameters None

Function This routine reformats text lines to fit within the current margins, starting at the cursor position. It will move words down from the end of a line or bring words up a line to fill in as much space as possible. It continues to reformat until it encounters either the end of the text stream or a line in which the Wrapped bit is not set.

Remarks If the routine causes new lines to be inserted, they are given the Wrapped attribute unless they are eventually the last line in the paragraph. This routine will operate whether or not Wordwrap mode is set for the current window.

Restrictions None

See Also EditLongLine
EditNewLine

EditReplace [QCMD.ED]

Declaration	`procedure EditReplace;`
Usage	EditReplace;
Parameters	None
Function	This routine finds a pattern in the current text stream, replacing it if a match is found.
Restrictions	None
See Also	EditFind

EditRightChar [CMD.ED]

Declaration	`procedure EditRightChar;`
Usage	EditRightChar;
Parameters	None
Function	This routine moves the cursor in the current window right one character.
Remarks	If the cursor is beyond the current line's buffer length, it is moved rightward anyway. If text is subsequently typed beyond the buffer length, *EditPrctxt* will attempt memory allocation.
Restrictions	None
See Also	EditLeftChar EditPrctxt

EditRightWord [CMD.ED]

Declaration	`procedure EditRightWord;`
Usage	EditRightWord;
Parameters	None
Function	This routine moves the cursor in the current window right one word.
Remarks	If the cursor is beyond the last non-blank character on the current line, it is moved to the beginning of the line below, if that line exists. If the cursor is on a space, it is moved to the first non-blank character to the right. If the cursor is not on a space, it is moved across all characters in the same class, and then across spaces, if any. The three character classes are alphanumerics, punctuation characters, and spaces. If the cursor is beyond the last non-blank character in the text stream, no action is performed.
Restrictions	None
See Also	EditLeftWord

EditSchedule [TASK.ED]

Declaration	`procedure EditSchedule;`
Usage	EditSchedule;
Parameters	None
Function	This routine determines if a character is available to process on the typeahead buffer, and if so, the input classifier is called. If no input is present, the background process is executed, which updates the screen, etc.
Restrictions	None
See Also	EditSystem EditBackground

EditScrollDown [CMD.ED]

Declaration	`procedure EditScrollDown;`
Usage	EditScrollDown;
Parameters	None
Function	This routine slides the current window down one line over its text stream.
Remarks	If the cursor is on the topmost displayed line of the window, it is moved down one line to keep it within the window. If the last line in the text stream is the only line displayed in the window, no action is performed.
Restrictions	None
See Also	EditScrollUp

EditScrollUp [CMD.ED]

Declaration	`procedure EditScrollUp;`
Usage	EditScrollUp;
Parameters	None
Function	This routine slides the current window up one line over its text stream.
Remarks	If the cursor is on the last displayed line in the window, it is moved up one line to keep it within the window. If the top line of the text stream is currently displayed at the top of the window, no action is performed.
Restrictions	None
See Also	EditScrollDown

EditSetLeftMargin [OCMD.ED]

Declaration

```
procedure EditSetLeftMargin (No : integer);
```

Usage

EditSetLeftMargin (No);

Parameters

No : column to set left margin to.

Function

This routine sets the left margin for the current window to the number passed to it.

Remarks

If the number is less than 1 or greater than the current right margin, an error message is displayed indicating the operation is not possible.

Restrictions

None

See Also

EditCpsetlm
EditCpsetrm

EditSetMarker [KCMD.ED]

Declaration	`procedure EditSetMarker (m : byte);`
Usage	EditSetMarker (m);
Parameters	m : marker number to set.
Function	This routine sets the specified marker to the current line in the current window.
Remarks	If the specified number is less than 1 or greater than Maxmarker, an error message is displayed indicating the operation is not possible, and the marker is undisturbed. Otherwise, the marker is set to the current line, and the previous value of the marker is destroyed.
Restrictions	None
See Also	EditCpsetmrk EditJumpMarker

EditSetRightMargin [OCMD.ED]

Declaration	`procedure EditSetRightMargin (No : integer);`
Usage	EditSetRightMargin (No);
Parameters	No : column number to set.
Function	This routine sets the right margin for the current window to the column specified.
Remarks	If the number is greater than Maxint or less than the current left margin, an error message is displayed indicating the operation is not possible.
Restrictions	None
See Also	Edit Cpsetrm

EditSetUndoLimit [OCMD.ED]

Declaration	`procedure EditSetUndoLimit (Limit : integer);`
Usage	EditSetUndoLimit (Limit);
Parameters	Limit : new Undo stack size.
Function	This routine sets the maximum number of lines that will be retained for *Undo*.
Remarks	If the number passed is greater than Maxint or less than 1, no operation is performed.
Restrictions	None

EditShiftLine [CMD.ED]

