TURBO Pascal Reference Manual

Version 3.0

BORLAND INTERNATIONAL Inc.
1800 Green Hill Road
Scotts Valley, CA 95066

This edition produced by

ALPHA SYSTEMS CORPORATION
711 Chatsworth Place
San Jose, CA 95128
(408) 297-5594

Third edition, December 1988

This CP/M-only edition of the TURBO Pascal reference manual was typed from a copy of the Second edition by Shirley Welch and Bill Lockwood of Home Word Shop, and corrected and typeset by David A.J. McGlone of Alpha Systems Corporation. Any errors in this edition which were not present in the previous edition are the sole responsibility of the editor, for which you have his apologies. Please bring them to our attention, so that they can be corrected.

The transcription of this edition to disk was done on two Eagle II computers, using Spellbinder Word Processor. Editing was done on a Micromint SB180FX with a Televideo 950 terminal, using Spellbinder Word Processor. Typesetting was done with MagicIndex on a Hewlett Packard LaserJet Plus.

The sans-serif fonts used in this manual are MagicDelite from Computer EdiType Systems. Special symbols such as □ were selected from MagicSymbol, also from CES. The italic font used is typeface 1003 from the Digi-Fonts typeface library.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

INTRODUCTION

This book is a reference manual for the TURBO Pascal system as implemented for the CP/M-80, Z-System, and compatible operating systems. Although making thorough use of examples, it is not meant as a Pascal tutorial or textbook, and at least a basic knowledge of Pascal is assumed.

THE PASCAL LANGUAGE

Pascal is a general-purpose, high-level programming language originally designed by Professor Niklaus Wirth of the Technical University of Zurich, Switzerland and named in honor of Blaise Pascal, the famous French Seventeenth Century philosopher and mathematician.

Professor Wirth's definition of the Pascal language, published in 1971, was intended to aid the teaching of a systematic approach to computer programming, specifically introducing *structured programming*. Pascal has since been used to program almost any task on almost any computer and it is today established as one of the foremost high-level languages, whether the application is education, hobby, or professional programming.

TURBO PASCAL

TURBO Pascal is designed to meet the requirements of all categories of users: it offers the student a friendly interactive environment which greatly aids the learning process; and in the hands of a programmer it becomes an extremely effective development tool providing both compilation and execution times second to none.

TURBO Pascal closely follows the definition of Standard Pascal as defined by K. Jensen and N. Wirth in the *Pascal User Manual and Report*. The few and minor differences are described in Appendix D. In addition to the standard, a number of extensions are provided, such as:

- Absolute address variables
- Bit/byte manipulation
- Direct access to CPU memory and data ports
- Dynamic strings
- Free ordering of sections within declaration part

Full support of operating system facilities
In-line machine code generation
Include files
Logical operations on integers
Overlay system
Program chaining with common variables
Random access data files
Structured constants
Type conversion functions

Furthermore, many extra standard procedures and functions are included to increase the versatility of TURBO Pascal.

STRUCTURE OF THIS MANUAL

The reader may be familiar with earlier editions of this manual, in which the earlier sections covered features common to PC-DOS, MS-DOS, CP/M-86 and CP/M-80 implementations of TURBO Pascal, and later chapters dealt with items that differed among implementations. This edition has been prepared by Alpha Systems Corporation to document the CP/M-80 implementation only. Information on features specific to implementations for incompatible operating systems such as MS-DOS, PC-DOS, and CP/M-86 has been taken out of this edition. Alpha Systems Corporation has a contract with Borland International to sell and support the CP/M-80 version of TURBO Pascal *only*. For copies of the software or the manual for operating systems incompatible with CP/M, contact Borland directly.

TYPOGRAPHY

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

Italics

Italics are used generally for the names of the TURBO editor commands, as in the *insert mode on/off* command. Pre-defined standard identifiers and elements in syntax descriptions (see below) are printed in italics. The meaning of the use of italics thus depends on the context.

Boldface

Boldface is used to mark TURBO menu commands, as the compiler **O**ptions command, and to denote other key combinations, as **Ctrl-K Y**. It is also used to mark reserved words, and to highlight particularly important passages in the text.

Syntax Descriptions

The entire syntax of the Pascal language expressed as *Backus-Naur Forms* is collected in Appendix I, which also describes the typography and special symbols used in these forms.

Where appropriate, syntax descriptions are also used more specifically to show the syntax of single language elements, as in the following syntax description of the function *Concat*:

Concat(*St1*,*St2*(,*StN*))

Reserved words are printed in **boldface**, identifiers used mixed upper and lower case, and elements explained in the text are printed in *italics*.

The text will explain that *St1*, *St2*, and *StN* must be string expressions. The syntax description shows that the word *Concat* must be followed by two or more string expressions, separated by commas and enclosed in parentheses. In other words, the following examples are legal (assuming that *Name* is a string variable):

```
Concat('TURBO',' Pascal')
Concat('TU','RBO',' Pascal')
Concat('T','U','R','B','O',Name)
```

Chapter 1
USING THE TURBO SYSTEM

This chapter describes the installation and use of the TURBO Pascal system, specifically the built-in editor.

1.1. BEFORE USE

Before using TURBO Pascal you should, for your own protection, make a work copy of the distribution diskette and store the original safely away. Remember that the User's License allows you to make as many copies as you need **for your own personal use** and for **backup purposes** only. Use a file-copy program to make the copy, and make sure that all files are successfully transferred.

1.2. IMPORTANT NOTE!!!

TURBO Pascal provides a number of compiler directives to control special runtime facilities such as index checking, recursion, etc. PLEASE NOTICE that the default settings of these directives will optimize execution speed and minimize code size. Thus, a number of run-time facilities (such as index checking and recursion) are de-selected until explicitly selected by the programmer. All compiler directives and their default values are described in Appendix C.

1.3. FILES ON THE DISTRIBUTION DISK

The distribution disk contains the following files:

TURBO.COM
: The TURBO Pascal program: compiler, editor, and all. When you enter the command TURBO on your terminal, this file will load, and TURBO will be up and running.

TURBO.OVR
: Overlay file for TURBO.COM. Needs only be present on the run-time disk if you want to execute .COM files from TURBO.

TURBO.MSG

Messages for the installation program. This file may be translated into any language desired.

.PAS files

Sample Pascal programs.

READ.ME

If present, this file contains the latest corrections or suggestions on the use of the system.

Only TURBO.COM **must** be on your run-time disk. A fully operative TURBO Pascal thus requires only **30 K** of disk space. TURBO.OVR is required only if you want to be able to execute programs from the TURBO menu. TURBO.MSG is needed only if you want on-line compile-time error messages. The TINST files are used only for the installation procedure. The example .PAS files, of course, may be included on the run-time disk if so desired, but they are not necessary.

1.4. STARTING TURBO PASCAL

When you have a copy of the system on your work disk, enter the command **TURBO** at your terminal. The system will log on with the following message:

```
TURBO Pascal system        Version N.NNX
                                        [System]

Copyright (c) 1983, 1984 by BORLAND Inc.

No terminal selected

Include error messages (Y/N)? ■
```

Figure 1-1: Log-on Message

N.NNX specifies your release number and [System] indicates the operating environment (operating system and CPU), for example **CP/M-80, Z-80**. The second line from last tells you which screen is installed (at the moment, none -- but more about that later).

If you enter a **Y** in response to the error message question, the error-message file will be read into memory (if it is on the disk), briefly displaying the message **Loading TURBO.MSG**. You may instead answer **N** and save about 1.5 Kbytes of memory. Then the TURBO main menu will appear:

```
Logged drive: A

Work file:
Main file:

Edit    Compile  Run    Save
Dir     Quit     compiler Options

Text:       0 bytes
Free:   62903 bytes
```

Figure 1-2: Main Menu

The menu shows you the commands available, each of which will be described in following sections. Each command is executed by entering the associated capital letter (highlighted after terminal installation, if your terminal has that feature). Don't press <RETURN>; the command executes immediately. The values above for Logged drive and memory use are for the sake of example only; the values shown will be the actual values for your computer.

You may use TURBO without installation if you don't plan to use the built-in editor. If you do, type **Q** now to leave TURBO for a minute to perform the installation.

1.5. INSTALLATION

Type **TINST** to start the installation program. All TINST files and the TURBO.COM file must be on the logged drive. This menu will appear:

```
                TURBO Pascal installation menu.
            Choose installation item from the following:

    [S]creen installation  |  [C]ommand installation  |  [Q]uit

                        Enter S, C, or Q:
```

Figure 1-3: Installation Main Menu

Now hit **S** to select Screen installation. A menu containing the names of the most-used terminals will appear, and you may choose the one that suits you by entering the appropriate number. If your terminal is not on the menu, nor compatible with any of these (note: a lot of terminals are compatible with an ADM-3A), then you must perform the installation yourself. This is quite straightforward, but you will need to consult the manual that came with your terminal to answer the questions asked by the installation menu. See Appendix L for details.

When you have chosen a terminal, you are asked if you want to modify it before installation. This can be used if you have, for example, an ADM-3A-compatible terminal with some additional features. Choose the ADM-3A and add the required commands to activate the special features. If you answer Yes, you will be taken through a series of questions as described in Appendix L.

Normally, you will answer **N**o to this question, which means that you are satisfied with the pre-defined terminal installation. Now you will be asked the operating frequency of your microprocessor. Enter the appropriate value (2, 4, 6, or 8, most probably 4).

After that, the main menu re-appears, and you may now continue with the **C**ommand installation described in the next section, or you may terminate the installation at this point by entering **Q** for Quit.

1.5.1. INSTALLATION OF EDITING COMMANDS

The built-in editor responds to a number of commands which are used to move the cursor around on the screen, delete and insert text, move text, etc. Each of these functions may be activated by either a primary of a secondary command. The secondary commands are installed by Borland, and comply with the standard set by WordStar. The primary commands are undefined for most systems, and may be defined easily to suit your taste or your keyboard, using the installation program.

Please turn to Appendix L for a full description of the editor command installation.

1.6. THE MENU

After installation, you activate TURBO Pascal again by typing the command **TURBO**. Your screen should now clear and display the menu, this time with the command letters highlighted. If not, check your installation data.

```
Logged drive:  A

Work file:
Main file:

Edit    Compile  Run    Save
Dir     Quit     compiler Options

Text:       0 bytes
Free:   62903 bytes

> ■
```

Figure 1-4: Main Menu

By the way, whenever highlighting is mentioned here, it is assumed that your screen has different video attributes to show text in different intensities, reversed, underlined, or some other way. If not, just disregard any mention of highlighting.

This menu shows you the commands available to you while working with TURBO Pascal. A command is activated by pressing the associated upper case (highlighted) letter. Don't press <RETURN>, the command is executed immediately. The menu may very well disappear from the screen when working with the system; it is easily restored by entering an "illegal command", i.e., any key that does not activate a command. <RETURN> or <SPACE> will do perfectly.

The following sections describe each command in detail.

1.6.1. LOGGED DRIVE SELECTION

The **L** command is used to change the currently logged drive. When you press **L**, the prompt

```
New drive: ■
```

invites you to enter a new drive name, that is, a letter from A through P, optionally followed by a colon and terminated with <RETURN>. If you don't want to change the current value, just hit <RETURN>. The **L** command performs a disk reset, even when you don't change the drive, and should therefore be used whenever you change disks, to avoid a fatal disk-write error.

The new drive is not immediately shown on the menu, as it is not automatically updated. Hit for example <SPACE> to display a fresh menu, which will show the new logged drive.

1.6.2. WORK FILE SELECTION

The **W** command is used to select a work file, which is the file to be used to Edit, Compile, Run, eXecute, and Save. The **W** command will display this prompt:

Work file name: ■

and you may respond with any legal file name (a name of one through eight characters, an optional period, and an optional file type of no more than three characters, for instance **FILENAME.TYP**).

If you enter a file name without period and file type, the file type **PAS** is automatically assumed and appended to the name. You may explicitly specify a file name with no file type by entering a period after the name, but omitting the type.

Examples:

PROGRAM	becomes PROGRAM.PAS
PROGRAM.	is not changed
PROGRAM.FIL	is not changed

File types .BAK, .CHN, and .COM should be avoided, as TURBO uses these names for special purposes.

When the work file has been specified, the file is read from disk, if present. If the files does not already exist, the message **New File** is displayed. If you have edited another file which you have not saved, the message

Workfile X:FILENAME.TYP not saved. Save (Y/N)? ■

warns you that you are about to load a new file into memory and write over the one you have just worked on. Answer **Y** to save, or **N** to skip.

The new work file name will show on the menu the next time it is updated, like when you hit <SPACE>.

1.6.3. MAIN FILE SELECTION

The **M** command may be used to define a main file when working with programs which use the compiler directive $I to include a file. The main file should be the file which contains the include directives. You can then define the work file to be different from the main file, and thus edit different include files while leaving the name of the main file unchanged.

When a compilation is started, and the work file is different from the main file, the current work file is automatically saved, and the main file is loaded into memory. If an error is found during compilation, the file containing the error (whether it is the main file or an include file) automatically becomes the work file, which may then be edited. When the error has been corrected, and compilation is started again, the corrected work file is automatically saved, and the main file is reloaded.

The main file name is specified as described for the work file name in the previous section.

1.6.4. EDIT COMMAND

The **E** command is used to invoke the built-in editor and edit the file defined as the work file. If no work file is specified, you are first asked to specify one. The menu disappears, and the editor is activated. More about the use of the editor starting on page 12.

While you may use the TURBO system to compile and run programs without installing a terminal, the use of the editor requires that your terminal be installed. See page 6.

1.6.5. COMPILE COMMAND

The **C** command is used to activate the compiler. If no main file is specified, the work file will be compiled, otherwise the main file will be compiled. In the latter case, if the work file has been edited, you will be asked whether to save it before the main file is loaded and compiled. The compilation may be interrupted at any moment by pressing a key.

The compilation may result either in a program residing in memory, in a .COM file, or in a .CHN file. The choice is made on the compiler Options menu described on page 143. The default is to have the program residing in memory.

1.6.6. RUN COMMAND

The **R** command is used to activate a program residing in memory or, if the **C** switch on the compiler **O**ptions menu is active, a TURBO object code file (.COM file). If a compiled program is already in memory, it will be activated. If not, a compilation will automatically take place as described above.

1.6.7. SAVE COMMAND

The **S** command is used to save the current work file on disk. The old version of this file, if any, will be renamed to .BAK, and the new version will be saved.

1.6.8. DIRECTORY COMMAND

The **D** command gives you a directory listing and information about remaining space on the logged drive. When hitting **D**, you are prompted thus:

```
Dir mask: ■
```

You may enter a drive designator, or a drive designator followed by a file name or a mask containing the usual wildcards * and ?. Or you may just hit <RETURN> to get a full directory listing of the logged drive.

1.6.9. QUIT COMMAND

The **Q**uit command is used to leave the TURBO system. If the work file has been edited since it was loaded, you areasked whether you want to save it before quitting.

1.6.10. COMPILER OPTIONS

The **O** command selects a menu on which you may view and change some default values of the compiler. It also provides a helpful function to find run-time errors in programs compiled into object code files.

As these options vary between implementations, further discussion is deferred to Chapter 22.

1.7. THE TURBO EDITOR

The built-in editor is a full-screen editor specifically designed for the creation of program source text. If you are familiar with MicroPro's WordStar, you need but little instruction in the use of the TURBO editor, as all editor commands are exactly like the ones you know from WordStar. There are a few minor differences, and the TURBO editor has a few extensions; these are discussed on page 28. You may install your own commands "on top" of the WordStar commands, as described on page 7. The WordStar commands, however, may still be used.

Using the TURBO editor is simple as can be. When you have defined a work file and hit **E**, the menu disappears, and the editor is activated. If the work file exists on the logged drive, it is loaded and the first page of text is displayed. If it is a new file, the screen is blank apart from the status line at the top.

You leave the editor and return to the menu by pressing **<CTRL>K D**; more about that later.

Text is entered on the keyboard just as if you were using a typewriter. To terminate a line, press the <RETURN> key (or CR or ENTER or whatever it is called on your keyboard). When you have entered enough lines to fill the screen, the top line will scroll off the screen. Don't worry, it isn't lost. You may page back and forth in your text with the editing commands described later.

Let us first take a look at the meaning of the **status line** at the top of the screen.

1.7.1. THE STATUS LINE

The top line on the screen is the status line containing the following information:

Line n	Col n	Insert	Indent	X:FILENAME.TYP

Figure 1-5: Editor Status Line

Line n
: Shows the number of the line containing the cursor, counted from the start of the file.

Col n
: Shows the number of the column containing the cursor, counted from the beginning of the line.

Insert
: Indicates that characters entered on the keyboard will be inserted at the cursor position. Existing text to the right of the cursor will move to the right as you write new text. Using the *insert mode on/off* command (**<CTRL>V** by default) will instead display the text **Overwrite**. Text entered on the keyboard will then write over characters under the cursor, instead of being inserted before them.

Indent
: Indicates that auto-indent is in effect. It may be switched off using the *auto-indent on/off* command (**<CTRL>Q I** by default).

X:FILENAME.TYP
: The drive, name, and type of the file being edited.

1.7.2. EDITING COMMANDS

As mentioned before, you use the editor almost as a typewriter, but as this is a computerized text editor, it offers you a number of editing facilities which make text manipulation, and in this case specifically program writing, much easier than on paper.

The TURBO editor accepts a total of 45 editing commands to move the cursor around, page through the text, find and replace text strings, etc. These commands can be grouped into four categories, each of which contains logically related commands which will be described separately in following sections. The following table provides an overview of the commands available:

CURSOR MOVEMENT COMMANDS:

Character left	Scroll down	To end of file
Character right	Page up	To left on line
Word left	Page down	To right on line
Word right	To top of screen	To beginning of block
Line up	To bottom of screen	To end of block
Line down	To top of file	To last cursor position
Scroll up		

INSERT AND DELETE COMMANDS:

Insert mode on/off	Delete to end of line	Delete character under cursor
Insert line		
Delete line	Delete right word	Delete left character

BLOCK COMMANDS:

Mark block begin	Copy block	Read block from disk
Mark block end	Move block	Write block to disk
Mark single word	Delete block	Hide/display block

MISC. EDITING COMMANDS:

End edit	Restore line	Repeat last find
Tab	Find and replace	Control character prefix
Auto tab on/off		

Table 1-1: Editing Command Overview

In a case like this, the best way of learning is by doing, so start TURBO, specify one of the demo Pascal programs as your work file, and enter E to start editing. Then use the commands as you read on.

Hang on, even if you find it a bit hard in the beginning. It is not just by chance that we have chosen to make the TURBO editor WordStar compatible. The logic of these commands, once learned, quickly becomes so much a part of you that the editor virtually turns into an extension of your mind. Take it from one who has written megabytes worth of text with that editor.

Each of the following descriptions consists of a heading defining the command, followed by the default keystrokes used to activate the command, with room in between to note which keys to use on your terminal, if you use other keys. If you have arrow keys and dedicated word processing keys (<INSERT>, <DELETE>, etc.) it might be convenient to use these. Please refer to pages 7 pp. for installation details.

The following descriptions of the commands assume the use of the default WordStar-compatible keystrokes.

1.7.3. A NOTE ON CONTROL CHARACTERS

All commands are issued using control characters. A control character is a special character generated by your keyboard when you hold down the <CONTROL> (or <CTRL>) key on your keyboard and press any key from A through Z, [, \,], or ^.

The <CONTROL> keys works like the <SHIFT> key. If you hold down the <SHIFT> key and press A, you get a capital A; if you hold down the <CONTROL> key and press A, you will get a Control-A (Ctrl-A for short).

1.7.4. BEFORE YOU START: HOW TO GET OUT

The command which takes you out of the editor is described on page 28, but you may find it useful to know now that the **Ctrl-K D** command (hold down the <CONTROL> key and press K, then release the <CONTROL> key and press D) exits the editor and returns you to the menu. This command does not automatically save the file; that must be done with the **Save** command from the menu.

1.7.5. BASIC MOVEMENT COMMANDS

The most basic thing to learn about an editor is how to move the cursor around on the screen. The TURBO editor uses a special group of control characters to do that, namely the control characters **A**, **S**, **D**, **F**, **E**, **R**, **X**, and **C**.

Why these? Because they are conveniently located close to the control key, so that your left little finger can rest on that while you use the middle and index fingers to activate the commands. Furthermore, the characters are arranged in such a way on the keyboard as to logically indicate their use. Let's examine the basic movements: cursor up, down, left, and right.

```
  E
S D
  X
```

These four characters are placed so that it is logical to assume that **Ctrl-E** moves the cursor up, **Ctrl-X** down, **Ctrl-S** to the left, and **Ctrl-D** to the right. And that is exactly what they do. Try to move the cursor around on the screen with these four commands. If your keyboard has repeating keys, you may just hold down the control key and one of these four keys, and the cursor will move rapidly across the screen.

Now let us look at some extensions of those movements.

```
  E R
A S D F
  X C
```

the location of the **Ctrl-R** next to the **Ctrl-E** suggests that **Ctrl-R** moves the cursor up, and so it does, only not one line at a time but a whole page. Similarly, **Ctrl-C** moves the cursor down one page at a time.

Likewise with **Ctrl-A** and **Ctrl-F**: **Ctrl-A** moves to the left like **Ctrl-S**, but a whole word at a time, while **Ctrl-F** moves one word to the right.

The two last basic movement commands do not move the cursor, but scroll the entire screen up or down in the file:

W E R
A S D F
Z X C

Ctrl-W scrolls upwards in the file (the lines on the screen move down), and **Ctrl-Z** scrolls downwards in the file (the lines on the screen move up).

Character left **Ctrl-S**

Moves the cursor one character to the left non-destructively, without affecting the character there. <BACKSPACE> may be installed to have the same effect. This command does not work across line breaks; when the cursor reaches the left edge of the screen, it stops.

Character right **Ctrl-D**

Moves the cursor one character to the right non-destructively, without affecting the character there. This command does not work across line breaks, i.e., when the cursor reaches the right end of the screen, the text starts scrolling horizontally until the cursor reaches the extreme right of the line, in column 128, where it stops.

Word left **Ctrl-A**

Moves the cursor to the beginning of the word to the left. A word is defined as a sequence of characters delimited by a space or one of the other following characters: < > , ; . ( ) [ ] ^ ' * + - / $. This command works across line breaks.

Word right **Ctrl-F**

Moves the cursor to the beginning of the word to the right. See the definition of a word, above. This command works across line breaks.

Line up **Ctrl-E**

Moves the cursor to the line above. If the cursor is on the top line, the screen scrolls down one line.

Line down **Ctrl-X**

Moves the cursor to the line below. If the cursor is on the second line from last, the screen scrolls up one line.

Scroll up **Ctrl-W**

Scrolls up towards the beginning of the file, one line at a time (the entire screen scrolls down). The cursor remains on its line until it reaches the bottom of the screen.

Scroll down **Ctrl-Z**

Scrolls down towards the end of the file, one line at a time (the entire screen scrolls up). The cursor remains on its line until it reaches the top of the screen.

Page up **Ctrl-R**

Moves the cursor one page up with an overlap of one line; the cursor moves one screenful, less one line, backwards in the text.

Page down **Ctrl-C**

Moves the cursor one page down with an overlap of one line; the cursor moves one screenful, less one line, forwards in the text.

1.7.6. EXTENDED MOVEMENT COMMANDS

The commands discussed above will let you move freely around in your program text, and they are easy to learn and understand. Try to use them all for a while and see how natural they feel.

Once you master them, you will probably sometimes want to move more rapidly. The TURBO editor provides six commands to move rapidly to the extreme ends of lines, to the beginning and end of the text, and to the last cursor position.

These commands require **two** characters to be entered; first a **Ctrl-Q**, then an **S**, **D**, **E**, **X**, **R**, or **C**. They repeat the pattern from before:

E R
S D
X C

Ctrl-Q S moves the cursor to the extreme left of the line, and **Ctrl-Q D** moves it to the extremely right of the line. **Ctrl-Q E** moves the cursor to the top of the screen, and **Ctrl-Q X** moves it to the bottom of the screen. **Ctrl-Q R** moves the cursor all the way up to the start of the file, while **Ctrl-Q C** moves it all the way down to the end of the file.

To left on line **Ctrl-Q S**

Moves the cursor all the way to the left edge of the screen, to column one.

To right on line **Ctrl-Q D**

Moves the cursor to the end of the line, to the position following the last printable character on the line. Trailing blanks are always removed from all lines to preserve space.

To top of screen **Ctrl-Q E**

Moves the cursor to the top of the screen.

To bottom of screen **Ctrl-Q X**

Moves the cursor to the bottom of the screen.

To top of file **Ctrl-Q R**

Moves to the first character of the text.

To end of file **Ctrl-Q C**

Moves to the last character of the text.

The **Ctrl-Q** prefix plus **B, K,** or **P** allows you to jump far within the file:

To beginning of block **Ctrl-Q B**

Moves the cursor to the position of the ***block begin*** marker set with **Ctrl-K B** (hence the logic of **Ctrl-Q B**). The command works even if the block is not displayed (see ***hide/display block*** later), or the ***block end*** marker is not set.

To end of block **Ctrl-Q K**

Moves the cursor to the position of the ***block end*** marker set with **Ctrl-K K** (hence the logic of **Ctrl-Q K**). The command works even if the block is not displayed (see ***hide/display block*** later), or the ***block begin*** marker is not set.

To last cursor position **Ctrl-Q P**

Moves to the last position of the cursor. This command is particularly useful to move back to the last position after a Save operation, or after a find or find and replace operation.

1.7.7. INSERT AND DELETE COMMANDS

These commands let you insert and delete characters, words, and lines. They can be divided into three groups: one command which controls the text entry mode (insert or overwrite), a number of simple commands, and one extended command.

Note: The TURBO editor provides a "regret" facility which lets you undo changes *as long as you have not left the line.* This command (**Ctrl-Q L**) is described on page 28.

Insert mode on/off **Ctrl-V**

When you enter text, you may choose between two entry modes: *Insert* and *Overwrite.* Insert mode is the default value when the editor is invoked, and it lets you insert new text into an existing text. The existing text to the right of the cursor simply moves to the right while you enter the new text.

Overwrite mode may be chosen if you wish to replace old text with new text. Characters entered then replace existing characters under the cursor.

You switch between these modes with the *insert mode on/off* command **Ctrl-V**, and the current mode is displayed in the status line at the top of the screen.

Delete left character **<DEL>**

Moves one character to the left and deletes the character there. Any characters to the right of the cursor move one position to the left. The <BACKSPACE> key, which normally backspaces non-destructively like **Ctrl-S**, may be installed to perform this function if it is more conveniently located on your keyboard, or if your keyboard lacks a <DELETE> key (sometimes labeled <DEL>, <RUBOUT>, or <RUB>). This command works across line breaks, and can be used to remove line breaks.

Delete character under cursor **Ctrl-G**

Deletes the character under the cursor, and moves any characters to the right of the cursor to the left. This command does not work across line breaks.

Delete right word **Ctrl-T**

Deletes the word to the right of the cursor. A word is defined as a sequence of characters delimited by the SPACE character, or by one of < > , ; . ( ) [ ] ^ ' * + - / $. This command works across line breaks, and may be used to remove line breaks.

Insert line **Ctrl-N**

Inserts a line break at the cursor position. The cursor does not move.

Delete line **Ctrl-Y**

Deletes the line containing the cursor and moves any lines below it one line up. The cursor moves to the left edge of the screen. No provision exists to restore a deleted line, so take care!

Delete to end of line **Ctrl-Q Y**

Deletes all text from the cursor position to the end of the line.

1.7.8. BLOCK COMMANDS

All block commands are extended commands (two characters each in the standard command definition), and you may ignore them at first if you feel a bit dazzled at this point. Later on, when you feel the need to move, delete, or copy whole chunks of text, you should return to this section.

For the persevering, we'll go on and discuss the use of blocks.

A block of text is simply any amount of text, from a single character to several pages of text. A block is marked by placing a *Begin block* marker at the first character and an *End block* marker at the last character of the desired portion of the text. Thus marked, the block may be copied, moved, deleted, or written to a file. A command is available to read an external file into the text as a block, and a special command conveniently marks a single word as a block.

Mark block begin **Ctrl-K B**

This command marks the beginning of a block. The marker itself is not visible on the screen, and the block only becomes visibly marked when the *End block* marker is set, and then only if the screen is installed to show some sort of highlighting. But even if the block is not visibly marked, it is internally marked and may be manipulated.

Mark block end **Ctrl-K K**

This command marks the end of a block. As above, the marker itself is not visible on the screen, and the block only becomes visibly marked when the *Begin block* marker is also set.

Mark single word **Ctrl-K T**

This command marks a single word as a block, and thus replaces the *Begin block -- End block* sequence which is a bit clumsy when marking just one word. If the cursor is placed within a word, then this word will be marked; if not, then the word to the left of the cursor will be marked. A word is defined as a sequence of characters delimited by either a space or one of < > , ; . ( ) ^ ' * + - / or $.

Hide/display block **Ctrl-K H**

This command causes the visual marking of a block (dim text) to be alternatively switched off and on. Block manipulation commands (copy, move, delete, and write to a file) work only when the block is displayed. Cursor movements for blocks (jump to beginning/end of block) work whether the block is hidden or displayed.

Copy block **Ctrl-K C**

This command places a copy of a previously marked block starting at the cursor position. The original block is left unchanged, and the markers are placed around the new copy of the block. If no block is marked, the command performs no operation, and no error message is issued.

Move block **Ctrl-K V**

This command moves a previously marked block from its original position to the cursor position. The block disappears from its original position and the markers remain around the block at its

new position. If no block is marked, the command performs no operation, and no error message is issued.

Delete block **Ctrl-K Y**

This command deletes the previously marked block. No provision exists to restore a deleted block, so be careful!

Read block from disk **Ctrl-K R**

This command is used to read a file into the current text at the cursor position, exactly as if it were a block that was moved or copied. The block read in is marked as a block. When this command is issued, you are prompted for the name of the file to read. The file specified may be any legal file name. If no file type is specified, .PAS is automatically assumed. A file without type is specified as a name followed by a period.

Write block to disk **Ctrl-K W**

This command is used to write a previously marked block to a file. The block is left unchanged, and the markers remain in place. When this command is issued, you are prompted for the name of the file to write to. If the file specified already exists, a warning is issued before the existing file is written over. If no block is marked, the command performs no operation, and no error message is issued. The file specified may be any legal file name. If no file type is specified, .PAS is automatically assumed. A file name without a file type is specified as a name followed by a period. Avoid the use of file types .BAK, .CHN, and .COM, as they are used for special purposes by the TURBO system.

1.7.9. MISCELLANEOUS EDITING COMMANDS

This section collects a number of commands which do not logically fall into any of the above categories. They are nonetheless important, especially this first one:

End edit **Ctrl-K D**

This command ends the edit and returns to the main menu. The editing has been performed entirely in memory, and any associated disk file is not affected. Saving the edited file on disk is done explicitly with the Save command from the main menu, or automatically, in connection with a compilation or definition of a new work file.

Tab **<TAB> or Ctrl-I**

There are no fixed tab positions in the TURBO editor. Instead, tab positions are automatically set to the beginning of each word on the line immediately above the cursor. This provides a very convenient automatic tabbing feature especially useful in program editing, where you often want to line up columns of related items, such as variable declarations. Remember that Pascal allows you to write extremely beautiful source texts. Do it, not for the sake of the purists, but more importantly to keep the program easy to understand, especially when you return to make changes after some time.

Auto indent on/off **Ctrl-Q I**

The auto-indent feature provides automatic indenting of successive lines. When active, the indent of the current line is repeated on each following line. That is, when you hit <RETURN>, the cursor does not return to column one, but to the starting column of the line you just terminated. When you want to change the indent, use any of the cursor right or left commands to select the new column. When auto indent is active, the message **Indent** is displayed in the status line, and when passive, the message is removed. Auto indent is active by default.

Restore line **Ctrl-Q L**

This command lets you regret changes made to a line ***as long as you have not left the line.*** The line is simply restored to its original contents, regardless of what changes you have made, but only as long as you remain on the line; the moment you leave it, changes are there to stay. For this reason, the *Delete line* (**Ctrl-Y**) command can only be regretted, not restored. Some days you may find yourself continuously falling asleep on the Ctrl-Y key, with vast consequences. A good long break usually helps.

Find **Ctrl-Q F**

The *Find* command lets you search for any string of up to 30 characters. When you enter this command, the status line is cleared, and you are prompted for a search string. Enter the string you are looking for and terminated with <RETURN>. The search string may contain any characters, even control

characters. Control characters are entered into the search string with the **Ctrl-P** prefix. For example, enter a **Ctrl-A** by holding down the <CONTROL> key while pressing first P, then A. You may thus include a line break in a search string by specifying **Ctrl-M Ctrl-J**. Notice that **Ctrl-A** has a special meaning: it matches any character, and may be used as a wildcard in search strings.

Search strings may be edited with the *Character Left, Character Right, Word Left,* and *Word Right* commands. *Word Right* recalls the previous search string, which may then be edited. The search operation may be aborted with the *Abort* command (**Ctrl-U**).

When the search string is specified, you are asked for search options. The following options are available:

B Search backwards. Search from the current cursor position towards the *beginning* of the text.

G Global search. Search the entire text, regardless of the current cursor position.

n n = any number. Find the nth occurrence of the search string, counted from the current cursor position.

U Ignore upper/lower case. Regard upper and lower case alphabeticals as identical.

W Search for whole words only. Skip matching patterns which are embedded in other words.

Examples:

W search for whole words only. The search string "term" will only match the word "term", not the string "term" in the word "terminal".

BU search backwards and ignore upper/lower case distinctions. "Block" will match "blockhead", "BLOCKADE", etc.

125 Find the 125th occurrence of the search string.

Terminate the list of options (if any) with <RETURN>, and the search starts. If the text contains a target matching the search string, the cursor is positioned at the end of the target. The search operation may be repeated by the *Repeat last find* command (**Ctrl-L**).

Find and replace **Ctrl-Q A**

The *Find and replace* command lets you search for any string of up to 30 characters and replace it with any other string of up to 30 characters. When you enter this command, the status line is cleared, and you are prompted for a search string. Enter the string you are looking for and hit <RETURN>. The search string may contain any characters, even control characters. Control characters are entered into the search string with the **Ctrl-P** prefix. For example, enter a **Ctrl-A** by holding down the <CONTROL> key while pressing first P, then A. You may thus include a line break in a search string by specifying **Ctrl-M Ctrl-J**. Notice that **Ctrl-A** has a special meaning: it matches any character, and may be used as a wildcard in search strings.

Search strings may be edited with the *Character Left, Character Right, Word Left,* and *Word Right* commands. *Word Right* recalls the previous search string, which may then be edited. The search operation may be aborted with the *Abort* command (**Ctrl-U**).

When the search string is specified, you are asked to enter the string to replace the search string. Enter up to 30 characters; entering control characters and editing is performed as above, but **Ctrl-A** has no special meaning in the replace string. If you just press <RETURN>, the target will be replaced with nothing, in effect deleted.

Finally you are prompted for options. The search and replace options are:

B Search and replace backwards. Search and replace from the current cursor position towards the *beginning* of the text.

G Global search and replace. Search and replace in the entire text, regardless of the current cursor position.

n n = any number. Find and replace *n* occurrences of the search string, counted from the current cursor position.

N Replace without asking. Do not stop and ask *Replace (Y/N)* for each occurrence of the search string.

U Ignore upper/lower case. Regard upper and lower case alphabeticals as identical.

W Search for whole words only. Skip matching patterns which are embedded in other words.

Examples:

N10 Find the next ten occurences of the search string and replace without asking.

GW Find and replace whole words in the entire text. Ignore upper/lower case distinctions.

End the list of options, if any, with <RETURN>, and the search and replace starts. Depending on the options specified, the string may be found. When found, and if the **N** option is not specified, the cursor is positioned at the end of the target, and you are asked the question *Replace (Y/N)?* on the prompt line at the top of the screen. You may abort the search and replace operation at this point with the *Abort* command (**Ctrl-U**). The search and replace operation may be repeated by the *Repeat last find* command (**Ctrl-L**).

Repeat last find **Ctrl-L**

This command repeats the last *Find* or *Find and replace* operation exactly as if all information had been re-entered.

Control character prefix **Ctrl-P**

The TURBO editor allows you to enter control characters into the file by prefixing the desired control character with a **Ctrl-P**. That is, first press **Ctrl-P**, then press the desired control character. Control characters will appear as half-intensity capital letters on the screen (or reverse video, depending on your terminal).

Abort operation **Ctrl-U**

The **Ctrl-U** command lets you abort any command in process whenever it pauses for input, as when *Search and replace* asks *Replace (Y/N)?*, or during entry of a search string or a file name.

1.8. THE TURBO EDITOR VS. WORDSTAR

Someone used to WordStar will notice that a few TURBO commands work slightly differently. Also, although TURBO contains only a subset of WordStar's commands, a number of special features not found in WordStar have been added to enhance the editing of program source code. These differences are:

1.8.1. CURSOR MOVEMENT

The cursor movement controls **Ctrl-S**, **D**, **E**, and **X** move freely around on the screen and do not jump to column one on empty lines. This does not mean that the screen is full of blanks; on the contrary, all trailing blanks are automatically deleted. This way of moving the cursor is especially useful, for example, when matching indented **begin - end** pairs.

Ctrl-S and **Ctrl-D** do not work across line breaks. To move from one line to another you must use **Ctrl-E**, **Ctrl-X**, **Ctrl-A**, or **Ctrl-F**.

1.8.2. MARK SINGLE WORD

Ctrl-K T is used to mark a single word as a block, which is more convenient than the two-step process of marking the beginning and the end of the word separately.

1.8.3. END EDIT

The **Ctrl-K D** command ends editing and returns you to the menu. As editing in TURBO is done entirely in memory, this command does not change the file on disk (as it does in WordStar). Updating the disk file must be done explicitly with the **Save** command from the main menu, or automatically in connection with a compilation or definition of a new work file. TURBO's **Ctrl-K D** does not resemble WordStar's **Ctrl-K Q** (quit edit) command either, as the changed text is not abandoned; it is left in memory, ready to be **Compiled** or **Saved**.

1.8.4. LINE RESTORE

The **Ctrl-Q L** command restores a line to its contents before edit *as long as the cursor has not left the line.*

1.8.5. TABULATOR

No fixed tab settings are provided. Instead, the automatic tab feature sets tabs to the start of each word on the line immediately above the cursor.

1.8.6. AUTO INDENTATION

The **Ctrl-Q I** command switches the auto indent feature on and off.

Chapter 2
BASIC LANGUAGE ELEMENTS

2.1. BASIC SYMBOLS

The basic vocabulary of TURBO Pascal consists of basic symbols divided into letters, digits, and special symbols:

Letters
A to Z, a to z, and _ (underscore)
Digits
0 1 2 3 4 5 6 7 8 9
Special symbols
+ - * / = ^ > < ( ) [ ] { } . , : ; ' # $

No distinction is made between upper and lower case letters. Certain operators and delimiters are formed using two special symbols:

Assignment operator: :=
Relational operators: <> <= >=
Subrange delimiter: ..
Brackets: (. and .) may be used instead of [and]
Comments: (* and *) may be used instead of { and }

2.2. RESERVED WORDS

Reserved words are integral parts of TURBO Pascal. They cannot be redefined, and therefore must not be used as user-defined identifiers.