Declaration	`procedure EditShiftLine (Lp : Plinedesc);`
Usage	EditShiftLine (Lp);
Parameters	Lp : pointer to line to be shifted
Function	This routine is local to *EditReformat*, and is used to shift the text on that line right to ensure that the first non-blank character on the line falls on or to the right of the current left margin. This improves clarity in the splicing algorithms.
Restrictions	None
See Also	EditReformat

EditShortline [CMD.ED]

Declaration	`procedure EditShortline;`
Usage	EditShortline;
Parameters	None
Function	This routine is local to *EditReformat*, and is used to remove words from the line below the current line and append them to the current line.
Remarks	It will only remove enough text to fill the space between the end of the current line and the right margin without extending beyond the margin. If there is no line below the current line, or if that line does not have the Wrapped bit set, the reformat operation is terminated.
Restrictions	None
See Also	EditLongLine EditReformat EditNewLine

EditSystem [TASK.ED]

Declaration	`procedure EditSystem;`
Usage	EditSystem;
Parameters	None
Function	This routine serves as an example for an editor main loop. It repeatedly calls *EditSchedule* until the variable Rundown is true, at which time it exits.
Restrictions	None
See Also	EditSchedule

EditTab [CMD.ED]

Declaration `procedure EditTab;`

Usage EditTab;

Parameters None

Function This routine moves the cursor to the next tab stop to the right. If Insert mode is set for the current window, tabs are inserted into the current line as the cursor is moved. If the resulting longer line exceeds the current line's buffer length, *EditTab* will attempt to allocate new memory for the line. If it could not allocate the memory, an error message is displayed and no action is performed.

Restrictions None

EditToggleAutoindent [QCMD.ED]

Declaration	`procedure EditToggleAutoindent;`
Usage	EditToggleAutoindent;
Parameters	None
Function	This routine is local to *Autoindent* mode for the current window. This mode is used by *EditNewLine* to determine where to position the cursor when <RETURN> is pressed.
Restrictions	None
See Also	EditNewLine

EditToggleInsert [CMD.ED]

Declaration	`procedure EditToggleInsert;`
Usage	EditToggleInsert;
Parameters	None
Function	This routine toggles Insert mode in the current window.
Restrictions	None

EditToggleWordwrap [OCMD.ED]

Declaration	`procedure EditToggleWordwrap;`
Usage	EditToggleWordwrap;
Parameters	None
Function	This routine toggle *Wordwrap* mode for the current window. This mode is used by *EditPrctxt* to determine whether text should automatically wrap to the next line when it exceeds the right margin.
Restrictions	None

EditTopBlock [QCMD.ED]

Declaration	`procedure EditTopBlock;`
Usage	EditTopBlock;
Parameters	None
Function	This routine moves the cursor to the top line of the current block.
Remarks	If the block is not defined, an error message is displayed indicating that the operation is not possible. If the block is defined in a window other than the current one, the current window's cursor position is saved and the cursor is moved to the window containing the block. The column position is not changed.
Restrictions	None
See Also	EditBottomBlock

EditUndo [CMD.ED]

Declaration	`procedure EditUndo;`
Usage	EditUndo;
Parameters	None
Function	This routine removes a text line from the Undo stack and inserts it into the current text stream above the current line.
Remarks	If the Undo stack size is set to zero, or if there are no lines on the stack to be removed, no action is performed.
Restrictions	None

EditUpcase [USER.ED]

Declaration	`procedure EditUpcase (var s : Varstring);`
Usage	EditUpcase (s);
Parameters	s : string to convert
Function	This routine changes all lowercase letters in s to their equivalent uppercase letters, and returns the new string in s.
Restrictions	None

EditUpdphyscr [SCREEN.ED]

Declaration	`procedure EditUpdphyscr;`
Usage	EditUpdphyscr;
Parameters	None
Function	This routine updates the physical screen. It updates the current line of the current window first, then the command line. If any input is pending, the operation is terminated at this point; otherwise, all windows including the current one are updated completely with *EditUpdwindow.*
Remarks	If input is encountered at any time during this process, the operation is terminated. However, if Intrflag is set to Nointrpt, the entir screen will be updated, even if input is present.
Restrictions	None
See Also	EditUpdwindow

EditUpdrowasm [SCREEN.ED]

Declaration	`procedure EditUpdrowasm (Row : byte);`
Usage	EditUpdrowasm (Row);
Parameters	Row : physical screen row to update.
Function	This routine is passed a row number to update. It takes the data for that screen row from the Screen array and copies it directly to screen memory.
Remarks	If Retracemode is set, inline code is used to wait for the display processor to indicate vertical retrace is active, so that no "snow" is observed during updating. This is the lowest-level screen updating routine.
Restrictions	The parameter must be in the range 1-Defnorows; if it is not, a run-time array bounds overflow error will occur. If array bounds checking is turned off and the row number is out of range, the results are unpredictable.