*	**absolute**	*	**external**		**nil**	*	**shl**
	and		**file**		**not**	*	**shr**
	array		**forward**	*	**overlay**	*	**string**
	begin		**for**		**of**		**then**
	case		**function**		**or**		**type**
	const		**goto**		**packed**		**to**
	div	*	**inline**		**procedure**		**until**
	do		**if**		**program**		**var**
	downto		**in**		**record**		**while**
	else		**label**		**repeat**		**with**
	end		**mod**		**set**	*	**xor**

Throughout this manual, reserved words are written in **boldface**. The asterisks indicate reserved words not defined in standard Pascal.

2.3. STANDARD IDENTIFIERS

TURBO Pascal defines a number standard identifiers of predefined types, constants, variables, procedures, and functions. Any of these identifiers may be redefined, but it will mean the loss of the facility offered by that particular identifier, and may lead to confusion. The following standard identifiers are therefore best left to their special purposes:

Addr	*Delay*	*Length*	*Release*
ArcTan	*Delete*	*Ln*	*Rename*
Assign	*EOF*	*Lo*	*Reset*
Aux	*EOLN*	*LowVideo*	*Rewrite*
AuxInPtr	*Erase*	*Lst*	*Round*
AuxOutPtr	*Execute*	*LstOutPtr*	*Seek*
BlockRead	*Exit*	*Mark*	*Sin*
BlockWrite	*Exp*	*MaxInt*	*SizeOf*
Boolean	*False*	*Mem*	*SeekEof*
BufLen	*FilePos*	*MemAvail*	*SeekEoln*
Byte	*FileSize*	*Move*	*Sqr*
Chain	*FillChar*	*New*	*Sqrt*
Char	*Flush*	*NormVideo*	*Str*
Chr	*Frac*	*Odd*	*Succ*
Close	*GetMem*	*Ord*	*Swap*
ClrEOL	*GotoXY*	*Output*	*Text*
ClrScr	*Halt*	*Pi*	*Trm*
Con	*HeapPtr*	*Port*	*True*
ConInPtr	*Hi*	*Pos*	*Trunc*
ConOutPtr	*IOresult*	*Pred*	*UpCase*
Concat	*Input*	*Ptr*	*Usr*
ConstPtr	*InsLine*	*Random*	*UsrInPtr*
Copy	*Insert*	*Randomize*	*UsrOutPtr*
Cos	*Int*	*Read*	*Val*
CrtExit	*Integer*	*ReadLn*	*Write*
CrtInit	*Kbd*	*Real*	*WriteLn*
DelLine	*KeyPressed*		

Each TURBO Pascal implementation further contains a number of dedicated standard identifiers. The dedicated standard identifiers for CP/M-80 are listed in Chapter 22.

Throughout this manual, all identifiers, including standard identifiers, are written in a combination of upper and lower case letters (see page 35). In the text (as opposed to program examples), they are furthermore *in italics.*

2.4. DELIMITERS

Language elements must be separated by at least one of the following delimiters: a blank, an end of line, or a comment.

2.5. PROGRAM LINES

The maximum length of a program line is 127 characters; any character beyond the 127th is ignored by the compiler. For this reason the TURBO editor allows only 127 characters on a line, but source code prepared with other editors may use longer lines. If such a text is read into the TURBO editor, line breaks will be inserted automatically, and a warning will be issued.

Chapter 3
STANDARD SCALAR TYPES

A data type defines the set of values a variable may assume. Every variable in a program must be associated with one and only one data type. Although data types in TURBO Pascal can be quite sophisticated, they are all built from simple (unstructured) types.

A simple type may either be defined by the programmer (it is then called a *declared scalar type*), or be one of the *standard scalar types*: **integer, real, boolean, char,** or **byte**. The following is a description of these five standard scalar types.

3.1. INTEGER

Integers are whole numbers. In TURBO Pascal, they are limited to a range of from -32768 to 32767. Integers occupy two bytes in memory.

Overflow of integer arithmetic operations is not detected. Notice in particular that partial results in integer expressions must be kept within the integer range. For instance, the expression 1000 * 100/50 will not yield 2000, as the multiplication causes an overflow.

3.2. BYTE

The type *Byte* is a subrange of the type *Integer*, of the range 0..255. Bytes are therefore compatible with integers. Whenever a *Byte* value is expected, an *Integer* value may be specified instead and vice versa, **except** when passed as parameters. Furthermore, *Bytes* and *Integers* may be mixed in expressions and *Byte* variables may be assigned integer values. A variable of type *Byte* occupies one byte in memory.

3.3. REAL

The range of real numbers is 1E-38 through 1E+38 with a mantissa of up to 11 significant digits. Reals occupy 6 bytes in memory.

Overflow during an arithmetic operation involving reals causes the program to halt, displaying an execution error. An underflow will cause a result of zero.

Although the type *Real* is included here as a standard scalar type, the following differences between reals and other scalar types should be noticed:

1) The functions *Pred* and *Succ* cannot take real arguments.
2) Reals cannot be used in array indexing.
3) Reals cannot be used to define the base type of a set.
4) Reals cannot be used in controlling **for** and **case** statements.
5) Subranges of reals are not allowed.

3.4. BOOLEAN

A boolean value can assume either of the logical truth values denoted by the standard identifiers *True* and *False*. These are defined such that *False* < *True*. A *Boolean* variable occupies one byte in memory.

3.5. CHAR

A *Char* value is one character in the ASCII character set. Characters are ordered according to their ASCII value, for example: 'A' < 'B'. The ordinal (ASCII) values of characters range from 0 to 255. A *Char* variable occupies one byte in memory.

Chapter 4
USER-DEFINED LANGUAGE ELEMENTS

4.1. IDENTIFIERS

Identifiers are used to denote labels, constants, types, variables, procedures, and functions. An identifier consists of a letter or underscore followed by any combination of letters, digits, or underscores. An identifier is limited in length only by the line length of 127 characters, and all characters are significant.

Examples:

TURBO	
square	
persons_counted	
BirthDate	
3rdRoot	illegal, starts with a digit
Two Words	illegal, must not contain a space

As TURBO Pascal does not distinguish between upper and lower case letters, the use of mixed upper and lower case as in *BirthDate* has no functional meaning. It is nevertheless encouraged, as it leads to more legible identifiers. *VeryLongIdentifier* is easier to read for the human reader than *VERYLONGIDENTIFIER*. This mixed mode will be used for all identifiers throughout this manual.

4.2. NUMBERS

Numbers are constants of integer type or of real type. Integer constants are whole numbers expressed in either decimal or hexadecimal notation. Hexadecimal constants are identified by being preceded by a dollar sign: $ABC is a hexadecimal constant. The decimal integer range is -32768 through 32767 and the hexadecimal integer range is $0000 through $FFFF.

Examples:

1
12345
-1
$123
$ABC

$123G	illegal, G is not a legal hexadecimal digit
1.2345	illegal as an integer, contains a decimal part

The range of *Real* numbers is 1E-38 through 1E+38 with a mantissa of up to 11 significant digits. Exponential notation may be used, with the letter E preceding the scale factor meaning "times ten to the power of". An integer constant is allowed anywhere a real constant is allowed. Separators are not allowed within numbers.

Examples:

1.0	
1234.5678	
-0.012	
1E6	
2E-5	
-1.2345678901E+12	
1	legal, but it is not a real, it is an integer

4.3. STRINGS

A string constant is a sequence of characters enclosed in single quotes:

'This is a string constant '

A single quote may be contained in a string by writing two successive single quotes. Strings containing only a single character are of the standard type *Char*. A string is compatible with an **array of** *Char* of the same length. All string constants are compatible with all **string** types.

Examples:

'TURBO'
'You''ll see'
''''
';'
''

As shown in examples 2 and 3, a single quote within a string is written as two consecutive quotes. The four consecutive single quotes in example 3 thus constitute a string containing *one* quote.

The last example – the quotes enclosing no characters, denoting an *empty string* – is compatible only with **string** types.

4.3.1. CONTROL CHARACTERS

TURBO Pascal also allows control characters to be embedded in strings. Two notations for control characters are supported:

1) The # symbol followed by an integer constant in the range 0..255 denotes a character of the corresponding ASCII value, and
2) the ^ symbol followed by a character, denotes the corresponding control character.

Examples:

#10	ASCII 10 decimal (Line Feed).
#$1B	ASCII 1B hex (Escape).
^g	Control-G (Bell).
^L	Control-L (Form Feed).
^[	Control-[(Escape).

Sequences of control characters may be concatenated into strings by writing them *without separators* between the individual characters:

```
#13#10
#27^U#20
^G^G^G^G
```

The above strings contain two, three, and four characters, respectively. Control characters may also be mixed with text strings:

```
'Waiting for input! '^G^G^G' Please wake up'
#27'U '
'This is another line of text '^M^J
```

These three strings contain 37, 3, and 31 characters, respectively.

4.4. COMMENTS

A comment may be inserted anywhere in the program where a delimiter is legal. It is delimited by the curly braces { and }, which may be replaced by the symbols (* and *).

Examples:

```
{This is a comment}
(* and so is this *)
```

Curly braces may not be nested within curly braces, and (*..*) may not be nested within (*..*). However, curly braces may be nested within (*..*) and vice versa, thus allowing entire sections of source code to be commented away, even if they contain comments.

4.5. COMPILER DIRECTIVES

A number of features of the TURBO Pascal compiler are controlled through compiler directives. A compiler directive is introduced as a comment with a special syntax, which means that whenever a comment is allowed in a program, a compiler directive is also allowed.

A compiler directive consists of an opening brace immediately followed by a dollar sign immediately followed by one compiler directive letter or a list of compiler directive letters separated by commas. The syntax of the directive or directive list depends upon the directive(s) selected. A full description of each compiler directive follows in the relevant sections, and a summary of compiler directives is located in Appendix C. File inclusion is discussed in Chapter 17.

Examples:

```
{$I-}
{$I INCLUDE.FIL}
{$R-,B+,V-}
(*$X-*)
```

Notice that no spaces are allowed before or after the dollar-sign.

Chapter 5
PROGRAM HEADING AND PROGRAM BLOCK

A Pascal program consists of a program heading followed by a program block. The program block is further divided into a declaration part, in which all objects local to the program are defined, and a statement part, which specifies the actions to be executed upon these objects. Each is described in detail in the following.

5.1. PROGRAM HEADING

In TURBO Pascal, the program heading is purely optional and of no significance to the program. If present, it gives the program a name, and optionally lists the parameters through which the program communicates with the environment. The list consists of a sequence of identifiers enclosed in parentheses and separated by commas.

Examples:
program Circles;
program Accountant (Input,Output);
program Writer (Input,Printer);

5.2. DECLARATION PART

The declaration part of a block declares all identifiers to be used within the statement part of that block (and possibly other blocks within it). The declaration part is divided into five different sections:

1) Label declaration part
2) Constant definition part
3) Type definition part
4) Variable declaration part
5) Procedure and function declaration part

Whereas standard Pascal specifies that each section may only occur zero or one time, and only in the above order, TURBO Pascal allows each of these sections to occur any number of times in any order in the declaration part.

5.2.1. LABEL DECLARATION PART

Any statement in a program may be prefixed with a **label**, enabling direct branching to that statement by a **goto** statement. A label consists of a label name followed by a colon. Before use, the label must be declared in a label declaration part. The reserved word **label** heads this part, and it is followed by a list of label identifiers separated by commas and terminated by a semi-colon.

Example:

```
label 10, error, 999, Quit;
```

Whereas standard Pascal limits labels to numbers of no more than 4 digits, TURBO Pascal allows both numbers and identifiers to be used as labels.

5.2.2. CONSTANT DEFINITION PART

The constant definition part introduces identifiers as synonyms for constant values. The reserved word **const** heads the constant definition part, and is followed by a list of constant assignments separated by semi-colons. Each constant assignment consists of an identifier followed by an equal sign and a constant. Constants are either strings or numbers, as defined on pages 35 and 36.

Example:

```
const
  Limit = 255;
  Max = 1024;
  PassWord = 'SESAM';
  CursHome = ^'V';
```

The following constants are predefined in TURBO Pascal, and may be "referenced" (referred to or used) without previous definition:

Name	**Type and value:**
Pi	Real (3.1415926536E+00).
False	Boolean (the truth value false).
True	Boolean (the truth value true).
Maxint	Integer (32767).

As described in Chapter 13, a constant definition part may also define typed constants.

5.2.3. TYPE DEFINITION PART

A data type in Pascal may be either directly described in the variable declaration part or referenced by a type identifier. Several standard type identifiers are provided, and the programmer may create his own types through the use of the type definition. The reserved word **type** heads the type definition part, and it is followed by one or more type assignments separated by semi-colons. Each type assignment consists of a type identifier followed by an equal sign and a type.

Example:

```
type
   Number = Integer;
   Day = (mon,tues,wed,thur,fri,sat,sun);
   List = array[1..10] of Real;
```

More examples of type definitions are found in subsequent sections.

5.2.4. VARIABLE DECLARATION PART

Every variable occurring in a program must be declared before use. The declaration must textually precede any use of the variable so that the variable is "known" to the compiler when it is used.

A variable declaration consists of the reserved word **var** followed by one or more identifier(s), separated by commas, each followed by a colon and a type. This creates a new variable of the specified type and associates it with the specified identifier.

The "scope" of this identifier is the block in which it is defined, and any block within that block. Note, however, that any such block within another block may define *another* variable using the *same* identifier. This variable is said to be *local* to the block in which it is declared (and any blocks within *that* block) and the variable declared on the outer level (the *global* variable) becomes inaccessible.

Example:

```
var
   Result, Intermediate, SubTotal: Real;
   I, J, X, Y: Integer;
   Accepted, Valid: Boolean;
   Period: Day;
   Buffer: array[0..127] of Byte;
```

5.2.5. PROCEDURE AND FUNCTION DECLARATION PART

A procedure declaration serves to define a procedure within the current procedure or program (see page 116). A procedure is activated from a procedure statement (see page 47), and upon completion, program execution continues with the statement immediately following the calling statement.

A function declaration serves to define a program part which computes and returns a value (see page 121). A function is activated when its designator is met as part of an expression (see page 46).

5.3. STATEMENT PART

The statement part is the last part of a block. It specifies the actions to be executed by the program. The statement part takes the form of a compound statement followed by a period or a semi-colon. A compound statement consists of the reserved word **begin**, followed by a list of statements separated by semicolons, terminated by the reserved word **end**.

Chapter 6
EXPRESSIONS

Expressions are algorithmic constructs specifying rules for the computation of values. They consist of operands (variables, constants, and function designators) combined by means of operators as defined in the following.

This section describes how to form expressions from the standard scalar types *Integer, Real, Boolean,* and *Char.* Expressions containing declared scalar types, **string** types, and **set** types are described on pages 54, 58, and 74, respectively.

6.1. OPERATORS

Operators fall into five categories, denoted by their order of precedence:

1) Unary minus (minus with one operand only).
2) **Not** operator.
3) Multiplying operators: *, /, **div**, **mod**, **and**, **shl**, and **shr**.
4) Adding operators: +, -, **or**, and **xor**.
5) Relational operators: =, <>, <, >, <=, >=, and **in**.

Sequences of operators of the same precedence are evaluated from left to right. Expressions within parentheses are evaluated first and independently of preceding or succeeding operators.

If both of the operands of the multiplying and adding operators are of type *Integer,* then the result is of type *Integer.* If one or both of the operands is of type *Real,* then the result is also of type *Real.*

6.1.1. UNARY MINUS

The unary minus denotes a negation of its operand, which may be either *Real* or *Integer* in type. A positive number so negated becomes negative, while a negative number becomes positive.

6.1.2. NOT OPERATOR

The **not** operator negates (inverses) the logical value of its Boolean operand:

```
not True    = False
not False   = True
```

TURBO Pascal also allows the **not** operator to be applied to an *Integer* operand, in which case bitwise negation takes place.

Examples:

```
not 0       = -1
not -15     = 14
not $2345   = $DCBA
```

6.1.3. MULTIPLYING OPERATORS

Operator	Operation	Operand type	Result type
*	multiplication	Real	Real
*	multiplication	Integer	Integer
*	multiplication	Real, Integer	Real
/	division	Real, Integer	Real
/	division	Integer	Real
/	division	Real	Real
div	Integer division	Integer	Integer
mod	modulus	Integer	Integer
and	arithmetic **and**	Integer	Integer
and	logical **and**	Boolean	Boolean
shl	shift left	Integer	Integer
shr	shift right	Integer	Integer

Examples:

```
12 * 34          = 408
123 / 4          = 30.75
123 div 4        = 30
12 mod 5         = 2
True and False   = False
12 and 22        = 4
2 shl 7          = 256
256 shr 7        = 2
```

6.1.4. ADDING OPERATORS

Operator	Operation	Operand type	Result type
+	addition	Real	Real
+	addition	Integer	Integer
+	addition	Real, Integer	Real
-	subtraction	Real	Real
-	subtraction	Integer	Integer
-	subtraction	Real, Integer	Real
or	arithmetic or	**Integer**	**Integer**
or	logical or	**Boolean**	**Boolean**
xor	arithmetic or	**Integer**	**Integer**
xor	logical or	**Boolean**	**Boolean**

Examples:

```
123 + 456        = 579
456 - 123.0      = 333.0
True or False    = True
12 or 22         = 30
True xor False   = True
12 xor 22        = 26
```

6.1.5. RELATIONAL OPERATORS

The relational operators work on all standard scalar types: *Real, Integer, Boolean, Char,* and *Byte.* Operands of type *Integer, Real,* and *Byte* may be mixed. The type of the result is always Boolean, i.e. *True* or *False.*

=	equal to
<>	not equal to
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to

Examples:

a = b	true if a is equal to b.
a <> b	true if a is not equal to b.
a > b	true if a is greater than b.
a < b	true if a is less than b.

a >= b	true if a is greater than or equal to b.
a <= b	true if a is less than or equal to b.

6.2. FUNCTION DESIGNATORS

A function designator is a function identifier optionally followed by a parameter list, which is one or more variables or expressions separated by commas and enclosed in parentheses. The occurrence of a function designator causes the function with that name to be activated. If the function is not one of the pre-defined standard functions, it must be declared before activation.

Examples:

```
Round(PlotPos)
Writeln(Pi * (Sqr(R)))
(Max(X,Y) < 25) and (Z > Sqrt(X * Y))
Volume(Radius,Height)
```

Chapter 7
STATEMENTS

The statement part defines the action to be carried out by the program (or subprogram) as a sequence of *statements*, each specifying one part of the action. In this sense Pascal is a sequential programming language: statements are executed sequentially in time; never simultaneously. The statement part is enclosed by the reserved words **begin** and **end**; within it, statements are separated by semi-colons. Statements may be either *simple* or *structured*.

7.1. SIMPLE STATEMENTS

Simple statements are statements which contain no other statements. These are the assignment statement, procedure statement, **goto** statement, and empty statement.

7.1.1. ASSIGNMENT STATEMENT

The most fundamental of all statements is the assignment statement. It is used to specify that a certain value is to be assigned to a certain variable. The assignment consists of a variable identifier followed by the assignment operator := followed by an expression.

Assignment is possible to variables of any type (except files) as long as the variable (or the function) and the expression are of the same type. As an exception, if the variable is of type *Real*, the type of the expression may be *Integer*.

Examples:

```
Angle := Angle * Pi;
AccessOK := False;
Entry := Answer = PassWord;
SphereVol := 4 * Pi * R * R;
```

7.1.2. PROCEDURE STATEMENT

A procedure statement serves to activate a previously defined user-defined procedure or a pre-defined standard procedure. The statement consists of a procedure identifier, optionally followed by a parameter list, which is a list of variables or expressions separated by

commas and enclosed in parentheses. When the procedure statement is encountered during program execution, control is transferred to the named procedure, and the value (or the address) of possible parameters are transferred to the procedure. When the procedure finishes, program execution continues from the statement following the procedure statement.

Examples:
```
Find(Name,Address);
Sort(Address);
UpperCase(Text);
UpdateCustFile(CustRecord);
```

7.1.3. GOTO STATEMENT

A **goto** statement consists of the reserved word **goto** followed by a label identifier. It serves to transfer further processing to that point in the program text which is marked by the label. The following rules should be observed when using **goto** statements:

1) Before use, labels must be declared. The declaration takes place in a label declaration in the declaration part of the block in which the label is used.

2) The scope of a label is the block in which it is declared. It is thus not possible to jump into or out of procedures and functions.

7.1.4. EMPTY STATEMENT

An empty statement is a statement which consists of no symbols, and which has no effect. It may occur whenever the syntax of Pascal requires a statement but no action is to take place.

Examples:
```
begin end.
while Answer <> '' do;
repeat until KeyPressed; {wait for any key to be hit}
```

7.2. STRUCTURED STATEMENTS

Structured statements are constructs composed of other statements which are to be executed in sequence (compound statements), conditionally (conditional statements), or repeatedly (repetitive statements). The discussion of the **with** statement is deferred to pages 70 pp.

7.2.1. COMPOUND STATEMENT

A compound statement is used if more than one statement is to be executed in a situation where the Pascal syntax allows only one statement to be specified. It consists of any number of statements separated by semi-colons and enclosed within the reserved words **begin** and **end**, and specifies that the component statements are to be executed in the sequence in which they are written.

Examples:

```
if Small > Big then
begin
   Tmp := Small;
   Small := Big;
   Big := Tmp;
end;
```

7.2.2. CONDITIONAL STATEMENTS

A conditional statement selects for execution a single one of its component statements.

7.2.2.1. IF STATEMENT

The **if** statement specifies that a statement be executed only if a certain condition (Boolean expression) is true. If it is false, then either no statement or the statement following the reserved word **else** is to be executed. Notice that **else** *must not* be preceded by a semicolon.

The syntactic ambiguity arising from the construct:

```
if expr1 then
   if expr2 then
      stmt1
   else
      stmt2
```

is resolved by interpreting the construct as follows:

```
if expr1 then
begin
   if expr2 then
      stmt1
   else
      stmt2
end
```

The **else** clause belongs generally to the last **if** statement which has no **else** clause.

Examples:

```
if Interest > 25 then
   Usury := True
else
   TakeLoan := OK;

if (Entry < 1) or (Entry > 100) then
begin
   Write('Range is 1 to 100, please re-enter: ');
   Read(Entry);
end
```

7.2.2.2. CASE STATEMENT

The **case** statement consists of an expression (the selector) and a list of statements, each preceded by a case label of the same type as the selector. It specifies that the one statement be executed whose case label is equal to the current value of the selector. If none of the case labels contain the value of the selector, then either no statement is executed, or, optionally, the statements following the reserved word **else** are executed. The **else** clause is an expansion of standard Pascal.

A case label consists of any number of constants or subranges separated by commas followed by a colon. A subrange is written as two constants separated by the subrange delimiter '..'. The type of the constants must be the same as the type of the selector. The statement following the case label is executed if the value of the selector equals one of the constants or if it lies within one of the subranges.

Valid selector types are all simple types, i.e., all scalar types except real.

Examples:

```
case Operator of
  '+': Result := Answer + Result;
  '-': Result := Answer - Result;
  '*': Result := Answer * Result;
  '/': Result := Answer / Result;
end;

case Year of
  Min..1939: begin
               Time := PreWorldWar2;
               Writeln('The world at peace...');
             end;
  1946..Max: begin
               Time := PostWorldWar2;
               Writeln('Building a new world.');
             end
     else    begin
               Time := WorldWar2;
               Writeln('We are at war');
             end;
end;
```

7.2.3. REPETITIVE STATEMENTS

Repetitive statements specify that certain statements are to be executed repeatedly. If the number of repetitions is known before the repetitions are started, the **for** statement is the appropriate construct to express this situation. Otherwise the **while** or the **repeat** statement should be used.

7.2.3.1. FOR STATEMENT

The **for** statement indicates that the component statement is to be repeatedly executed while a progression of values is assigned to a variable which is called the *control variable.* The progression can be ascending: **to** or descending: **downto** the *final value.*

The control variable, the *initial value,* and the *final value* must all be of the same type. Valid types are all simple types, i.e., all scalar types except real.

If the initial value is greater than the final value when using the **to** clause, or if the initial value is less than the final value when using the **downto** clause, the component statement is not executed at all.

Examples:

```
for I := 2 to 100 do if A[I] > Max then Max := A[I];
for I := 1 to NoOfLines do
begin
   ReadIn(Line);
   if Length(Line) < Limit then
        ShortLines := ShortLines + 1
   else
        LongLines := LongLines + 1
end;
```

The component statement of a **for** statement must *not* contain assignments to the control variable. If the repetition is to be terminated before the final value is reached, a **goto** statement must be used, although such constructs are not recommended -- it is better programming practice to use a **while** or **repeat** statement instead.

Upon completion of a **for** statement, the *control variable* equals the *final value,* unless the loop was not executed at all, in which case no assignment is made to the control variable.

7.2.3.2. WHILE STATEMENT

The expression controlling the repetition of a **while** statement must be of type Boolean. The statement is repeatedly executed as long as *expression* is *True.* If its value is false at the beginning, the statement is not executed at all.

Examples:

```
while Size > 1 do Size := Sqrt(Size);

while ThisMonth do
begin
   ThisMonth := CurMonth = SampleMonth;
   Process;
   {process this sample by the Process procedure}
end;
```

7.2.3.3. REPEAT STATEMENT

The expression controlling the repetition of a **repeat** statement must be of type Boolean. The sequence of statements between the reserved words **repeat** and **until** is executed repeatedly until the expression becomes true. As opposed to the **while** statement, the **repeat** statement is always executed at least once, as evaluation of the condition takes place at the end of the loop.

Examples:

```
repeat
   Write(^M, 'Delete this item? (Y/N)');
   Read(Answer);
until UpCase(Answer) in ['Y','N'];
```

Chapter 8
SCALAR AND SUBRANGE TYPES

The basic data types of Pascal are the scalar types. Scalar types constitute a finite and linear ordered set of values. Although the standard type *Real* is included as a scalar type, it does not conform to this definition. Therefore, *Reals* may not always be used in the same context as other scalar types.

8.1. SCALAR TYPE

Apart from the standard scalar types (*Integer, Real, Boolean, Char,* and *Byte*), Pascal supports user-defined scalar types, also called declared scalar types. The definition of a scalar type specifies, in order, all of its possible values. The values of the new type will be represented by identifiers, which will be the constants of the new type.

Examples:

```
type
   Operator = (Plus,Minus,Multi,Divide);
   Day      = (Mon,Tues,Wed,Thur,Fri,Sat,Sun);
   Month    = (Jan,Feb,Mar,Apr,May,Jun,
               Jul,Aug,Sep,Oct,Nov,Dec);
   Card     = (Club,Diamond,Heart,Spade);
```

Variables of the above type *Card* can assume one of four values, namely *Club, Diamond, Heart,* or *Spade.* You are already acquainted with the standard scalar type Boolean, which is defined as:

```
type
   Boolean = (False,True);
```

The relational operators =, <>, >, <, >=, and <= can be applied to all scalar types, as long as both operands are of the same type (reals and integers may be mixed). The ordering of the scalar type is used as the basis of the comparison, i.e., the order in which the values are introduced in the type definition. For the above type *Card*, the following is true:

```
Club < Diamond < Heart < Spade
```

The following standard functions can be used with arguments of scalar type:

Succ(Diamond)	The successor of *Diamond* (Heart).
Pred(Diamond)	The predecessor of *Diamond* (Club).
Ord(Diamond)	The ordinal value of *Diamond* (1).

The result type of *Succ* and *Pred* is the same as the argument type. The result type of *Ord* is *Integer*. The ordinal value of the first value of a scalar type is 0.

8.2. SUBRANGE TYPE

A type may be defined as a subrange of another already defined scalar type. Such types are called subranges. The definition of a subrange simply specifies the least and the largest value in the subrange. The first constant specifies the lower bound and must not be greater than the second constant, the upper bound. A subrange of type *Real* is not allowed.

Examples:

```
type
  Hemisphere   = (North, South, East, West);
  World        = (East..West)
  CompassRange = 0..360;
  Upper        = 'A'..'Z';
  Lower        = 'a'..'z';
  Degree       = (Celc, Fahr, Ream, Kelv);
  Wine         = (Red, White, Rose, Sparkling);
```

The type *World* is a subrange of the scalar type *Hemisphere* (called the *associated scalar type*). The associated scalar type of *CompassRange* is *Integer*, and the associated scalar type of *Upper* and *Lower* is *Char*.

You already know the standard subrange type *Byte*, defined as:

```
type
  Byte = 0..255;
```

A subrange type retains all the properties of its associated scalar type, being restricted only in its range of values.

The use of defined scalar types and subrange types is strongly recommended, as it greatly improves the readability of programs. Furthermore, run-time checks may be included in the program code (see page 56) to verify the values assigned to defined scalar variables and subrange variables. Another advantage of defined types and subrange types is that they often save memory. TURBO Pascal allocates only one byte of memory for variables of a defined scalar type or a subrange type with a total number of elements less than 256. Similarly, integer subrange variables, where lower and upper bounds are both within the range 0 through 255, occupy only one byte of memory.

8.3. TYPE CONVERSION

The *Ord* function may be used to convert scalar types into values of type integer. Standard Pascal does not provide a way to reverse this process, i.e., a way of converting an integer into a scalar value.

In TURBO Pascal, a value of a scalar type may be converted into a value of another scalar type, with the same ordinal value, by means of the *Retype* facility. Retyping is achieved by using the type identifier of the desired type as a function designator followed by one parameter enclosed in parentheses. The parameter may be a value of any scalar type except *Real*. Assuming the type definitions on pages 54 and 55, then:

```
Integer(Heart)      = 2
Month(10)           = Nov
HemiSphere(2)       = East
Upper(14)           ='O'
Degree(3)           = Kelv
Char(78)            = 'N'
Integer('7')        = 55
```

8.4. RANGE CHECKING

The generation of code to perform run-time range checks on scalar and subrange variables is controlled with the **R** compiler directive. The default setting is {$R-}, i.e., no checking is performed. When an assignment is made to a scalar or a subrange variable while this directive is active ({$R+}), assignment values are checked to be within

range. It is recommended to use this setting as long as a program is not fully debugged.

Examples:

```
program Rangecheck;
type
  Digit = 0..9;
Var
  Dig1,Dig2,Dig3: digit;
begin
  Dig1  := 5;          {valid}
  Dig2  := Dig1 + 3  {valid as Dig1 + 3 < = 9}
  Dig3  := 47;         {invalid but causes no error}
  {$R+} Dig3 := 55;  {invalid and causes a run time error}
  {$R-} Dig3 := 167;{invalid but causes no error}
end.
```

Chapter 9
STRING TYPE

TURBO Pascal offers the convenience of **string** types for processing of character strings, i.e., sequences of characters. String types are structured types, and in many ways are similar to **array** types (see Chapter 10). There is, however, one major difference between them. The number of characters in a string (i.e., the *length* of the string) may vary dynamically between 0 and a specified upper limit, whereas the number of elements in an array is fixed.

9.1. STRING TYPE DEFINITION

The definition of a string type must specify the maximum number of characters it can contain, i.e., the maximum length of strings of that type. The definition consists of the reserved word **string** followed by the maximum length enclosed in square brackets. The length is specified by an integer constant in the range 1 through 255. Notice that strings do not have a default length; the length must always be specified.

Example:

```
type
  FileName = string[14];
  ScreenLine = string[80];
```

String variables occupy the defined maximum length in memory plus one byte which contains the current length of the variable. The individual characters within a string are indexed from 1 through the length of the string.

9.2. STRING EXPRESSIONS

Strings are manipulated by the use of *string expressions*. String expressions consist of string constants, string variables, function designators, and operators.

The plus sign may be used to concatenate strings. The *Concat* function (see page 62) performs the same function, but the + operator is often more convenient. If the length of the result is greater than 255, a run-time error occurs.

Example:

```
'TURBO  ' + 'Pascal'     = 'TURBO Pascal'
'123' + '.' + '456'      = '123.456'
'A ' + 'B' + ' C ' + 'D ' = 'ABCD'
```

The relational operators =, <>, >, <, >=, and <= are lower in precedence than the concatenation operator. When applied to string operands, the result is a Boolean value (*True* or *False*). When comparing two strings, single characters are compared from left to right according to their ASCII values. If the strings are of different length, but identical up to and including the last character of the shorter string, then the shorter string is considered the smaller. Strings are equal only if their lengths as well as their contents are identical.

Examples:

```
'A' < 'B'                              is true
'A' > 'b'                              is false
'2' < '12'                             is false
'TURBO' = 'TURBO'                      is true
'TURBO ' = 'TURBO'                     is false
'Pascal Compiler' < 'Pascal compiler'  is true
```

9.3. STRING ASSIGNMENT

The assignment operator is used to assign the value of a string expression to a string variable.

Example:

```
Age   := 'fiftieth';
Line  := 'Many happy returns on your ' + Age + ' birthday.'
```

If the maximum length of a string variable is exceeded (by assigning too many characters to the variable), the excess characters are truncated. E.g., if the variable *Age* above was declared to be of type **string**[5], then after the assignment, the variable will only contain the five leftmost characters: 'fifth'.

9.4. STRING PROCEDURES

The following standard string procedures are available in TURBO Pascal:

9.4.1. DELETE

Syntax: Delete (*St, Pos, Num*);

Delete removes a substring containing *Num* characters from *St* starting at position *Pos*. *St* is a string variable and both *Pos* and *Num* are integer expressions. If *Pos* is greater than *Length*(*St*), no characters are removed. If an attempt is made to delete characters beyond the end of the string (i.e., *Pos* + *Num* exceeds the length of the string), only characters within the string are deleted. If *Pos* is outside the range 1..255, a run time error occurs.

If *St* has the value 'ABCDEFG' then:
Delete (St, 2, 4) will give *St* the value 'AFG'.
Delete (St, 2, 10) will give *St* the value 'A'.

9.4.2. INSERT

Syntax: Insert (*Obj, Target, Pos*);

Insert inserts the string *Obj* into the string *Target* at the position *Pos*. *Obj* is a string expression, *Target* is a string variable, and *Pos* is an integer expression. If *Pos* is greater than Length(*Target*), then *Obj* is concatenated to *Target*. If the result is longer than the maximum length of *Target*, then excess characters will be truncated and *Target* will only contain the leftmost characters. If *Pos* is outside the range 1..255, a run-time error occurs.

If *St* has the value 'ABCDEFG' then:
Insert('XX',St,3) will give *St* the value 'ABXXCDEFG'

9.4.3. STR

Syntax: Str(*Value,St*);

The *Str* procedure converts the numeric value of *Value* into a string and stores the result in *St*. *Value* is a write parameter of type integer or of type real, and *St* is a string variable. Write parameters are expressions with special formatting commands (see page 98).

If *I* has the value 1234 then:
Str (I:5,St) gives *St* the value ' 1234'.

If *X* has the value 2.5E4 then:
Str(X:10:0,St) gives *St* the value ' 2500'.

A function using the *Str* procedure must **never** be called by an expression in a *Write* or *Writeln* statement.

9.4.4. VAL

Syntax: Val(*St,Var,Code*);

Val converts the string expression *St* to an integer or a real value (depending on the type of *Var*) and stores this value in *Var*. *St* must be a string expressing a numeric value according to the rules applying to numeric constants (see page 35). Neither leading nor trailing spaces are allowed. *Var* must be an *Integer* or a *Real* variable and *Code* must be an integer variable. If no errors are detected, the variable *Code* is set to 0. Otherwise *Code* is set to the position of the first character in error, and the value of *Var* is undefined.

If *St* has the value '234' then:
Val (St,I,Result) gives *I* the value 234 and *Result* the value 0.

If *St* has the value '12x' then:
Val (St,I,Result) gives *I* an undefined value and *Result* the value 3.

If *St* has the value '2.5E4', and *X* is a *Real* variable, then:
Val (St,X,Result) gives *X* the value 2500 and *Result* the value 0.

A function using the *Var* procedure must **never** be called by an expression in a *Write* or *Writeln* statement.

9.5. STRING FUNCTIONS

The following standard string functions are available in TURBO Pascal:

9.5.1. COPY

Syntax: Copy(*St,Pos,Num*);

Copy returns a substring containing *Num* characters from *St* starting at position *Pos*. *St* is a string expression and both *Pos* and *Num* are integer expressions. If *Pos* exceeds the length of the string, the empty string is returned. If an attempt is made to get characters beyond the end of the string (i.e., *Pos* + *Num* exceeds the length of the string), only the characters within the string are returned. If *Pos* is outside the range 1..255, a run-time error occurs.

If *St* has the value 'ABCDEFG' then:
Copy(St,3,2) returns the value 'CD'
Copy(St,4,10) returns the value 'DEFG'
Copy(St,4,2) returns the value 'DE'

9.5.2. CONCAT

Syntax: Concat(*St1,St2*{,*StN*});

The *Concat* function returns a string which is the concatenation of its arguments in the order in which they are specified. The arguments may be any number of string expressions separated by commas (*St1, St2 .. StN*). If the length of the result is greater than 255, a run-time error occurs. As explained on page 58, the + operator can be used to obtain the same result, often more conveniently. *Concat* is included only to maintain compatibility with other Pascal compilers.

If *St1* has the value 'TURBO' and *St2* the value 'is fastest' then

```
Concat (St1,' PASCAL ', St2)
```

returns the value 'TURBO PASCAL is fastest'.

9.5.3. LENGTH

Syntax: Length(*St*);

Returns the length of the string expression *St*, i.e., the number of characters in *St*. The type of the result is integer.

If *St* has the value '123456789' then:
Length(St) returns the value 9

9.5.4. POS

Syntax: Pos(*Obj*,*Target*);

The *Pos* function scans the string *Target* to find the first occurrence of *Obj* within *Target*. *Obj* and *Target* are string expressions, and the type of the result is integer. The result is an integer denoting the position within *Target* of the first character of the matched pattern. The position of the first character in a string is 1. If the pattern is not found, *Pos* returns 0.

If *ST* has the value 'ABCDEFG' then
Pos('DE',St) returns the value 4
Pos('H',St) returns the value 0

9.6. STRINGS AND CHARACTERS

String types and the standard scalar type *Char* are compatible. Thus, whenever a string value is expected, a char value may be specified instead and vice versa. Furthermore, strings and characters may be mixed in expressions. When a character is assigned a string value, the length of the string must be exactly one; otherwise a run-time error occurs.

The characters of a string variable may be accessed individually through string indexing. This is achieved by appending an index expression of type integer, enclosed in square brackets, to the string variable.

Examples:

```
Buffer[5]
Line[Length(Line)-1]
Ord(Line[0])
```

As the first character of the string (at index 0) contains the length of the string, Length (String) is the same as Ord(String[0]). If assignment is made to the length indicator, it is the responsibility of the programmer to check that it is less than the maximum length of the string variable. When the range check compiler directive **R** is active ({$R+}), code is generated which insures that the value of a string index expression does not exceed the maximum length of the string variable. It is, however, still possible to index a string beyond its current dynamic length. The characters thus read are random, and assignments beyond the current length will not affect the actual value of the string variable.

Chapter 10
ARRAY TYPE

An array is a structured type consisting of a fixed number of components which are all of the same type, called the *component type* or the *base type*. Each component can be explicitly accessed by indices into the array. Indices are expressions of any scalar type placed in square brackets suffixed to the *array identifier*, and their type is called the *index type*.

10.1. ARRAY DEFINITION

The definition of an array consists of the reserved word **array** followed by the index type, enclosed in square brackets, followed by the reserved word **of**, followed by the component type.