EditUpdwindow [SCREEN.ED]

Declaration	`procedure EditUpdwindow (w : Pwindesc);`
Usage	EditUpdwindow(Curwin);
Parameters	w : The window to be updated
Function	This routine updates the status line, then updates every line in the text region by calling *EditWrline* and *EditUpdrowasm*.
Restrictions	None
See Also	EditUpdwinsl EditUpdphyscr

EditUpdwinsl [SCREEN.ED]

Declaration	`procedure EditUpdwinsl (w : Pwindesc);`
Usage	EditUpdwinsl(Curwin);
Parameters	w : The window whose status line is to be updated
Function	This routine updates the status line for the specified window.
Restrictions	None
See Also	EditUpdwindow EditUpdphyscr

EditUpLine [CMD.ED]

Declaration	`procedure EditUpLine;`
Usage	EditUpLine;
Parameters	None
Function	This routine moves the cursor in the current window up one line.
Remarks	If the cursor is on the topmost displayed line in the window, the window will be scrolled up one line to keep the cursor within the window. If the cursor is on the topmost line in the text stream, no action is performed.
Restrictions	None
See Also	EditDownLine

EditUpPage [CMD.ED]

Declaration	`procedure EditUpPage;`
Usage	EditUpPage;
Parameters	None
Function	This routine slides the current window up one page over its text stream.
Remarks	The total number of lines scrolled is one less than the number of lines displayed in the window. The window will never be scrolled so as to position the top line of the text stream below the topmost screen line of the window. If the top line of the text stream is displayed at the top of the window, no action is performed.
Restrictions	None
See Also	EditDownPage

EditUserpush [INPUT.ED]

Declaration `procedure EditUserpush (s : Varstring);`

Usage EditUserpush (s);

Parameters s : String to be pushed onto the typeahead buffer.

Function This routine pushes the string passed onto the front of the typeahead buffer in reverse order, so that the characters of the string will be removed by *Edit-Getinput* in the correct order. *EditPushtbf* is called to perform the actual operation on each element of the string.

Remarks This routine is useful in implementing macro processing. If the string passed is NULL, no action is performed.

Restrictions None

See Also EditGetinput
EditPushtbf

EditWindowBottomFile [QCMD.ED]

Declaration	`procedure EditWindowBottomFile;`
Usage	EditWindowBottomFile;
Parameters	None
Function	This routine moves the cursor in the current window to the bottom of its text stream. The cursor column is set to 1, and the last line of the text stream is displayed on the top line of the window.
Restrictions	None
See Also	EditWindowTopFile

EditWindowCreate [OCMD.ED]

Declaration `procedure EditWindowCreate (Size : byte; Win : byte);`

Usage EditWindowCreate (Size, Win);

Parameters Size : number of screen lines to be given to the new window Win : window number to be compressed

Function This routine creates a new window and inserts it into the linked list of displayed windows. The requested size must be at least three lines, one status line and two text lines. The new window may not compress the specified one smaller than three lines. If either of these conditions occurs, an appropriate error message is displayed and the operation is terminated.

Remarks If the new window cannot be created using *EditCrewindow*, an out of memory error is reported. Otherwise, the window is initialized and inserted into the linked list. The compressed window's Lastlineno field is adjusted to compress its visible span. If that window's current line was below the new lower limit, it is moved to the last displayed line of that window.

Restrictions None

See Also EditCrewindow

EditWindowDelete [OCMD.ED]

Declaration `procedure EditWindowDelete (Wno : byte);`

Usage EditWindowDelete (Wno);

Parameters Wno : window number to delete

Function This routine deletes a window from the linked list of display windows. If there is only one window displayed, an error message is reported. Otherwise, Wno is interpreted modulo the number of windows defined.

Remarks If the requested number is window 1, the second window will be given the freed space, otherwise the window above the deleted one will claim the freed space. The window's text stream will be deleted but not placed on the undo stack, so it is lost. If the deleted window contained the currently defined block, it also is deleted and the limit pointers reset to nil. If the window is linked to another, its text stream will not be deleted, and the text will be present in the other window; but the link will be destroyed. After the text is deleted, the window record is placed on the free list.

Restrictions None

EditWindowDeleteText [QCMD.ED]

Declaration	`procedure EditWindowDeleteText;`
Usage	EditWindowDeleteText;
Parameters	None
Function	This routine deletes all text from the current window. The text lines are not placed on the undo stack, so they are lost. The filename is reset to noname, and a blank text line is created for the new first line of the window.
Remarks	If another window is linked to the cleared one, its text stream is deleted as well, and the link is destroyed. If the block was in the current window or a window linked to it, it too is deleted, and its limit pointers are reset to nil. There are no error conditions.
Restrictions	None

EditWindowDown [QCMD.ED]

Declaration	`procedure EditWindowDown;`
Usage	EditWindowDown;
Parameters	None
Function	This routine changes the current window to the window immediately below. The cursor position in the old current window is saved, and the cursor position in the new current window will be the previously saved cursor position there. If the current window is the bottom window on the screen, the cursor is moved to window 1. There are no error conditions, and no manipulation of text is performed.
Restrictions	None
See Also	EditWindowUp

EditWindowGoto [QCMD.ED]

Declaration `procedure EditWindowGoto (Wno : byte);`

Usage EditWindowGoto (Wno);

Parameters Wno : window number to move to

Function This routine changes the current window to the window number specified. The cursor position in the old current window is saved, and the cursor position in the new current window will be the previously saved cursor position there. The requested window number is interpreted modulo the number of windows currently defined. There are no error conditions, and no manipulation of text is performed.