Examples:

```
type
  Day = (Mon, Tue, Wed, Thu, Fri, Sat, Sun)
var
  WorkHour      : array[1..8] of Integer;
  Week          : array[1..7] of Day;

type
  Players       = (Player1,Player2,Player3,Player4);
  Hand          = (One,Two,Pair,TwoPair,Three,Straight,
                   Flush,FullHouse,Four,StraightFlush,RSF);
  LegalBid      = 1..200;
  Bid           = array[Players] of LegalBid;
var
  Player        : array[Players] of Hand;
  Pot           : Bid;
```

An array component is accessed by suffixing an index enclosed in square brackets to the array variable identifier.

```
Player[Player3]     := FullHouse;
Pot[Player3]        := 100;
Player[Player4]     := Flush;
Pot[Player4]        := 50;
```

As assignment is allowed between any two variables of identical type, entire arrays can be copied with a single assignment statement.

The **R** compiler directive controls the generation of code which will perform range checks on array index expressions at run time. The default mode is passive, i.e., {$R-}, and the {$R+} setting causes all index expressions to be checked against the limits of their index type.

10.2. MULTIDIMENSIONAL ARRAYS

The component type of an array may be any data type, i.e., the component type may be another array. Such a structure is called a *multidimensional array*.

Example:

```
type
  Card        = (Two,Three, Four, Five,Six,Seven,Eight,
                 Nine, Ten, Knight, Queen, King, Ace);
  Suit        = (Heart, Spade, Clubs, Diamonds);
  AllCards    = array[Suit] of array[1..13] of Card;

var
  Deck: AllCards;
```

A multi-dimensional array may be defined more conveniently by specifying the multiple indices thus:

```
type
  AllCards    = array[Suit, 1..13] of Card;
```

A similar abbreviation may be used when selectng an array component:

Deck[Hearts,10] is equivalent to Deck[Hearts][10]

It is, of course, possible to define multi-dimensional arrays in terms of previously defined array types.

Example:

```
type
  Pupils      = string[20];
  Class       = array[1..30] of Pupils;
```

```
  School          = array[1..100] of Class;
var
  J,P,Vacant      : Integer
  ClassA,
  ClassB          : Class;
  NewTownSchool   : School;
```

After these definitions, all of the following assignments are legal:

```
ClassA[J] := 'Peter';
NewTownSchool[5][21] := 'Peter Brown';
{Pupil no. J changes class:}
NewTownSchool[8,J] := NewTownSchool[7,J];
(Pupil no. P changes class and number:}
ClassA[Vacant] := ClassB[P];
```

10.3. CHARACTER ARRAYS

Character arrays are arrays with one index and components of the standard scalar type *Char*. Character arrays may be thought of as *strings* with a constant length.

In TURBO Pascal, character arrays may participate in *string* expressions, in which case the array is converted into a string of the length of the array. Thus, arrays may be compared and manipulated in the same way as strings, and string constants may be assigned to character arrays, as long as they are of the same length. String variables and values computed from string expressions cannot be assigned to character arrays.

10.4. PREDEFINED ARRAYS

TURBO Pascal offers two predefined arrays of type *Byte*, called *Mem* and *Port*, which are used to access CPU memory and data ports. These are discussed in Chapter 22.

Chapter 11
RECORD TYPE

A **record** is a structure consisting of a fixed number of components, called *fields*. Fields may be of different types. Each field is given a name, the *field identifier*, which is used to select it.

11.1. RECORD DEFINITION

The definition of a record type consists of the reserved word **record** succeeded by a *field list* and terminated by the reserved word **end.** The field list is a sequence of *record sections* separated by semi-colons, each consistng of one or more identifiers separated by commas, followed by a colon and either a type *identifier* or a type *descriptor.* Each record section thus specifies the identifier and type of one or more fields.

Example:

```
type
  DaysOfMonth = 1..31;
  Date = record
           Day:    DaysOfMonth;
           Month:  (Jan,Feb,Mar,Apr,May,Jun,
                    Jul,Aug,Sep,Oct,Nov,Dec);
           Year:   1900..1999;
         end
Var
  Birth: Date;
  WorkDay: array[1..5] of Date;
```

Day, Month, and *Year* are field identifiers. A field identifier must be unique only within the record in which it is defined. A field is referenced by the variable indentifier and the field identifier separated by a period.

Examples:

```
Birth.Month          := Jun;
Birth.Year           :=  1950;
WorkDay[Current]     := WorkDay[Current-1];
```

As with array types, assignment is allowed between entire records of identical types. Since record components may be of any type, constructs like the following "record of records of records" are possible:

```
type
  Name   = record
             FamilyName:  string[32];
             ChristianNames:  array[1..3] of string[16];
           end;
  Rate   = record
             NormalRate, Overtime,
             NightTime, Weekend:  Integer
           end;
  Date   = record
             Day:   1..31;
             Month: (Jan, Feb, Mar, Apr, May, Jun,
                     July, Aug, Sep, Oct, Nov, Dec);
             Year:  1900..1999;
           end;
  Person = record
             ID:  Name;
             Time:  Date;
           end;
  Wages  = record
             Individual:  Person;
             Cost:  Rate;
           end

Var        Salary, Fee:  Wages;
```

Assuming these definitions, the following assignments are legal:

```
Salary := Fee;
Salary.Cost.Overtime := 950;
Salary.Individual.Time := Fee.Individual.Time;
Salary.Individual.ID.FamilyName := 'Smith'
```

11.2. WITH STATEMENT

The use of records as described above does sometimes result in rather lengthy statements; it would often be easier if we could access individual fields in a record as if they were simple variables. This is the function of the **with** statement: it "opens up" a record so that field identifiers may be used as variable identifiers.

A **with** statement consists of the reseved word **with** followed by a list of record variables separated by commas followed by the reserved word **do** and finally a statement.

Within a **with** statement, a field is designated only by its field identifier, i.e., without the record variable identifier:

```
with Salary do
begin
   Individual := NewEmployee;
   Cost := StandardRates;
end;
```

Records may be ***nested*** within **with** statements, i.e., records of records may be "opened" as shown here:

```
with Salary, Individual, ID do
begin
   FamilyName := 'Smith';
   ChristianNames[1] := 'James';
end
```

This is equivalent to:

```
with Salary do with Individual do with ID do ...
```

The maximum "depth" of this nesting of **with** sentences, i.e., the maximum number of records which may be "opened" within one block, depends on your implementation and is discussed in Chapter 22.

11.3. VARIANT RECORDS

The syntax of a record type also provides for a variant part, i.e., alternative record structures. The variant part allows fields of a record to consist of a different number of different types of components, usually depending on the value of a *tag field.*

A variant part consists of a *tag field* of a previously defined type, whose values determine the variant, followed by labels corresponding to each possible value of the tag field. Each label heads a *field list* which defines the type of the variant corresponding to the label.

Assuming the existence of the type:

```
Origin = (Citizen,Alien);
```

and of the types *Name* and *Date,* the following record allows the field *CitizenShip* to have different structures depending on whether the value of the field is *Citizen* or *Alien:*

```
type
  Person =          record
                      PersonNameName;
                      BirthDate:  Date;
                      case CitizenShip: Origin of
                        Citizen:   (BirthPlace:  Name);
                        Alien:     (CountryofOrigin: Name;
                                    DateOfEntry: Date;
                                    PermitedUntil: Date;
                                    PortOfEntry:  Name);
                    end
```

In this variant record definition, the tag field is an explicit field which may be selected and updated like any other field. Thus, if *Passenger* is a variable of type *Person,* statements like the following are perfectly legal:

```
Passenger.CitizenShip := Citizen;

with Passenger, PersonName do
  if CitizenShip = Alien then Writeln(FamilyName);
```

The fixed part of a record, i.e., the part containing the common fields, must always precede the *variant part*. In the above example, the fields *PersonName* and *BirthDate* are the fixed fields. A record can only have one variant part. In a variant, the parentheses must be present, even if they will enclose nothing.

The maintenance of tag-field values is the responsibility of the programmer and not of TURBO Pascal. Thus, in the *Person* type above, the field *DateOfEntry* can be accessed even if the value of the tag field *CitizenShip* is not *Alien*. Actually, the tag-field identifier may be omitted altogether, leaving only the type identifier. Such record variants are known as *free unions*, as opposed to record variants with tag fields, which are called *discriminated unions*. The use of free unions is infrequent and should only be practiced by experienced programmers.

Chapter 12
SET TYPE

A **set** is a collection of related objects which may be thought of as a whole. Each object in such a set is caled a ***member*** or an ***element*** of the set. Examples of sets could be:

1) All integers between 0 and 100
2) The letters of the alphabet
3) The consonants of the alphabet

Two sets are equal if and only if their elements are the same. There is no ordering involved, so the sets [1,3,5], [5,3,1] and [3,5,1] are all equal. If the members of one set are also members of another set, then the first set is said to be included in the second. In the examples above, the third set is included in the second one.

There are three operations involving sets, similar to the addition, multiplication and subtraction operations on numbers:

The ***union*** (or sum) of two sets A and B (written A + B) is the set whose members are members of either A or B. For instance, the union of [1,3,5,7] and [2,3,4] is [1,2,3,4,5,7].

The ***intersection*** (or product) of two sets A and B (written A * B) is the set whose members are the members of both A and B. Thus, the intersection of [1,3,4,5,7] and [2,3,4] is [3,4].

The ***relative complement*** of B with respect to A (written A - B) is the set whose members are members of A but not of B; e.g., [1,3,5,7] - [2,3,4] is [1,5,7].

12.1. SET TYPE DEFINITION

Although in mathematics there are no restrictions on the objects which may be members of a set, Pascal only offers a restricted form of sets. The members of a set must all be of the same type, called the ***base type***, and the base type must be a simple type, i.e., any scalar type except real. A set type is introduced by the reserved words **set of** followed by a simple type.

Examples:

```
type
  DaysOfMonth = set of 0..31;
  WordWeek = set of Mon..Fri;
  Letter = set of 'A'..'Z';
  AdditiveColors = set of (Red,Green,Blue);
  Characters = set of Char;
```

In TURBO Pascal, the maximum number of elements in a set is 256, and the ordinal values of the base type must be within the range 0 through 255.

12.2. SET EXPRESSIONS

Set values may be computed from other set values through set expressions. Set expressions consist of set constants, set variables, set constructors, and set operators.

12.2.1. SET CONSTRUCTORS

A set constructor consists of one or more element specifications, separated by commas, and enclosed in square brackets. An element specification is an expression of the same type as the base type of the set, or a range expressed as two such expressions separated by two consecutive periods (..).

Examples:

```
['T','U','R','B','O']
[X,Y]
[X..Y]
[1..5]
['A'..'Z','a'..'z','0'..'9']
[1,33..10,12]
[]
```

The last example shows the *empty set*, which, as it contains no expressions to indicate its base type, is compatible with all set types. The set [1..5] is equivalent to the set [1,2,3,4,5]. If X > Y then [X..Y] denotes the empty set.

12.2.2. SET OPERATORS

The rules of composition specify set operator precedency according to the following three classes of operators:

1) * Set intersection.

2) + Set union.
   - Set difference.

3) = Test on equality.
   <> Test on inequality.
   >= *True* if all members of the second operand are included in the first operand.
   <= *True* if all members of the first operand are included in the second operand.
   in Test on set membership. The second operand is of a set type, and the first operand is an expression of the same type as the base type of the set. The result is true if the first operand is a member of the second operand, otherwise it is false.

Set disjunction (when two sets contain no common members) may be expressed as:

```
A * B = [];
```

that is, the intersection between the two sets is the empty set. Set expressions are often useful to clarify complicated tests. For instance, the test:

```
if (Ch='T') or (Ch='U') or (Ch='R') or (Ch='B') or (Ch='O')
```

can be expressed much more clearly as:

```
Ch in ['T','U','R','B','O']
```

And the test:

```
if (Ch >= '0') and (Ch <= '9') then ...
```

is better expressed as:

```
if Ch in ['0'..'9'] then ...
```

12.3. SET ASSIGNMENTS

Values resulting from set expressions are assigned to set variables using the assignment operator :=.

Examples:

```
type
   ASCII = set of 0..127;
var
   NoPrint,Print,AllChars: ASCII;
begin
   AllChars := [0..127];
   NoPrint := [0..31,127];
   Print := AllChars - NoPrint;
end.
```

Chapter 13
TYPED CONSTANTS

Typed constants are a TURBO specialty. A typed constant may be used exactly like a variable of the same type. Typed constants may thus be used as "initialized variables", because the value of a typed constant is defined, whereas the value of a variable is undefined until an assignment is made. Care should be taken, of course, not to assign values to typed constants whose values are actually meant to be **constant**.

The use of a typed constant saves code if the constant is used often in a program, because a typed constant is included in the program code only once, whereas an untyped constant is included every time it is used.

Typed constants are defined like untyped constants (see page 40), except that the definition specifies not only the *value* of the constant but also the *type*. In the definition the typed constant identifier is succeeded by a colon and a type identifier, which is then followed by an equal sign and the actual constant.

13.1. UNSTRUCTURED TYPED CONSTANTS

An unstructured typed constant is a constant defined as one of the scalar types:

```
const
  NumberOfCars: Integer = 1267;
  Interest: Real = 12.67;
  Heading: string[7] = 'SECTION';
  Xon: Char = ^Q;
```

Contrary to untyped constants, a typed constant may be used in place of a variable as a variable parameter to a procedure or a function. As a typed constant is actually a variable with a constant value, it cannot be used in the definition of other constants or types. Thus, as *Min* and *Max* are typed constants, the following construct is **illegal**:

```
const
  Min:  Integer = 0;
  Max:  Integer = 50;
type
  Range:  array[Min..Max] of integer
```

13.2. STRUCTURED TYPED CONSTANTS

Structured constants comprise *array constants, record constants,* and *set constants.* They are often used to provide initialized tables and sets for tests, conversions, mapping functions, etc. The following sections describe each type in detail.

13.2.1. ARRAY CONSTANTS

The definition of an array constant consists of the constant identifier succeeded by a colon and the type identifier of a previously defined array type followed by an equal sign and the constant value expressed as a set of constants separated by commas and enclosed in parentheses.

Examples:

```
type
  Status          = (Active,Passive,Waiting);
  StringRep       = array[Status] of string[7];
const
  Stat:  StringRep = ('active','passive','waiting');
```

The example defines the array constant *Stat,* which may be used to convert values of the scalar type *Status* into their corresponding string representations. The components of *Stat* are:

```
Stat[Active]    = 'active'
Stat[Passive]   = 'passive'
Stat[Waiting]   = 'waiting'
```

The component type of an array constant may be any type except *File* types and *Pointer* types. Character array constants may be specified both as single characters and as strings. Thus, the definition:

```
const
  Digits: array[0..9] of Char =
  ('0','1','2','3','4','5','6','7','8','9');
```

may be expressed more conveniently as:

```
const
  Digits: array[0..9] of Char = '0123456789';
```

13.2.2. MULTI-DIMENSIONAL ARRAY CONSTANTS

Multi-dimensional array constants are defined by enclosing the constants of each dimension in separate sets of parentheses, separated by commas. The innermost constants correspond to the rightmost dimensions.

```
Examples:
type
  Cube = array[0..1,0..1,0..1] of integer;
const
  Maze: Cube = (((0,1),(2,3)),((4,5),(6,7)));
begin
  Writeln(Maze[0,0,0],' = 0');
  Writeln(Maze[0,0,1],' = 1');
  Writeln(Maze[0,1,0],' = 2');
  Writeln(Maze[0,1,1],' = 3');
  Writeln(Maze[1,0,0],' = 4');
  Writeln(Maze[1,0,1],' = 5');
  Writeln(Maze[1,1,0],' = 6');
  Writeln(Maze[1,1,1],' = 7');
end.
```

13.2.3. RECORD CONSTANTS

The definition of a record constant consists of the constant identifier succeeded by a colon and the type identifier of a previously defined record type followed by an equal sign and the constant value expressed as a list of field constants separated by semi-colons and enclosed in parentheses.

Examples:

```
type
  Point          = record
                     X,Y,Z: integer;
                   end;
  OS             = (CPM,CPMPlus,ZSystem,OS280);
  UI             = (CCP,ZCPR,BGii,VFiler,VMenu);
  Computer       = record
                     OperatingSystems: array[1..4] of OS;
                     UserInterface: UI;
                   end;
const
  Origo: Point = (X:0; Y:0; Z:0);
  SuperComp:       Computer =
      (OperatingSystems: (CPM,CPMPlus,ZSystem,OS280);
       UserInterface: VMenu);
  Plane: array[1..3] of Point =
      ((X:1;Y:4;Z:5),(X:10;Y:-78;Z:45),(X:100;Y:10;Z:-7));
```

The field constants must be specified in the same order as they appear in the definition of the record type. If a record contains fields of file types or pointer types, then constants of that record type cannot be specified. If a record constant contains a variant, then it is the responsibility of the programmer to specify only the fields of the valid variant. If the variant contains a tag field, then its value must be specified.

13.2.4. SET CONSTANTS

A set constant consists of one or more element specifications separated by commas, and enclosed in square brackets. An element specification must be a constant or a range expression consisting of two constants separated by two consecutive periods (..).

Example:

```
type
  Up  = set of 'A'..'Z';
  Low = set of 'a'..'z';
const
  UpperCase : Up  = ['A'..'Z'];
  Vowels    : Low = ['a','e','i','o','u','y'];
  Delimiter : set of Char = [' '..'/', ':'..'?', '['..'`', '{'..' '];
```

Chapter 14
FILE TYPES

Files provide a program with channels through which it can pass data. A file can either be a ***disk file***, in which case data is written to and read from a magnetic device of some type, or a ***logical device***, such as the predefined files *Input* and *Output* which refer to the computer's standard I/O channels; the keyboard and the screen.

A *file* consists of a sequence of components of equal type. The number of components in a file (the size of the file) is not determined by the definition of the file; instead the Pascal system keeps track of file accesses through a *file pointer*, and each time a component is written to or read from a file, the file pointer of that file is advanced to the next component. As all components of a file are of equal length, the position of a specific component can be calculated. Thus, the file pointer can be moved to any component in the file, providing random access to any element of the file.

14.1. FILE TYPE DEFINITION

A file type is defined by the reserved words **file of** followed by the type of the components of the file, and a file identifier is declared by the same words followed by the definition of a previously defined file type.

Examples:

```
type
   ProductName = string[80];
   Product = file of record
                        Name: ProductName;
                        ItemNumber: Real;
                        InStock: Real;
                        MinStock: Real;
                        Supplier: Integer;
                     end
var
   ProductFile: Product;
   ProductNames: file of ProductName;
```

The component type of a file may be any type, except a file type; that is, with reference to the example above, **file of** *Product* is **not** allowed. File variables may appear in neither assignments nor expressions.

14.2. OPERATIONS ON FILES

The following sections describe the procedures available for file handling. The identifier *FilVar* used throughout denotes a file variable identifier declared as described above.

14.2.1. ASSIGN

Syntax: Assign(*FilVar,Str*);

Str is a string expression yielding any legal file name. This file name is assigned to the file variable *FilVar*, and all further operation on *FilVar* will operate on the disk file *Str*. Assign should never be used on a file which is in use.

14.2.2. REWRITE

Syntax: Rewrite(*FilVar*);

A new disk file of the name assigned to the file variable *FilVar* is created and prepared for processing, and the file pointer is set to the beginning of the file, i.e., component number 0. Any previously existing file with the same name is erased. A disk file created by rewrite is initially empty, i.e., it contains no elements.

14.2.3. RESET

Syntax: Reset(*FilVar*);

The disk file of the name assigned to the file variable *FilVar* is prepared for processing, and the file pointer is set to the beginning of the file, i.e., component number 0. *FilVar* must name an existing file, otherwise an I/O error occurs.

14.2.4. READ

Syntax: Read(*FilVar,Var*);

Var denotes one or more variables of the component type of *FilVar*, separated by commas. Each variable is read from the disk file, and following each read operation, the file pointer is advanced to the next component.

14.2.5. WRITE

Syntax: Write(*FilVar,Var*);

Var denotes one or more variables of the component type of *FilVar*, separated by commas. Each variable is written to the disk file, and following each write operation, the file pointer is advanced to the next component.

14.2.6. SEEK

Syntax: Seek(*FilVar,n*);

Seek moves the file pointer to the nth component of the file denoted by *FilVar*. *n* is an integer expression. The position of the first component is 0. Note that in order to expand a file it is possible to seek one component beyond the last component. The statement

```
Seek(FilVar, FileSize(FilVar));
```

thus places the file pointer at the end of the file. *FileSize* returns the number of components in the file, and as the components are numbered from zero, the returned number is one greater than the number of the last component.

14.2.7. FLUSH

Syntax: Flush(*FilVar*);

Flush empties the internal sector buffer of the disk file *FilVar*, and thus assures that the sector buffer is written to the disk if any write operations have taken place since the last disk update. *Flush* also

insures that the next read operation will actually perform a physical read from the disk file. *Flush* should never be used on a closed file.

14.2.8. CLOSE

Syntax: Close(*FilVar*);

The disk file associated with *FilVar* is closed, and the disk directory is updated to reflect the new status of the file. Notice that it is necessary to *Close* a file, even it it has only been read from – you would otherwise quickly run out of file handles.

14.2.9. ERASE

Syntax: Erase(*FilVar*);

The disk file associated with *FilVar* is erased. If the file is open, i.e., if the file has been reset or rewritten but not closed, it is good programming practice to close the file before erasing it.

14.2.10. RENAME

Syntax: Rename(*FilVar,Str*);

The name of the disk file associated with *FilVar* is changed to a new name given by the string expression *Str*. The disk directory is updated to show the new name of the file, and further operations on *FilVar* will operate on the file with the new name. *Rename* should never be used on an open file.

Note: It is the programmer's responsibility to assure that the file named by *Str* does not already exist. If it does, multiple occurrences of the same name may result. The following function returns *True* if the file name passed as a parameter exists, otherwise it returns *False*:

```
type
  Name=string[66];
:
:
function Exist(FileName: Name): boolean;
var
  Fil: file;
begin
  Assign(Fil, FileName);
  {$I-}
  Reset(Fil);
  {$I+}
  Exist := (IOresult = 0)
end;
```

14.3. FILE STANDARD FUNCTIONS

The following standard functions are applicable to files:

14.3.1. EOF

Syntax: EOF(*FilVar*);

A Boolean function which returns *True* if the file pointer is positioned at the end of the disk file, i.e., beyond the last component of the file. If not, *EOF* returns *False*.

14.3.2. FILEPOS

Syntax: FilePos(*FilVar*);

An integer function which returns the current position of the file pointer. The first component of a file is 0.

14.3.3. FILESIZE

Syntax: FileSize(*FilVar*);

An integer function which returns the size of the disk file, expressed as the number of components in the file. If *FileSize(FilVar)* is zero, the file is empty.

14.4. USING FILES

Before using a file, the *Assign* procedure must be called to assign the file name to a file variable. Before input and/or output operations are performed, the file must be opened with a call to *Rewrite* or *Reset*. This call will set the file pointer to point to the first component of the disk file, i.e., *FilePos(FilVar)* = 0. After *Rewrite, FileSize(FilVar)* is 0.

A disk file can be expanded only by adding components to the end of the existing file. The file pointer can be moved to the end of the file by executing the following sentence:

```
Seek(FilVar, FileSize(FilVar));
```

When a program has finished its input/output operations on a file, it should always call the *Close* procedure. Failure to do so may result in loss of data, as the disk directory is not properly updated.

The program below creates a disk file called PRODUCTS.DTA, and writes 100 records of the type *Product* to the file. This initializes the file for subsequent random access, so that records may be read and written anywhere in the file).

```
program InitProductFile;
const
  MaxNumberOfProducts = 100;
type
  ProductName = string[20];
  Product =          record
                       Name:  ProductName;
                       ItemNumber:  Integer;
                       InStock:  Real;
                       Supplier:  Integer;
                     end;
var
  ProductFile: file of Product;
  ProductRec:  Product;
  I:  Integer;
begin
  Assign(ProductFile, 'PRODUCT.DTA');
  Rewrite(ProductFile); {open the file and delete any data}
```

```
    with ProductRec do
    begin
      Name := ''; InStock := 0; Supplier := 0;
      for I := 1 to MaxNumberOfProducts do
      begin
        ItemNumber := I;
        Write(ProductFile,ProductRec);
      end;
    end;
    Close(ProductFile);
end.
```

The following program demonstrates the use of *Seek* on random files. The program is used to update the *ProductFile* created by the program in the previous example.

```
program UpDateProductFile;
const
  MaxNumberOfProducts = 100;
type
  ProductName = string[20];
  Product =          record
                       Name:  ProductName;
                       ItemNumber:  Integer;
                       InStock:  Real;
                       Supplier:  Integer;
                     end;
var
  ProductFile: file of Product;
  ProductRec:  Product;
  I, Pnr:  Integer;
begin
  Assign(ProductFile,'PRODUCT.DTA'); Reset(ProductFile);
  ClrScr;
  Write('Enter product number (0 = stop) '); Readln(Pnr);
  while Pnr in [1..MaxNumberOfProducts] do
  begin
    Seek(ProductFile,Pnr-1); Read(ProductFile,ProductRec);
    with ProductRec do
    begin
      Write('Enter name of product (',Name:20,') ');
```

```
            Readln(Name);
            Write('Enter number in stock (',InStock:20:0,') ');
            Readln(InStock);
            Write('Enter supplier number (',Supplier:20,') ');
            Readln(Supplier);
            ItemNumber := Pnr;
          end;
          Seek(ProductFile,Pnr-1);
          Write(ProductFile,ProductRec);
          ClrScr; Writeln;
          Write('Enter product number (0 = stop) '); Readln(Pnr);
        end;
        Close(ProductFile);
      end.
```

14.5. TEXT FILES

Unlike all other file types, *text files* are not simply sequences of values of some type. Although the basic components of a text file are characters, they are structured into lines, each line being terminated by an *end-of-line* marker (a CR/LF sequence). The file is further ended by an *end-of-file* marker (a Ctrl-Z). As the length of lines may vary, the position of a given line in a file cannot be calculated. Text files can therefore only be processed sequentially. Furthermore, input and output cannot be performed simultaneously to a text file.

14.5.1. OPERATIONS ON TEXT FILES

A text file variable is declared by referring to the standard type identifier *Text*. Subsequent file operations must be preceded by a call to *Assign* and a call to *Reset* or *Rewrite* must furthermore precede input or output operations.

Rewrite is used to create a new text file, and the only operation then allowed on the file is the appending of new components to the end of the file. *Reset* is used to open an existing file for reading, and the only operation allowed on the file is sequential reading. When a new text file is closed, an end-of-file mark is appended to the file automatically.

Character input and output on text files is made with the standard procedures *Read* and *Write*. Lines are processed with the special text file operators *Readln*, *Writeln*, and *Eoln*:

14.5.1.1. READLN

Syntax: Readln(*FilVar*);

Skips to the beginning of the next line, i.e., skips all characters up to and including the next CR/LF sequence.

14.5.1.2. WRITELN

Syntax: Writeln(*FilVar*);

Writes a line marker (a CR/LF sequence) to the text file.

14.5.1.3. EOLN

Syntax: Eoln(*FilVar*);

A Boolean function which returns *True* if the end of the current line has been reached, i.e., if the file pointer is positioned at the CR character of the CR/LF line marker. If EOF(*FilVar*) is true, Eoln(*FilVar*) is also true.

14.5.1.4. SEEKEOLN

Syntax: SeekEoln(*FilVar*);

Similar to Eoln, except that it skips blanks and tabs before it tests for an end-of-line marker. The type of the result is boolean.

14.5.1.5. SEEKEOF

Syntax: SeekEof(*FilVar*);

Similar to EOF, except that it skips blanks, tabs, and end-of-line markers (CR/LF sequences) before it tests for an end-of-file marker. The type of the result is boolean.

When applied to a text file, the EOF function returns the value *True* if the file pointer is positioned at the end-of-file mark (the CTRL/Z character ending the file). The *Seek* and *Flush* procedures and the *FilePos* and *FileSize* functions are not applicable to text files.

The following sample program reads a text file from disk and prints it on the pre-defined device *Lst*, which is the printer. Words surrounded by Ctrl-S in the file are printed underlined by this program.

```
program TextFileDemo;
var
  FilVar:          Text;
  Line,
  ExtraLine:       string[255];
  I:               Integer;
  UnderLine:       Boolean;
  FileName:        string[14];
begin
  UnderLine := False;
  Write('Enter name of file to list: ');
  Readln(FileName);
  Assign(FilVar,FileName);
  Reset(FilVar);
  while not Eof(FilVar) do
  begin
    Readln(FilVar,Line);
    I := 1; ExtraLine := '';
    for I := 1 to Length(Line) do
    begin
      if Line[I] <>^S [then
      begin
        Write(Lst,Line[I]);
        if UnderLine then ExtraLine := ExtraLine + '_'
        else ExtraLine := ExtraLine + ' ';
      end
      else UnderLine := not UnderLine;
    end;
    Write(Lst,^M); Writeln(Lst,ExtraLine);
  end; {while not Eof}
end.
```

Further extensions of the procedures *Read* and *Write*, which facilitate convenient handling of formatted input and output, are described on page 95.

14.5.2. LOGICAL DEVICES

In TURBO Pascal, external devices such as terminals, printers, and modems are regarded as *logical devices* which are treated like text files. The following logical devices are available:

CON: The console device. Output is sent to the operating system's console output device, usually the CRT, and input is obtained from the console input device, usually the keyboard. Contrary to the TRM: device (see below), the CON: device provides buffered input. Each *Read* or *Readln* from a text file assigned to the CON: device will input an entire line into a line buffer, and the operator is provided with a set of editing facilities during line input. For more details on console input, please refer to pages 92 and 95.

TRM: The terminal device. Output is sent to the operating system's console output device, usually the CRT, and input is obtained from the console input device, usually the keyboard. Input characters are echoed, unless they are control characters. The only control character echoed is a carriage return (CR), which is echoed as CR/LF.

KBD: The keyboard device (input only). Input is obtained from the operating system's console input device, usually the keyboard. Input is not echoed.

LST: The list device (output only). Output is sent to the operating system's list device, typically the line printer.

AUX: The auxiliary device. In CP/M, this is RDR: and PUN:.

USR: The user device. Output is sent to the user output routine, and input is obtained from the user input routine. For further details on user input and output, please refer to page 155.

These logical devices may be accessed through the pre-assigned files discussed on page 92, or they may be assigned to file variables, exactly like a disk file. There is no difference between *Rewrite* and *Reset* on a file assigned to a logical device. *Close* on a file assigned to a logical device performs no function, and an attempt to *Erase* such a file will cause an I/O error.

The standard functions *Eof* and *Eoln* operate differently on logical devices than on disk files. On a disk file, *Eof* returns *True* when the next character in the file is a Ctrl-Z, or when physical EOF is encountered, and *Eoln* returns *True* when the next character is a CR or a Ctrl-Z. Thus, *Eof* and *Eoln* are in fact "look ahead" routines.

As you cannot look ahead on a logical device, *Eoln* and *Eof* operate on the *last* character read instead of on the *next* character. In effect, *Eof* returns *True* when the last character read was a Ctrl-Z, and *Eoln* returns *True* when the last character read was a CR or a Ctrl-Z. The following table provides an overview of the operation of *Eoln* and *Eof*:

	On Files	On Logical Devices
Eoln is true if	**next** character is CR or Ctrl-Z, or if EOF is true	**current** character is CR or Ctrl-Z
Eof is true if	**next** character is Ctrl-Z, or if physical EOF is met	**current** character is Ctrl-Z

Table 14-1: Operation of Eoln and Eof

Similarly, the *Readln* procedure works differently on logical devices than on disk files. On a disk file, *Readln* reads all characters up to and including the CR/LF sequence, whereas on a logical device it only reads up to and including the first CR. The reason for this is again the inability to "look ahead" on logical devices, which means that the system has no way of knowing what character will follow the CR.

14.5.3. STANDARD FILES

As an alternative to assigning text files to logical devices as described above, TURBO Pascal offers a number of pre-declared text files which have already been assigned to specific logical devices and prepared for processing. Thus, the programmer is saved the reset/rewrite and close processes, and the use of these standard files further saves code:

Input The primary input file. This file is assigned to either the CON: device or to the TRM: device (see below for further detail).

Output The primary output file. This file is assigned to either the CON: device or to the TRM: device (see below for further detail).

Con Assigned to the console device (CON:).

Trm Assigned to the terminal device (TRM:).

Kbd Assigned to the keyboard device (KBD:).

Lst Assigned to the list device (LST:).

Aux Assigned to the auxiliary device (AUX:).

Usr Assigned to the user device (USR:).

The use of *Assign, Reset, Rewrite,* and *Close* on these files is illegal.

When the *Read* procedure is used without specifying a file identifier, it always inputs a line, even if some characters still remain to be read from the line buffer, and it ignores Ctrl-Z, forcing the operator to terminate the line with <RETURN>. The terminating <RETURN> is not echoed, and internally the line is stored with a Ctrl-Z appended to the end of it. Thus, when fewer values are specified on the input line than there are parameters in the parameter list, any *Char* variables in excess will be set to Ctrl-Z, strings will be empty, and numeric variables will remain unaltered.

The **B** compiler directive is used to control the "forced read" feature above. The default state is {$B+}. In this state, read statements without a file variable will always cause a line to be input from the console. If a {$B-} compiler directive is placed at the beginning of the program (**before** the declaration part), the shortened version of read will act as if the input standard file has been specified, i.e.:

Read(v1,v2...,vn) equals *Real(input,v1,v2,..vn)*

In this case, lines are only input when the line buffer has been emptied. The {$B-} state follows the definition of Standard Pascal I/O, whereas the default {$B+} state, not confirming to the standard in all aspects, provides better control of input operations.

If you don't want input echoed to the screen, you should read from the standard file *Kbd:*

Read(Kbd,Var)

As the standard files *Input* and *Output* are used very frequently, they are chosen by default if no file identifier is stated. The following list shows the abbreviated text file operations and their equivalents:

Write(Ch)	*Write(Output,Ch)*
Read(Ch)	*Read(Input,Ch)*
Writeln	*Writeln(Output)*
Readlin	*Readln(Input)*
Eof	*Eof(Input)*
Eoln	*Eoln(Input)*

The following program shows the use of the standard file *Lst* to list the file *ProductFile* (see page 86) on the printer:

```
program ListProductFile;
const
  MaxNumberOfProducts = 100;
type
  ProductName = string[20];
  Product =          record
                       Name: ProductName;
                       ItemNumber: Integer;
                       InStock: Real;
                       Supplier: Integer;
                     end;
var
  ProductFile: file of Product;
  ProductRec: Product;
  I: Integer;
begin
  Assign(ProductFile,'PRODUCT.DTA'); Reset(ProductFile);
  for I := 1 to MaxNumberOfProducts do
  begin
    Read(ProductFile,ProductRec);
    with ProductRec do
    begin
      if Name <> '' then
        Writeln(Lst,'Item: ',ItemNumber:5,' ', Name:20,
                    ' From: ', Supplier:5,
```

```
                    ' Now in stock: ',InStock:0:0);
        end;
    end;
    Close(ProductFile);
end.
```

14.6. TEXT INPUT AND OUTPUT

Input and output of data in readable form is done through *text files*, as described on page 88. A text file may be assigned to any device, i.e., a disk file or one of the standard I/O devices. Input and output on text files is done with the standard procedures *Read*, *Readln*, *Write*, and *Writeln*, which use a special syntax for their parameter lists to facilitate maximum flexibility of input and output.

In particular, parameters may be of different types, in which case the I/O procedures provide automatic data conversation to and from the basic *Char* type of text files.

If the first parameter of an I/O procedure is a variable identifier representing a text file, then I/O will act on that file. If not, I/O will act on the standard files *Input* and *Output*. See page 92 for more detail.

14.6.1. READ PROCEDURE

The *Read* procedure provides input of characters, strings, and numeric data. The syntax of the *Read* statement is:

```
Read(Var1,Var2,...,VarN)
or
Read(FilVar,Var1,Var2,...,VarN)
```

where *Var1, Var2,...,VarN* are variables of type *Char, String, Integer* or *Real*. In the first case, the variables are input from the standard file *Input*, usually the keyboard. In the second case, the variables are input from the text file which is previously assigned to *FilVar* and prepared for reading.

With a variable of type *Char*, *Read* reads one character from the file and assigns that character to the variable. If the file is a disk file, *Eoln*

is true if the next character is a CR or a Ctrl-Z, and *Eof* is true if the next character is a Ctrl-Z, or the physical end-of-file is met. If the file is a logical device (including the standard files *Input* and *Output*), *Eoln* is true if the character read was a CR or if *Eof* is *True*, and *Eof* is true if the character read was a Ctrl-Z.

With a variable of type **string**, *Read* reads as many characters as allowed by the defined maximum length of the string, unless *Eoln* or *Eof* is reached first. *Eoln* is true if the character read was a CR or if *Eof* is *True*, and *Eof* is true if the last character read is a Ctrl-Z, or the actual end of the file is met.

With a numeric variable (*Integer* or *Real*), *Read* expects a string of characters which complies with the format of a numeric constant of the relevant type as defined on page 35. Any blanks, tabs, CRs, or LFs preceding the string are skipped. The string must be no longer than 30 characters, and it must be followed by a blank, a tab, a CR, or a Ctrl-Z. If the string does not conform to the expected format, an I/O error occurs. Otherwise the numeric string is converted to a value of the appropriate type and assigned to the variable. When reading from a disk file, if the input string ends with a blank or a tab, the next *Read* or *Readln* will start with the character immediately following that blank or tab. For both disk files and logical devices, *Eoln* is true if the string ends with a CR or a Ctrl-Z, and *Eof* is true if the string ends with a Ctrl-Z.

A special case of numeric input is when *Eoln* or *Eof* is true at the beginning of the *Read* (e.g., if input from the keyboard is only a CR). In that case no new value is assigned to the variable, and the variable retains its former value.

If the input file is assigned to the console device (CON:), or if the standard file *Input* is used in the {$B-} mode (default), special rules apply to the reading of variables. On a call to *Read* or *Readln*, a line is input from the console and stored into a buffer, and the reading of variables then uses this buffer as the input source. This allows for editing during entry. The following editing facilities are available:

BACKSPACE or **DEL**

Backspaces one character position and deletes the character there. BACKSPACE is usually generated by pressing the key

marked BS or BACKSPACE or by pressing Ctrl-H. DEL is usually generated by the key thus marked, or in some cases RUB or RUBOUT.

ESCAPE and **Ctrl-X**
Backspaces to the beginning of the line and erases all characters input.

Ctrl-D
Recalls one character from the last input line.

Ctrl-R
Recalls the last input line.

RETURN or **Ctrl-M**
Terminates the input line and stores an end-of-line marker (a CR/LF sequence) in the line buffer. This code is generated by pressing the key marked RETURN or ENTER. The CR/LF is **not** echoed to the screen.

Ctrl-Z
Terminates the input line and stores an end-of-file marker (a Ctrl-Z character) in the line buffer.

The input line is stored internally with a Ctrl-Z appended to the end of it. Thus, if fewer values are specified on the input line than the number of variables in *Read*'s parameter list, any *Char* variables in excess will be set to Ctrl-Z, any *String* will be empty, and numeric variables will remain unchanged.