Restrictions None

EditWindowLink [QCMD.ED]

Declaration `procedure EditWindowLink (Wto : byte; Wfrom : byte);`

Usage EditWindowLink (Wto, Wfrom);

Parameters Wto : window number of destination window
Wfrom : window number of source window

Function This routine links two windows together. The text in window number *Wto* is deleted (unless another window is linked to it), and its stream is made identical to that of window number *Wfrom*. Both parameters are interpreted modulo the number of windows currently displayed.

Remarks If both window numbers are the same or if their windows already reference the same text stream, an error message is displayed indicating that the operation was not possible. If the operation was successful, any further manipulation of text in one of the linked windows will be reflected in the other's text, though they remain able to be scrolled independently.

Restrictions None

EditWindowTopFile [QCMD.ED]

Declaration	`procedure EditWindowTopFile;`
Usage	EditWindowTopFile;
Parameters	None
Function	This routine moves the cursor in the current window to the top of its text stream. The line and column positions are set to 1.
Restrictions	None
See Also	EditWindowBottomFile

EditWindowUp [QCMD.ED]

Declaration	`procedure EditWindowUp;`
Usage	EditWindowUp;
Parameters	None
Function	This routine changes the current window to the window immediately above. The cursor position in the old current window is saved, and the cursor position in the new current window will be the previously saved cursor position there.
Remarks	If the current window is window 1, the cursor is moved to the bottom window on the screen. There are no error conditions, and no manipulation of text is performed.
Restrictions	None
See Also	EditWindowDown

EditZapcmdnam [USER.ED]

Declaration	`procedure EditZapcmdnam;`
Usage	EditZapcmdnam;
Parameters	None
Function	This routine clears the command line. It sets the command line string to spaces to blank it, and sets Cmdcol to 1 so that *EditAppcmdnam* and *EditAskfor* will operate starting in column one.
Restrictions	None
See Also	EditAppcmdnam EditAskfor

MoveFromScreen [SCREEN.ED]

Declaration	`procedure MoveFromScreen(Var Source,Dest; Length: Integer);`
Usage	MoveFromScreen(ScreenLoc, Dest, Size);
Parameters	Source : The screen memory location to move from Dest : The non-screen memory location to move to Size : The number of bytes to move
Function	Move memory, as Turbo's *Move* procedure, but assume that the source is in video memory. Prevent screen flicker based on this assumption, unless *RetraceMode* is false. Timing is *very* tight: if the code were 1 clock cycle slower, it would cause flicker.
Restrictions	Should be used only to move data from the screen.
See Also	MoveToScreen

MoveToScreen [SCREEN.ED]

Declaration	`procedure MoveToScreen(Var Source,Dest; Length: Integer);`
Usage	MoveToScreen(ScreenLoc, Dest, Size);
Parameters	Source : The non-screen memory location to move from Dest : The screen memory location to move to Size: The number of bytes to move
Function	Move memory, as Turbo's *Move* procedure, but assume that the target is in video memory. Prevent screen flicker based on this assumption, unless *RetraceMode* is false. Timing is *very* tight: if the code were 1 clock cycle slower, it would cause flicker.
Restrictions	Should be used only to move data to the screen.
See Also	MoveFromScreen

Pokechr [INPUT.ED]

Declaration `procedure Pokechr (Ch : char);`

Usage Pokechr (Ch);

Parameters Ch : character to place on the typeahead buffer.

Function This routine is called to place a character on the end of the typeahead buffer, in a queue style. If the character passed is a Ctrl-U, *EditAbort* is called directly. If the typeahead buffer is full, it will be emptied, an error message will be displayed, and the character passed will be lost.

Restrictions None

See Also EditAbort

SUBJECT INDEX

PROCEDURES AND FUNCTIONS INDEX

60 DAY MONEY-BACK GUARANTEE

CATALOG OF BORLAND PRODUCTS

BORLAND
INTERNATIONAL
4585 Scotts Valley Drive
Scotts Valley, CA 95066

Available at better dealers nationwide. Call (800) 556-2283 for the dealer nearest you. To order by Credit Card call (800) 255-8008, CA (800) 742-1133

SIDEKICK® VERSION 1.5

INFOWORLD'S SOFTWARE PRODUCT OF THE YEAR

Whether you're running WordStar™, Lotus™, dBase™, or any other program, SIDEKICK puts all these desktop accessories at your fingertips. Instantly.

A full-screen WordStar-like Editor You may jot down notes and edit files up to 25 pages long.

A Phone Directory for your names, addresses and telephone numbers. Finding a name or a number becomes a snap.

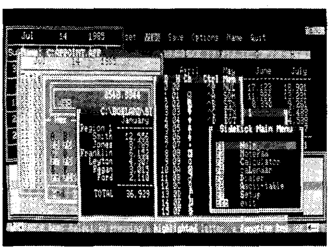
An Autodialer for all your phone calls. It will look up and dial telephone numbers for you. (A modem is required to use this function.)

A Monthly Calendar functional from year 1901 through year 2099.

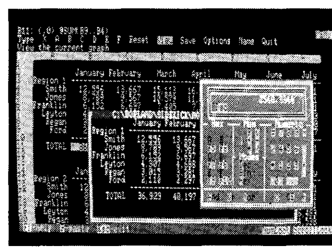
A Datebook to remind you of important meetings and appointments.

A full-featured Calculator ideal for business use. It also performs decimal to hexadecimal to binary conversions.

An ASCII Table for easy reference.



All the SIDEKICK windows stacked up over Lotus 1-2-3. From bottom to top: SIDEKICK'S "Menu Window," ASCII Table, Notepad, Calculator, Datebook, Monthly Calendar and Phone Dialer.



Here's SIDEKICK running over Lotus 1-2-3. In the SIDEKICK Notepad you'll notice data that's been imported directly from the Lotus screen. In the upper right you can see the Calculator.

The Critics' Choice

"In a simple, beautiful implementation of WordStar's™ block copy commands, SIDEKICK can transport all or any part of the display screen (even an area overlaid by the notepad display) to the notepad."
—Charles Petzold, PC MAGAZINE

"SIDEKICK deserves a place in every PC."
—Garry Ray, PC WEEK

"SIDEKICK is by far the best we've seen. It is also the least expensive." **—Ron Mansfield, ENTREPRENEUR**

"If you use a PC, get SIDEKICK. You'll soon become dependent on it." **—Jerry Pournelle, BYTE**

SIDEKICK IS AN UNPARALLELED BARGAIN AT ONLY $54.95 (copy-protected) OR $84.95 (not copy-protected)

Minimum System Configuration: SIDEKICK is available now for your IBM PC, XT, AT, PCjr., and 100% compatible microcomputers. The IBM PC jr. will only accept the SIDEKICK not copy-protected version. Your computer must have at least 128K RAM, one disk drive and PC-DOS 2.0 or greater. A Hayes™ compatible modem, IBM PCjr.™ internal modem, or AT&T® Modem 4000 is required for the autodialer function.

BORLAND
INTERNATIONAL

SuperKey®

INCREASE YOUR PRODUCTIVITY BY 50% OR YOUR MONEY BACK

SuperKey turns 1,000 keystrokes into 1!

Yes, SuperKey can *record* lengthy keystroke sequences and play them back at the touch of a single key. Instantly. Like Magic.

Say, for example, you want to add a column of figures in 1-2-3. Without SuperKey you'd have to type seven keystrokes just to get started. ["shift-@-s-u-m-shift-("]. With SuperKey you can turn those 7 keystrokes into 1.

SuperKey keeps your 'confidential' files. . .CONFIDENTIAL!

Time after time you've experienced it: anyone can walk up to your PC, and read your confidential files (tax returns, business plans, customer lists, personal letters. . .).

With SuperKey you can encrypt any file, even while running another program. As long as you keep the password secret, only YOU can decode your file. SuperKey implements the U.S. government Data Encryption Standard (DES).

SuperKey helps protect your capital investment.

SuperKey, at your convenience, will make your screen go blank after a predetermined time of screen/keyboard inactivity. You've paid hard-earned money for your PC. SuperKey will protect your monitor's precious phosphor. . .and your investment.

SuperKey protects your work from intruders while you take a break.

Now you can lock your keyboard at any time. Prevent anyone from changing hours of work. Type in your secret password and everything comes back to life. . .just as you left it.

SUPERKEY is now available for an unbelievable $69.95 (not copy-protected).

Minimum System Configuration: SUPERKEY is compatible with your IBM PC, XT, AT, PCjr. and 100% compatible microcomputers. Your computer must have at least 128K RAM, one disk drive and PC-DOS 2.0 or greater.

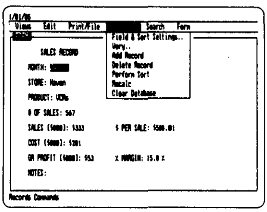
BORLAND
INTERNATIONAL

REFLEX

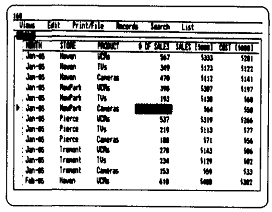
THE ANALYST™

Reflex™ is the most amazing and easy to use database management system. And if you already use Lotus 1-2-3, dBASE or PFS File, you need Reflex—because it's a totally new way to look at your data. It shows you patterns and interrelationships you didn't know were there, because they were hidden in data and numbers. It's also the greatest report generator for 1-2-3.