The maximum number of characters that can be entered on an input line from the console is 127 by default. However, you may lower this limit by assigning an integer in the range 0 through 127 to the predefined variable *BufLen*.

Example:

```
Write('File name (max. 14 chars): ');
BufLen := 14;
Read(FileName);
```

Assignments to *BufLen* affect only the immediately following *Read*. After that, *BufLen* is restored to 127.

14.6.2. READLN PROCEDURE

The *Readln* procedure is identical to the *Read* procedure, except that after the last variable has been read, the remainder of the line is skipped, i.e., all characters up to and including the next CR/LF sequence (or the next CR on a logical device) are skipped. The syntax of the procedure statement is:

```
Readln(Var1,Var2,...,VarN)
or
Readln(FilVar,Var1,Var2,...,VarN)
```

After a *Readln*, the following *Read* or *Readln* will read from the beginning of the next line. *Readln* may also be called without parameters:

```
Readln
or
Readln(FilVar)
```

in which case the remaining of the line is skipped. When *Readln* is reading from the console (standard file *Input* or a file assigned to CON:), the terminating CR is echoed to the screen as a CR/LF sequence, as opposed to *Read*.

14.6.3. WRITE PROCEDURE

The *Write* procedure provides output of characters, strings, boolean values, and numeric values. The syntax of a *Write* statement is:

```
Write(Var1,Var2,...,VarN)
or
Write(FilVar,Var1,Var2,...,VarN)
```

where *Var1, Var2,...,VarN* (the *write parameters*) are variables of type *Char, String, Boolean, Integer* or *Real*, optionally followed by a colon and an integer expression defining the width of the output field. In the first case, the variables are output to the standard file *Output*, usually

the screen. In the second case, the variables are output to the text file which is previously assigned to *FilVar*.

The format of a *write parameter* depends on the type of the variable. In the following descriptions of the different formats and their effects, the symbols:

l, m, n	denote *Integer* expressions,
R	denotes a *Real* expression,
Ch	denotes a *Char* expression,
S	denotes a *String* expression, and
B	denotes a *Boolean* expression.

14.6.3.1. WRITE PARAMETERS

Ch The character *Ch* is output.

Ch:n The character *Ch* is output right-adjusted in a field which is *n* characters wide, i.e., *Ch* is preceded by *n*-1 blanks.

S The string *S* is output. Arrays of characters may also be output, as they are compatible with strings.

S:n The string *S* is output right-adjusted in a field which is *n* characters wide, i.e., *S* is preceded by *n - Length(S)* blanks.

B Depending on the value of *B*, either the word TRUE or the word FALSE is output right-adjusted in a field which is *n* characters wide.

I The decimal representation of the value of *I* is output.

I:n The decimal representation of the value of *I* is right-adjusted and output in a field which is *n* characters wide.

R The decimal representation of the value of *R* is right-adjusted and output in a field which is 18 characters wide, using floating point format. For *R* >= 0.0, the format is:

□□#.##########E*##

For R < 0.0, the format is:

□-#.##########E*##

where □ represents a blank, # represents a digit, and * represents either plus or minus.

R:n The decimal representation of the value of *R* is right-adjusted and output in a field *n* characters wide, using floating point format. For *R* >= 0.0:

blanks#.digitsE*##

For R < 0.0:

blanks-#.digitsE*EE

where *blanks* represent zero or more blanks, *digits* represents from one to ten digits, # represents a digit, and * represents either plus or minus. As at least one digit is output after the decimal point, the field width is at minimum 7 characters (8 for R < 0.0).

R:n:m The decimal representation of the value of *R* is right-adjusted and output in a field *n* characters wide, using fixed point format with *m* digits after the decimal point. No decimal part, and no decimal point, is output if *m* is 0. *m* must be in the range 0 through 24; otherwise floating point format is used. The number is preceded by an appropriate number of blanks to make the field width *n*.

14.6.4. WRITELN PROCEDURE

The *Writeln* procedure is identical to the *Write* procedure, except that a CR/LF sequence is output after the last value. The syntax of the *Writeln* statement is:

Writeln(Var1,Var2,...,VarN)
or
Writeln(FilVar,Var1,Var2,...,VarN)

A *Writeln* with no write parameters outputs an empty line consisting of a CR/LF sequence:

Writeln or *Writeln(FilVar)*

14.7. UNTYPED FILES

Untyped files are low-level I/O channels primarily used for direct access to any disk file using a record size of 128 bytes.

In input and output operations to untyped files, data is transferred directly between the disk file and the variable, thus saving the space required by the sector buffer required by typed files. An untyped file variable therefore occupies less memory than other file variables. As an untyped file is furthermore compatible with any file, the use of an untyped file is therefore to be preferred if a file variable is required only for *Erase*, *Rename* or other operations not requiring input/output.

An untyped file is declared with the reserved word **file**:

```
var
  DataFile: file;
```

14.7.1. BLOCKREAD AND BLOCKWRITE

All standard file-handling procedures and functions except *Read*, *Write*, and *Flush* are allowed on untyped files. *Read* and *Write* are replaced by two special high-speed transfer procedures: *BlockRead* and *BlockWrite*. The syntax of a call to these procedures is:

BlockRead(FilVar,Var,Recs)
BlockWrite(FilVar,Var,Recs)

or

BlockRead(FilVar,Var,Recs,Result)
BlockWrite(FilVar,Var,Recs,Result)

where *FilVar* is the variable identifier of an untyped file, *Var* is any variable, and *Recs* is an integer expression defining the number of 128-byte records to be transferred between the disk file and the variable. The **optional** parameter *Result* returns the number of records actually transferred.

The transfer starts with the first byte occupied by the variable *Var*. The programmer must insure that the variable *Var* occupies enough space to accommodate the entire data transfer. A call to *BlockRead* or *BlockWrite* also advances the file pointer *Recs* records.

A file to be operated on by *BlockRead* or *BlockWrite* must first be prepared by *Assign* and *Rewrite* or *Reset*. *Rewrite* creates and opens a new file, and *Reset* opens an existing file. After processing, *Close* should be used to ensure proper termination.

The standard function *EOF* works as with typed files. So do standard functions *FilePos* and *FileSize* and standard procedure *Seek*, using a component size of 128 bytes (the record size used by *BlockRead* and *BlockWrite*.

The following program uses untyped files to copy files of any type. Notice the use of the optional fourth parameter on *BlockRead* to check the number of records actually read from the source file.

```
program FileCopy;
const
  RecSize          = 128;
  BufSize          = 200;
var
  Source, Dest: File;
  SourceName,
  DestName:        string[14];
  Buffer:          array[1..RecSize,1..BufSize] of Byte;
  RecsRead:        Integer;
begin
  Write('Copy from: ');
  Readln(SourceName);
  Assign(Source, SourceName);
  Reset(Source);
  Write('      To: ');
  Readln(DestName);
  Assign(Dest, DestName);
  Rewrite(Dest);
  repeat
    BlockRead(Source,Buffer,BufSize,RecsRead);
    BlockWrite(Dest,Buffer,RecsRead);
```

```
    until RecsRead = 0;
    Close(Source); Close(Dest);
end.
```

14.8. I/O CHECKING

The **I** compiler directive is used to control generation of run-time I/O error checking code. The default state is active, i.e., {$I+}, which causes calls to an I/O check routine after each I/O operation. I/O errors then cause the program to terminate, and an error message indicating the type of error is displayed.

If I/O checking is passive, i.e., {$I-}, no run-time checks are performed. An I/O error thus does not cause the program to stop, but suspends any further I/O until the standard function *IOresult* is called. When this is done, the error condition is reset and I/O may be performed again. It is now the programmer's responsibility to take proper action according to the type of I/O error. A zero returned by *IOresult* indicates a successful operation, anything else means that an error occurred during the last I/O operation. Appendix G lists all error messages and their numbers. **Notice** that as the error condition is reset when IOresult is called, subsequent calls to IOresult will return zero until the next I/O error occurs.

The *IOresult* function is very convenient in situations where a program halt is an unacceptable result of an I/O error, as in the following example. This procedure continues to ask for a file name until the attempt to reset the file is successful (i.e., until an existing file name is entered):

```
procedure  OpenInFile;
begin
  repeat
    Write('Enter name of input file ');
    Readln(InFileName);
    Assign(InFile, InFileName);
    {$I-} Reset(InFile) {$I+} ;
    OK := (IOresult = 0);
    if not OK then
      Writeln('Cannot find file ',InFileName);
  until OK;
```

```
end;
```

When the I directive is passive ({$I-}), the following standard procedures should be followed by a check of IOresult to ensure proper error handling:

Assign	Close	Read	Rewrite
BlockRead	Erase	ReadLn	Seek
BlockWrite	Execute	Rename	Write
Chain	Flush	Reset	WriteLn

Chapter 15
POINTER TYPES

Variables discussed up to now have been *static*, i.e., their form and size is pre-determined, and they exist throughout the entire execution of the block in which they are declared. Programs, however, frequently need the use of a data structure which varies in form and size during execution. *Dynamic* variables serve this purpose, as they are generated as the need arises and may be discarded after use.

Such dynamic variables are not declared in an explicit variable declaration like static variables, and they cannot be referenced directly by identifiers. Instead, a special variable containing the memory address of the variable is used to *point* to the variable. This special variable is called a *pointer variable.*

15.1. DEFINING A POINTER VARIABLE

A pointer type is defined by the pointer symbol ^ succeeded by the type *identifier* of the dynamic variables which may be referenced by pointer variables of this type.

The following shows how to declare a record with associated pointers. The type *PersonPointer* is declared as a *pointer* to variables of type *PersonRecord*:

```
type
  PersonPointer = ^PersonRecord;
  PersonRecord =    record
                      Name: string[50];
                      Job: string[50];
                      Next: PersonPointer;
                    end;
var
  FirstPerson, LastPerson, NewPerson: PersonPointer;
```

The variables *FirstPerson, LastPerson* and *NewPerson* are thus *pointer variables* which can point at records of type *PersonRecord.* As shown above, the type identifier in a pointer type definition may refer to an identifier which is not yet defined.

15.2. ALLOCATING VARIABLES (NEW)

Before it makes any sense to use any of these pointer variables we must, of course, have some variables to point at. New variables of any type are allocated with the standard procedure *New*. The procedure has one parameter which must be a pointer to variables of the type we want to create.

A new variable of type *PersonRecord* can thus be created by the statement:

```
New(FirstPerson);
```

which has the effect of having *FirstPerson* point at a dynamically allocated record of type *PersonRecord*.

Assignments between pointer variables can be made as long as both pointers are of identical type. Pointers of identical type may also be compared using the relational operators = and <>, returning a *Boolean* result (*True* or *False*).

The pointer value **nil** is compatible with all pointer types. **nil** points to no dynamic variable, and may be assigned to pointer variables to indicate the absence of a usable pointer. **nil** may also be used in comparisons.

Variables created by the standard procedure *New* are stored in a stack-like structure called the *heap*. The TURBO Pascal system controls the heap by maintaining a heap pointer which at the beginning of the program is initialized to the address of the first free byte in memory. On each call to *New*, the heap pointer is moved towards the top of free memory the number of bytes corresponding to the size of the new dynamic variable.

15.3. MARK AND RELEASE

When a dynamic variable is no longer required by the program, the standard procedures *Mark* and *Release* are used to reclaim the memory allocated to these variables. The *Mark* procedure assigns the value of the heap pointer to a variable. The syntax of a call to *Mark* is:

```
Mark(Var);
```

where *Var* is a pointer variable. The *Release* procedure sets the heap pointer to the address contained in its argument. The syntax is:

```
Release(Var);
```

where *Var* is a pointer variable, previously set by *Mark*. *Release* thus discards *all* dynamic variables above this address, and cannot release the space used by variables in the middle of the heap. If you want to do that, you should use *Dispose* (see page 109) instead of *Mark* and *Release*.

The standard function *MemAvail* is available to determine the available space on the heap at any given time. Further discussion is deferred to Chapter 22.

15.4. USING POINTERS

Suppose we have used the *New* procedure to create a series of records of type *PersonRecord* (as in the example on the following page) and that the field *Next* in each record points at the next *PersonRecord* created. Then the following statements will go through the list and write the contents of each record (*FirstPerson* points to the first person in the list):

```
while FirstPerson <> nil do
with FirstPerson^ do
begin
   Writeln(Name,' is a ',Job);
   FirstPerson := Next;
end;
```

FirstPerson^.Name may be read as *FirstPerson's.Name*, i.e., the field *Name* in the record pointed to by *FirstPerson*.

The following demonstrates the use of pointers to maintain a list of names and related job desires. Names and job desires will be read in until a blank name is entered. Then the entire list is printed. Finally, the memory used by the list is released for other use. The pointer variable *HeapTop* is used only for the purpose of recording and storing the

initial value of the heap pointer. Its definition as a ^Integer (pointer to integer) is thus totally arbitrary.

```
procedure Jobs;
type
  PersonPointer = ^PersonRecord;
  PersonRecord =   record
                     Name: string[50];
                     Job: string[50];
                     Next: PersonPointer;
                   end;
var
  HeapTop: ^Integer;
  FirstPerson, LastPerson, NewPerson: PersonPointer;
  Name: string[50];
begin
  FirstPerson := nil;
  Mark(HeapTop);
  repeat
    Write('Enter name: ');
    Readln(Name);
    if Name <> '' then
    begin
      New(NewPerson);
      NewPerson^.Name := Name;
      Write('Enter profession: ');
      Readln(NewPerson^.Job);
      Writeln;
      if FirstPerson = nil then
          FirstPerson := NewPerson
      else
        LastPerson^.Next := NewPerson;
        LastPerson := NewPerson;
        LastPerson^.Next := nil;
    end;
  until Name = '';
  Writeln;
  while FirstPerson <> nil do
  with FirstPerson^ do
  begin
    Writeln(Name,' is a ',Job);
```

```
        FirstPerson := Next;
      end;
      Release(HeapTop);
    end.
```

15.5. DISPOSE

Instead of *Mark* and *Release,* standard Pascal's *Dispose* procedure may be used to reclaim space on the heap.

NOTE: *Dispose* and *Mark/Release* use entirely different approaches to heap management — **and never the twain shall meet!** Any one program uses **either** *Dispose* **or** *Mark/Release* to manage the heap. Mixing them will produce unpredictable results.

The syntax is:

```
Dispose(Var);
```

where *Var* is a pointer variable.

Dispose allows dynamic memory used by a *specific pointer variable* to be reclaimed for use, as opposed to *Mark* and *Release* which releases the entire heap *from the specified pointer variable upward.*

Suppose you have a number of variables which have been allocated on the heap. The following figure illustrates the contents of the heap and the effect of *Dispose(Var3)* and *Mark(Var3)/Release(Var3)*:

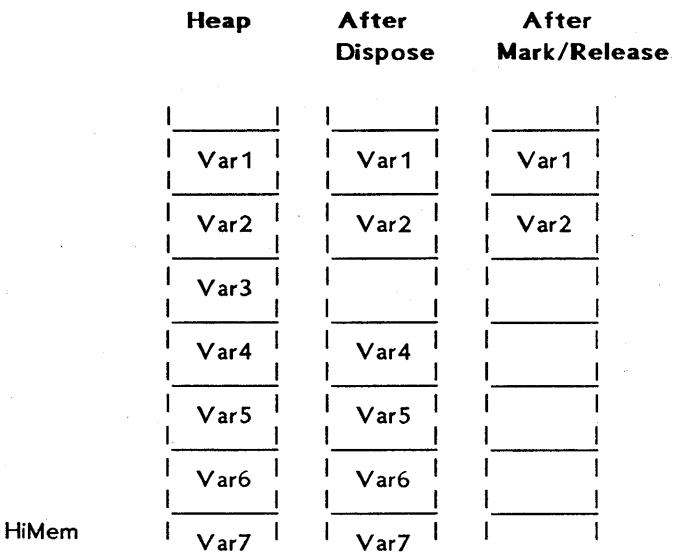


Figure 15-1: Using Dispose

After *Disposing* a pointer variable, the heap may thus consist of a number of memory areas in use interspersed with a number of free areas. Subsequent calls to *New* will use these if the new pointer variable fits into the space.

15.6. GETMEM

The standard procedure *GetMem* is used to allocate space on the heap. Unlike *New*, which allocates as much space as required by the **type** pointed to by its argument, *GetMem* allows the programmer to control the amount of space allocated. *GetMem* is called with two parameters:

```
GetMem(PVar, I)
```

where *PVar* is any pointer variable, and *I* is an integer expression giving the number of bytes to be allocated.

15.7. FREEMEM

Syntax: FreeMem;

The *FreeMem* standard procedure is used to reclaim an entire block of space on the heap. It is thus the counterpart of *GetMem*. *FreeMem* is called with two parameters:

```
FreeMem(PVar, I);
```

where *PVar* is any pointer variable, and *I* is an integer expressing giving the number of bytes to be reclaimmed, which must be **exactly** the number of bytes previously allocated to that variable by *GetMem*.

15.8. MAXAVAIL

Syntax: MaxAvail;

The *MaxAvail* standard function returns the size of the largest consecutive block of free space on the heap. On 8-bit systems this space is in bytes. The result is an *Integer*, and if more than 32767 bytes are available, *MaxAvail* returns a negative number. The correct number of free bytes is then calculated as 65536.0 + *MaxAvail*. Notice the use of a real constant to generate a *Real* result, because the result is greater than *MaxInt*.

15.9. HINTS

No range checking is done on pointers. It is the responsibility of the programmer to ensure that a pointer points to a legal address.

If you have difficulties using pointers, a drawing of what you are attempting to do often clears things up.

Chapter 16
PROCEDURES AND FUNCTIONS

A Pascal program consists of one or more ***blocks***, each of which may again consist of blocks, etc. One such block is a ***procedure***, another is a ***function*** (in common called ***subprograms***). Thus, a procedure is a separate part of a program, and it is activated from elsewhere in the program by a ***procedure statement*** (see page 47). A function is rather similar, but it computes and returns a value when its identifier, or ***designator***, is encountered during execution (see page 46).

16.1. PARAMETERS

Values may be passed to procedures and functions through ***parameters***. Parameters provide a substitution mechanism which allows the logic of the subprogram to be used with different initial values, thus producing different results.

The procedure statement or function designator which invokes the subprogram may contain a list of parameters, called the ***actual parameters***. These are passed to the ***formal parameters*** specified in the subprogram ***heading***. The order of parameter passing is the order of appearance in the parameter lists. Pascal supports two different methods of parameter passing: by ***value*** and by ***reference***, which determines the effect that changes of the formal parameters have on the actual parameters.

When parameters are passed by ***value***, the formal parameter represents a local variable in the subprogram, and changes of the formal parameters have no effect on the actual parameter. The actual parameter may be any expression, including a variable, with the same type as the corresponding formal parameter. Such parameters are called ***value parameters*** and are declared in the subprogram heading as in the following example. This and the following examples show procedure headings; see page 121 for a description of function headings.

```
procedure Example(Num1,Num2: Number; Str1,Str2: Txt);
```

Number and *Txt* are previously defined types (e.g., *Integer* and **string**[255]), and *Num1*, *Num2*, *Str1*, and *Str2* are the *formal parameters*

to which the value of the *actual parameters* are passed. The types of the formal and the actual parameters must correspond.

Notice that the type of the parameters in the parameter part must be specified as a previously defined *type identifier*. Thus, the construct:

```
procedure Select(Model: array[1..500] of Integer);
```

is **not** allowed. Instead, the desired type should be defined in the **type** definition of the block, and the *type identifier* should then be used in the parameter declaration:

```
type
  Range = array[1..500] of Integer;

procedure Select(Model: Range);
```

When a parameter is passed *by reference*, the formal parameter in fact represents the actual parameter throughout the execution of the subprogram. Any changes made to the formal parameter is thus made to the actual parameter, which must therefore be a *variable*. Parameters passed by reference are called *variable parameters*, and are declared as follows:

```
procedure Example(var Num1,Num2: Number)
```

Value parameters and variable parameters may be mixed in the same procedure as in the following example:

```
procedure Example(var Num1,Num2: Number; Str1,Str2: Txt);
```

in which *Num1* and *Num2* are variable parameters and *Str1* and *Str2* are value parameters.