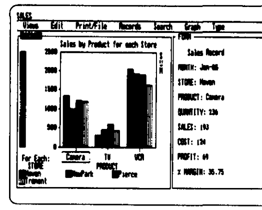
REFLEX OPENS MULTIPLE WINDOWS WITH NEW VIEWS AND GRAPHIC INSIGHTS INTO YOUR DATA.



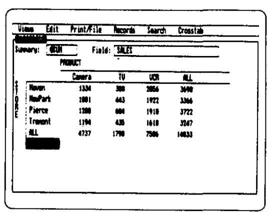
The FORM VIEW lets you build and view your database.



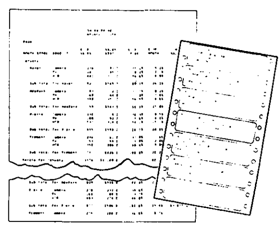
The LIST VIEW lets you put data in tabular List form just like a spreadsheet.



The GRAPH VIEW gives you instant interactive graphic representations.



The CROSSTAB VIEW gives you amazing "cross-referenced" pictures of the links and relationships hidden in your data.



The REPORT VIEW allows you to import and export to and from Reflex, 1-2-3, dBASE, PFS File and other applications and prints out information in the formats you want.

So Reflex shows you. Instant answers. Instant pictures. Instant analysis. Instant understanding.

THE CRITICS' CHOICE:

"The next generation of software has officially arrived."

Peter Norton, PC WEEK

"Reflex is one of the most powerful database programs on the market. Its multiple views, interactive windows and graphics, great report writer, pull-down menus and cross tabulation make this one of the best programs we have seen in a long time . . . The program is easy to use and not intimidating to the novice . . . Reflex not only handles the usual database functions such as sorting and searching, but also "what-if" and statistical analysis . . . it can create interactive graphics with the graphics module. The separate report module is one of the best we've ever seen."

Marc Stern, INFOWORLD

Minimum System Requirements: Reflex runs on the IBM® PC, XT, AT and compatibles. 384K RAM minimum. IBM Color Graphics Adapter®, Hercules Monochrome Graphics Card™, or equivalent. PC-DOS 2.0 or greater. Hard disk and mouse optional. Lotus 1-2-3, dBASE, or PFS File optional.

BORLAND
INTERNATIONAL

If you use an IBM PC, you need

TURBO Lightning™

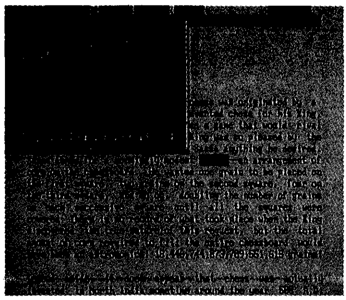
Turbo Lightning™ teams up with the Random House spelling Dictionary® to check your spelling as you type!

Turbo Lightning, using the 83,000-word Random House Dictionary, checks your spelling *as you type*. If you misspell a word, it alerts you with a 'beep'. At the touch of a key, *Turbo* Lightning opens a window on top of your application program and suggests the correct spelling. Just press ENTER and the misspelled word is instantly replaced with the correct word. It's that easy!

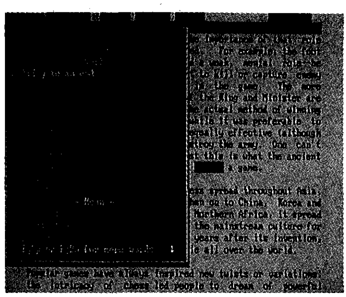
Turbo Lightning works hand-in-hand with the Random House Thesaurus® to give you instant access to synonyms.

Turbo Lightning lets you choose just the right word from a list of alternates, so you don't say the same thing the same way every time. Once *Turbo* Lightning opens the Thesaurus window, you see a list of alternate words, organized by parts of speech. You just select the word you want, press ENTER and your new word will instantly replace the original word. Pure magic!

If you ever write a word, think a word, or say a word, you need Turbo Lightning.



The Turbo Lightning Dictionary.



The Turbo Lightning Thesaurus.

Turbo Lightning's intelligence lets you teach it new words. The more you use Turbo Lightning, the smarter it gets!

You can also *teach* your new *Turbo* Lightning your name, business associates' names, street names, addresses, correct capitalizations, and any specialized words you use frequently. Teach *Turbo Lightning* once, and it knows forever.

Turbo Lightning™ is the engine that powers Borland's Turbo Lightning Library™.

Turbo Lightning brings electronic power to the Random House Dictionary® and Random House Thesaurus®. They're at your fingertips —even while you're running other programs. *Turbo* Lightning will also 'drive' soon-to-be-released encyclopedias, extended thesauruses, specialized dictionaries, and many other popular reference works. You get a head start with this first volume in the *Turbo* Lightning Library.

And because *Turbo* Lightning is a Borland product, you know you can rely on our quality, our 60-day money-back guarantee, and our eminently fair prices.

BORLAND
INTERNATIONAL

Suggested Retail Price $99.95
(not copy-protected)

Minimum System Requirements:
128K IBM PC® or 100% compatible computer, with 2 floppy disk drives and PC-DOS (MS-DOS) 2.0 or greater.

WITH COMMENTED SOURCE CODE !

TURBOPASCAL®

VERSION 3.0

THE CRITICS' CHOICE:

"Language deal of the century . . . Turbo Pascal: it introduces a new programming environment and runs like magic."
—*Jeff Duntemann, PC Magazine*

"Most Pascal compilers barely fit on a disk, but Turbo Pascal packs an editor, compiler, linker, and run-time library into just 39K bytes of random-access memory."
—*Dave Garland, Popular Computing*

"What I think the computer industry is headed for: well - documented, standard, plenty of good features, and a reasonable price." **—*Jerry Pournelle, BYTE***

THE FEATURES:

One-Step Compile: No hunting & fishing expeditions! Turbo finds the errors, takes you to them, lets you correct, then instantly recompiles. You're off and running in record time.

Built-in Interactive Editor: WordStar-like easy editing lets you debug quickly.

Automatic Overlays: Fits big programs into small amounts of memory.

Microcalc: A sample spreadsheet on your disk with ready-to-compile source code.

IBM PC VERSION: Supports Turtle Graphics, Color, Sound, Full Tree Directories, Window Routines, Input/Output Redirection and much more.

LOOK AT TURBO NOW!

- ☐ More than 400,000 users worldwide.
- ☐ TURBO PASCAL is proclaimed as the de facto industry standard.
- ☐ TURBO PASCAL PC MAGAZINE'S award for technical excellence.
- ☐ TURBO PASCAL named 'Most Significant Product of the Year' by PC WEEK.
- ☐ TURBO PASCAL 3.0 — the FASTEST Pascal development environment on the planet, PERIOD.

OPTIONS FOR 16-BIT SYSTEMS:

8087 math co-processor support for intensive calculations.

Binary Coded Decimals (BCD): Eliminates round-off error! A *must* for any serious business application. (No additional hardware required.)

Turbo Pascal 3.0 is available now for $69.95.

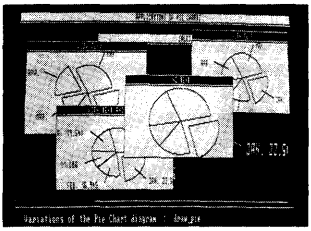
Options: Turbo Pascal with 8087 or BCD at a low $109.90. Turbo Pascal with both options (8087 and BCD) priced at $124.95.

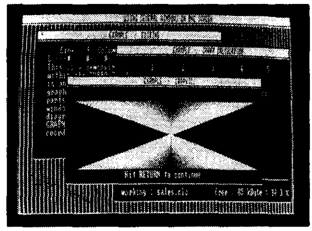
MINIMUM SYSTEM CONFIGURATION: To use Turbo Pascal 3.0 requires 64K RAM, one disk drive, Z-80, 8088/86, 80186 or 80286 microprocessor running either CP/M-80 2.2 or greater, CP/M-86 1.1 or greater, MS-DOS 2.0 or greater or PC-DOS 2.0 greater, MS-DOS 2.0 or greater or PC-DOS 2.0 or greater. A XENIX version of Turbo Pascal will soon be announced, and before the end of the year, Turbo Pascal will be running on most 68000-based microcomputers.

BORLAND
I N T E R N A T I O N A L

TURBO *GRAPHIX* TOOLBOX™

HIGH RESOLUTION GRAPHICS AND GRAPHIC WINDOW MANAGEMENT FOR THE IBM PC





Dazzling graphics and painless windows.

The Turbo Graphix Toolbox™ will give even a beginning programmer the expert's edge. It's a complete library of Pascal procedures that include:

- Full graphics window management.
- Tools that allow you to draw and hatch pie charts, bar charts, circles, rectangles and a full range of geometric shapes.
- Procedures that save and restore graphic images to and from disk.
- Functions that allow you to precisely plot curves.
- Tools that allow you to create animation or solve those difficult curve fitting problems.

No sweat and no royalties.

You can incorporate part, or all of these tools in your programs, and yet, we won't charge you any royalties. Best of all, these functions and procedures come complete with source code on disk ready to compile!

John Markoff & Paul Freiberger, syndicated columnists:

"While most people only talk about low-cost personal computer software, Borland has been doing something about it. And Borland provides good technical support as part of the price."

***Turbo Graphix Toolbox—only $54.95** (not copy protected).*

Minimum System Configuration: Turbo Graphix Toolbox is available today for your computer running Turbo Pascal 2.0 or greater for PC-DOS, or truly compatible MS-DOS. Your computer must have at least 128K RAM, one disk drive and PC-DOS 2.0 or greater, and MS-DOS 2.0 or greater with IBM Graphics Adapter or Enhanced Graphics Adapter, IBM-compatible Graphics Adapter, or Hercules Graphics Card.

BORLAND
INTERNATIONAL

TURBO DATABASE TOOLBOX™

Is The Perfect Complement To Turbo Pascal.

It contains a complete library of Pascal procedures that allows you to sort and search your data and build powerful applications. It's another set of tools from Borland that will give even the beginning programmer the expert's edge.

THE TOOLS YOU NEED!

TURBOACCESS Files Using B+Trees - The best way to organize and search your data. Makes it possible to access records in a file using key words instead of numbers. Now available with complete source code on disk ready to be included in your programs.

TURBOSORT - The fastest way to sort data—and TURBOSORT is the method preferred by knowledgeable professionals. Includes source code.

GINST (General Installation Program) - Gets your programs up and running on other terminals. This feature alone will save hours of work and research. Adds tremendous value to all your programs.

GET STARTED RIGHT AWAY: FREE DATABASE!

Included on every Toolbox disk is the source code to a working database which demonstrates the power and simplicity of our Turbo Access search system. Modify it to suit your individual needs or just compile it and run. Remember, no royalties!

THE CRITICS' CHOICE!

"The tools include a B+tree search and a sorting system. I've seen stuff like this, but not as well thought out, sell for hundreds of dollars."

—Jerry Pournelle, BYTE MAGAZINE

"The Turbo Database Toolbox is solid enough and useful enough to come recommended."

—Jeff Duntemann, PC TECH JOURNAL

TURBO DATABASE TOOLBOX—ONLY $54.95 (not copy-protected).

Minimum system configurations: 64K RAM and one disk drive. 16-bit systems: TURBO PASCAL 2.0 or greater for MS-DOS or PC-DOS 2.0 or greater. TURBO PASCAL 2.1 or greater for CP/M-86 1.1 or greater. Eight-bit systems: TURBO PASCAL 2.0 or greater for CP/M-80 2.2 or greater.

BORLAND
INTERNATIONAL

HOW TO BUY BORLAND SOFTWARE

BORLAND
INTERNATIONAL

NOT COPY-PROTECTED

For The Dealer Nearest You, Call
(800) 556-2283

60 DAY MONEY-BACK GUARANTEE

To Order By Credit Card, Call
(800) 255-8008

In California (800) 742-113

TURBO EDITOR TOOLBOX

It's All You Need To Build Your Own Text Editor Or Word Processor.

Build your own lightning-fast editor and incorporate it into your Turbo Pascal programs. Turbo Editor Toolbox™ gives you easy-to-install modules. Now you can integrate a fast and powerful editor into your own programs. You get the source code, the manual and the know how.

Create your own word processor. We provide all the editing routines. You plug in the features you want. You could build a WordStar®-like editor with pull-down menus like Microsoft's® Word, and make it work as fast as WordPerfect™.

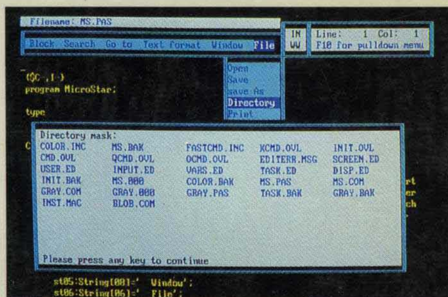
To demonstrate the tremendous power of Turbo Editor Toolbox, we give you the source code for two sample editors:

Simple Editor A complete editor ready to include in your programs. With windows, block commands, and memory-mapped screen routines.

MicroStar™ A full-blown text editor with a complete pull-down menu user interface, plus a lot more. Modify MicroStar's pull-down menu system and include it in your Turbo Pascal programs.

The Turbo Editor Toolbox gives you all the standard features you would expect to find in any word processor:

- Word wrap
- UNDO last change
- Auto indent
- Find and Find/Replace with options
- Set left and right margin
- Block mark, move and copy.
- Tab, insert and overstrike modes, centering, etc.



MicroStar's pull-down menus.

And Turbo Editor Toolbox has features that word processors selling for several hundred dollars can't begin to match. Just to name a few:

- ***RAM-based editor.*** You can edit very large files and yet editing is lightning fast.
- ***Memory-mapped screen routines.*** Instant paging, scrolling and text display.
- ***Keyboard installation.*** Change control keys from WordStar-like commands to any that you prefer.
- ***Multiple windows.*** See and edit up to eight documents—or up to eight parts of the same document—all at the same time.
- ***Multi-Tasking.*** Automatically save your text. Plug in a digital clock . . . an appointment alarm—see how it's done with MicroStar's "background" printing.

Best of all, **source code is included for everything in the Editor Toolbox.** Use any of the Turbo Editor Toolbox's features in your programs. And pay no royalties.

Minimum system configuration: The Turbo Editor Toolbox requires an IBM PC, XT, AT, 3270, PCjr or true compatible with a minimum 192K RAM, running PC-DOS (MS-DOS) 2.0 or greater. You must be using Turbo Pascal 3.0 for IBM and compatibles.

BORLAND INTERNATIONAL

4585 SCOTTS VALLEY DRIVE
SCOTTS VALLEY, CALIFORNIA 95066

ISBN 0-87524-148-4