All address calculations are done at the time of the procedure call. Thus, if a variable is a component of an array, its index expression(s) are evaluated when the subprogram is called.

Notice that **file** parameters must always be declared as variable parameters.

When a large data structure, such as an array, is to be passed to a subprogram as a parameter, the use of a variable parameter will save both time and storage space, as the only information then passed on to the subprogram is the address of the actual parameter. A value parameter would require storage for an extra copy of the entire data structure, and the time involved in copying it.

16.1.1. RELAXATIONS ON PARAMETER TYPE CHECKING

Normally, when using variable parameters, the formal and the actual parameters must match exactly. This means that subprograms employing variable parameters of type *String* will accept only strings of the exact length defined in the subprogram. This restriction may be overridden by the **V** compiler directive. The default active state {$V+} indicates strict type checking, whereas the passive state {$V-} relaxes the type checking and allows actual parameters of any string length to be passed, regardless of the length of the formal parameters.

Example:

```
program Encoder;
{$V-}
type
   WorkString = string[255];
var
   Line1: string[80];
   Line2: string[100];
procedure Encode(var LineToEncode: WorkString);
var I: Integer;
begin
   for I := 1 to Length(LineToEncode) do
      LineToEncode[I] := Chr(Ord(LineToEncode[I])-30);
end;
begin
   Line1 := 'This is a secret message';
   Encode(Line1);
   Line2 := 'Here is another (longer) secret message';
   Endcode(Line2);
end.
```

16.1.2. UNTYPED VARIABLE PARAMETERS

If the type of a formal parameter is not defined, i.e., the type definition is omitted from the parameter section of the subprogram heading, then that parameter is said to be ***untyped***. Thus, the corresponding actual parameter may be any type.

The untyped formal parameter itself is incompatible with all types, and may be used only in contexts where the data type is of no significance, for example as a parameter to *Addr*, *BlockRead*, *BlockWrite*, *FillCar*, or *Move*, or as the address specification of **absolute** variables.

The *SwitchVar* procedure in the following example demonstrates the use of untyped parameters. It moves the contents of the variable *A1* to *A2* and the contents of *A2* to *A1*.

```
procedure SwitchVar(var A1p,A2p; Size: Integer);
type
  A = array[1..MaxInt] of Byte;
var
  A1: A absolute A1p;
  A2: A absolute A2p;
  Tmp: Byte;
  Count: Integer;
begin
  for Count := 1 to Size do
  begin
    Tmp := A1[Count];
    A1[Count] := A2[Count];
    A2[Count] := Tmp;
  end;
end;
```

Assuming the declarations:

```
type
  Matrix = array[1..50,1..25] of Real;
var
  TestMatrix,BestMatrix: Matrix;
```

then *SwitchVar* may be used to switch values between the two matrices:

```
SwitchVar(TestMatrix,BestMatrix, SizeOf(Matrix));
```

16.2. PROCEDURES

A procedure may be either pre-declared (or "standard") or user-declared, i.e., declared by the programmer. Pre-declared procedures are parts of the TURBO Pascal system and may be called with no further declaration. A user-declared procedure may be given the name of a standard procedure; but that standard procedure then becomes inaccessible within the scope of the user-declared procedure.

16.2.1. PROCEDURE DECLARATION

A procedure declaration consists of a procedure heading followed by a block which consists of a declaration part and a statement part.

The procedure heading consists of the reserved word **procedure** followed by an identifier which becomes the name of the procedure, optionally followed by a formal parameter list as described on page 112.

Examples:

```
procedure LogOn;
procedure Position(X,Y: Integer);
procedure Compute(var Data: Matrix; Scale: Real);
```

The declaration part of a procedure has the same form as that of a program. All identifiers declared in the formal parameter list and the declaration part are local to that procedure, and any procedures within it. This is called the *scope* of an identifier, outside which they are not known. A procedure may reference any constant, type, variable, procedure, or function defined in an outer block.

The statement part specifies the action to be executed when the procedure is invoked, and it takes the form of a compound statement (see page 49). If the procedure identifier is used within the statement part of the procedure itself, the procedure will execute recursively. (**Note:** the **A** compiler directive must be passive {$A-} when recursion is used (see Appendix C.)

The next example shows a program which uses a procedure and passes a parameter to this procedure. As the actual parameter passed to the procedure is in some instances a constant (a simple expression), the formal parameter must be a value parameter.

```
program Box;
var
   I: Integer;
procedure DrawBox(X1,Y1,X2,Y2: Integer);
var I: Integer;
begin
   GotoXY(X1,Y1);
   for I := X1 to X2 do write ('-');
   for I := Y1+1 to Y2 do
   begin
      GotoXY(X1,I); Write('!');
      GotoXY(X2,I); Write('!');
   end;
   GotoXY(X1,Y2);
   for I := X1 to X2 do Write('-');
end; {of procedure DrawBox}
begin
   ClrScr;
   for I := 1 to 5 do DrawBox(I*4,I*2,10*I,4*I);
   DrawBox(1,1,80,23);
end.
```

Often the changes made to the formal parameters in the procedure should also affect the actual parameters. In such cases ***variable parameters*** are used, as in the following example:

```
procedure Switch(var A,B: Integer);
var Tmp: Integer;
begin
   Tmp := A; A := B; B := Tmp;
end;
```

When this procedure is called by the statement:

```
Switch(I,J);
```

the values of **I** and **J** will be switched. If the procedure heading in **Switch** was declared as:

```
procedure Switch(A,B: Integer);
```

i.e., with a *value* parameter, then the statement Switch(I,J) would **not** change *I* and *J*.

16.2.2. STANDARD PROCEDURES

TURBO Pascal contains a number of standard procedures. These are:

1) string-handling procedures (described on page 60 pp),
2) file-handling procedures (described on pages 82, 88, and 101),
3) procedures for allocation of dynamic variables (described on pages 106 and 111), and
4) input and output procedures (described on pages 95 pp).

In addition to these, the following standard procedures are available, provided that the associated commands have been installed for your terminal (see pages 6 pp):

16.2.2.1. CLREOL

Syntax: ClrEol;

Clears all characters from the cursor position to the end of the line without moving the cursor.

16.2.2.2. CLRSCR

Syntax: ClrScr;

Clears the screen and places the cursor in the upper left-hand corner. Beware that some screens also reset the video attributes when clearing the screen, possibly disturbing any user-set attributes.

16.2.2.3. CRTINIT

Syntax: CrtInit;

Sends the *Terminal Initialization String* defined in the installation procedure to the screen.

16.2.2.4. CRTEXIT

Syntax: CrtExit;

Sends the *Terminal Reset String* defined in the installation procedure to the screen.

16.2.2.5. DELAY

Syntax: Delay(*Time*);

The *Delay* procedure creates a loop which runs for approximately as many milliseconds as defined by its argument *Time*, which must be an integer. The exact time may vary somewhat in different operating environments.

16.2.2.6. DELLINE

Syntax: DelLine;

Deletes the line containing the cursor and moves all lines below one line up.

16.2.2.7. INSLINE

Syntax: InsLine;

Inserts an empty line at the cursor position. All lines below are moved one line down and the bottom line scrolls off the screen.

16.2.2.8. GOTOXY

Syntax: GotoXY(*Xpos,Ypos*);

Moves the cursor to the position on the screen specified by the integer expressions *Xpos* (horizontal value, or *row*) and *Ypos* (vertical value, or *column*). The upper left corner (home position) is (1,1).

16.2.2.9. EXIT

Syntax: Exit;

Exits the current block. When *Exit* is executed in a subroutine, it causes the subroutine to return. When it is executed in the statement part of a program, it causes the program to terminate. A call to *Exit* may be compared to a **goto** statement addressing a label just before the **end** of a block.

16.2.2.10. HALT

Syntax: Halt;

Stops program execution and returns to the operating system.

16.2.2.11. LOWVIDEO

Syntax: LowVideo;

Sets the screen to the video attribute defined as "Start of Low Video", i.e., "dim" characters, in the installation procedure.

16.2.2.12. NORMVIDEO

Syntax: NormVideo;

Sets the screen to the video attribute defined as "Start of Normal Video" in the installation procedure, i.e., the "normal" screen mode.

16.2.2.13. RANDOMIZE

Syntax: Randomize;

Initializes the random-number generator with a random value.

16.2.2.14. MOVE

Syntax: Move(*var1,var2,Num);*

Does a mass copy directly in memory of a specified number of bytes. *var1* and *var2* are two variables of any type, and *Num* is an integer expression. The procedure copies a block of *Num* bytes, starting at the first byte occupied by *var1*, to the block starting at the first byte occupied by *var2*. You may notice the absence of explicit "moveright" and "moveleft" procedures. This is because *Move* automatically handles possible overlap during the move process.

16.2.2.15. FILLCHAR

Syntax: FillChar(*Var,Num,Value*);

Fills a range of memory with a given value. *Var* is a variable of any type, *Num* is an integer expression, and *Value* is an expression of type *Byte* or *Char*. *Num* bytes, starting at the first byte occupied by *Var*, are filled with the value *Value*.

16.3. FUNCTIONS

Like procedures, functions are either standard (pre-declared) or declared by the programmer.

16.3.1. FUNCTION DECLARATION

A function declaration consists of a function *heading* and a *block*, which is a declaration part followed by a statement part.

The function heading is equivalent to the procedure header, except that the header must define the *type* of the function result. This is done by adding a colon and a type to the header as shown here:

```
function KeyHit: Boolean;
function Compute(var Value: Sample): Real;
function Power(X,Y: Real): Real;
```

The result type of a function must be a scalar type (i.e., *Integer, Real, Boolean, Char,* declared scalar or subrange), a **string** type, or a pointer type.

The declaration part of a function is the same as that of a procedure.

The statement part of a function is a compound statement as described on page 49. Within the statement part at least one statement assigning a value to the function identifier must occur. The last assignment executed determines the result of the function. If the function designator appears in the statement part of the function itself, the function will be invoked recursively. **Note:** the **A** compiler directive must be passive {$A-} when recursion is used (see Appendix C.)

The following example shows the use of a function to compute the sum of a row of integers from **I** to **J**.

```
function RowSum(I,J: Integer): Integer;
  function SimpleRowSum(S: Integer): Integer;
  begin
    SimpleRowSum := S*(S+1) div 2;
  end;
begin
  RowSum := SimpleRowSum(J) - SimpleRowSum(I-1);
end;
```

The function *SimpleRowSum* is nested within the function *RowSum*. *SimpleRowSum* is therefore only available within the scope of *RowSum*.

The following program is the classical demonstration of the use of a recursive function to calculate the factorial of an integer number:

```
{$A-} {A- directive allows recursion in 8-bit version}
program Factorial;
var Number: Integer;
function Factorial(Value: Integer): Real;
```

```
begin
  if Value = 0 then Factorial := 1
  else Factorial := Value * Factorial(Value-1);
end;
begin
  Read(Number);
  Writeln(^M,Number,'! = ',Factorial(Number));
end.
```

The type used in the definition of a function type must be previously specified as a *type declaration.* Thus, the construct:

```
function LowCase(Line: UserLine): string[80];
```

is **not** allowed. Instead, a type identifier should be associated with the type **string**[80], and that type identifier should then be used to define the function result type, for example:

```
type
  Str80 = string[80];

function LowCase(Line: UserLine): Str80;
```

Because of the implementation of the standard procedures *Write* and *Writeln,* a function using any of the standard procedures *Read, Readln, Write,* or *Writeln* must **never** be called by an expression within a *Write* or *Writeln* statement. In 8-bit systems this is also true for the standard procedures *Str* and *Val.*

16.3.2. STANDARD FUNCTIONS

The following standard (pre-declared) functions are implemented in TURBO Pascal:

1) string-handling functions (described on pages 62 pp).
2) file-handling functions (described on pages 85 and 88),
3) pointer-related functions (described on pages 106 and 111).

16.3.2.1. ARITHMETIC FUNCTIONS

16.3.2.1.1. ABS

Syntax: Abs(*Num*);

Returns the absolute value of *Num*. The argument *Num* must be either *Real* or *Integer*, and the result is of the same type as the argument.

16.3.2.1.2. ARCTAN

Syntax: ArcTan(*Num*);

Returns the angle, in radians, whose tangent is *Num*. The argument *Num* must be either *Real* or *Integer*, and the result is *Real*.

16.3.2.1.3. COS

Syntax: Cos(*Num*);

Returns the cosine of *Num*. The argument *Num* is expressed in radians, and its type must be either *Real* or *Integer*. The result is of type *Real*.

16.3.2.1.4. EXP

Syntax: Exp(*Num*);

Returns the exponential of *Num*, i.e., e^{Num}. The argument *Num* must be either *Real* or *Integer*, and the result is *Real*.

16.3.2.1.5. FRAC

Syntax: Frac(*Num*);

Returns the fractional part of *Num*, i.e., Frac(*Num*) = *Num* - Int(*Num*). The argument *Num* must be either *Real* or *Integer*, and the result is *Real*.

16.3.2.1.6. INT

Syntax: Int(*Num*);

Returns the integer part of *Num*, i.e., the greatest integer number less than or equal to *Num*, if *Num* >= 0, or the smallest integer number greater than or equal to *Num*, if *Num* < 0. The argument *Num* must be either *Real* or *Integer*, and the result is *Integer*.

16.3.2.1.7. LN

Syntax: Ln(*Num*);

Returns the natural logarithm of *Num*. The argument *Num* must be either *Real* or *Integer*, and the result is *Real*.

16.3.2.1.8. SIN

Syntax: Sin(*Num*);

Returns the sine of *Num*. The argument *Num* is expressed in radians, and its type must be either *Real* or *Integer*. The result is of type *Real*.

16.3.2.1.9. SQR

Syntax: Sqr(*Num*);

Returns the square of *Num*, i.e., *Num* * *Num*. The argument *Num* must be either *Real* or *Integer*, and the result is of the same type as the argument.

16.3.2.1.10. SQRT

Syntax: Sqrt(*Num*);

Returns the square root of *Num*. The argument *Num* must be either *Real* or *Integer*, and the result is *Real*.

16.3.2.2. SCALAR FUNCTIONS

16.3.2.2.1. PRED

Syntax: Pred(*Num*);

Returns the predecessor of *Num* (if it exists). *Num* is of any scalar type.

16.3.2.2.2. SUCC

Syntax: Succ(*Num*);

Returns the successor of *Num* (if it exists). *Num* is of any scalar type.

16.3.2.2.3. ODD

Syntax: Odd(*Num*);

Returns boolean *True* if *Num* is an odd number, and *False* if *Num* is even. *Num* must be of type *Integer*.

16.3.2.3. TRANSFER FUNCTIONS

The transfer functions are used to convert values of one scalar type to that of another scalar type. In addition to the following functions, the *retype* facility described on page 56 serves this purpose.

16.3.2.3.1. CHR

Syntax: Chr(*Num*);

Returns the character with the ordinal value given by the integer expression *Num*. Example: *Chr(65)* returns the character "A".

16.3.2.3.2. ORD

Syntax: Ord(*Var*);

Returns the ordinal number of the value *Var* in the set defined by the type *Var*. *Ord(Var)* is equivalent to *Integer(Var)* (see Type

Conversion on page 56). *Var* may be of any scalar type, except *Real*, and the result is of type *Integer*.

16.3.2.3.3. ROUND

Syntax: Round(*Num*);

Returns the value of *Num* rounded to the nearest integer as follows: if *Num* >= 0, then *Round(Num) = Trunc(Num + 0.5)*. If *Num* < 0, then *Round(Num) = Trunc(Num - 0.5)*. *Num* must be of type *Real*, and the result is of type *Integer*.

16.3.2.3.4. TRUNC

Syntax: Trunc(*Num*);

Returns the greatest integer less than or equal to *Num*, if *Num* >= 0, or the smallest integer greater than or equal to *Num*, if *Num* < 0. *Num* must be of type *Real*, and the result is of type *Integer*.

16.3.2.4. MISCELLANEOUS STANDARD FUNCTIONS

16.3.2.4.1. HI

Syntax: Hi(*I*);

The low order byte of the result contains the high order byte of the value of the integer expression *I*. The high order byte of the result is zero. The type of the result is *Integer*.

16.3.2.4.2. KEYPRESSED

Syntax: KeyPressed;

Returns boolean *True* if a key has been pressed at the console, and *False* if no key has been pressed. The result is obtained by calling the operating system console status routine.

16.3.2.4.3. LO

Syntax: Lo(*I*);

Returns the low order byte of the value of the integer expression *I* with the high order byte forced to zero. The type of the result is *Integer*.

16.3.2.4.4. RANDOM

Syntax: Random;

Returns a random number greater than or equal to zero and less than one. The type is *Real*.

16.3.2.4.5. RANDOM(NUM)

Syntax: Random(*Num*);

Returns a random number greater than or equal to zero and less than *Num*. *Num* and the random number are both *Integers*.

16.3.2.4.6. PARAMCOUNT

Syntax: ParamCount;

This integer function returns the number of parameters passed to the program in the command-line buffer. Space and tab characters serve as separators.

16.3.2.4.7. PARAMSTR

Syntax: ParamStr(*N*);

This string function returns the *Nth* parameter from the command-line buffer.

16.3.2.4.8. SIZEOF

Syntax: SizeOf(*Name*);

Returns the number of bytes occupied in memory by the variable or type *Name*. The result is of type *Integer*.

16.3.2.4.9. SWAP

Syntax: Swap(*Num*);

The Swap function exchanges the high and low order bytes of its integer argument *Num* and returns the resulting value as an integer.

Example:
Swap($1234) returns *$3412* (values in hex for clarity).

16.3.2.4.10. UPCASE

Syntax: UpCase(*ch*);

Returns the upper-case equivalent of its argument *ch*, which must be of type *Char*. If no upper-case equivalent exists, the argument is returned unchanged.

16.4. FORWARD REFERENCES

A subprogram is **forward** declared by specifying its header separately from the block. This separate subprogram header is exactly like the normal header, except that it ends with the reserved word **forward**. The block follows late within the same declaration part. Notice that the block is initiated by a copy of the header, specifying only the name and no parameters, types, etc.

```
Example:
program Catch 22;
var
   X: Integer;
function Up(Var I: Integer): Integer; forward;
function Down(Var I: Integer): Integer;
begin
   I := I div 2; Writeln(I);
   if I <> 1 then I := Up(I);
end;
function Up;
begin
   while I mod 2 <> 0 do
   begin
      I := I*3+1; Writeln(I);
   end;
   I := Down(I);
end;
begin
   Write('Enter any integer: ');
   Readln(X);
   X := Up(X);
   Write('Ok.  Program stopped again.');
end.
```

When the program is executed, if you enter 6, it returns:

```
3
10
5
16
8
4
2
1
Ok.  Program stopped again.
```

The above program is actually a more complicated version of the following program:

```
program Catch 222;
var
   X: Integer;
begin
   Write('Enter any integer: ');
   Readln(X);
   while X <> 1 do
   begin
      if X mod 2 = 0 then X := X div 2 else X := X*3+1;
      Writeln(X);
   end;
   Write('Ok.  Program stopped again.');
end.
```

It may interest you to know that it cannot be proved whether this small and very simple program actually **will** stop for any integer!

Chapter 17
INCLUDING FILES

The fact that the TURBO editor performs editing only within memory limits the size of source code handled by the editor. The **I** compiler directive can be used to circumvent this restriction, as it provides the ability to split the source code into smaller "lumps" and put it back together at compilation time. The include facility also aids program clarity, as commonly used subprograms, once tested and debugged, may be kept as a "library" of files from which the necessary files can be included in any other program.

The syntax for the **I** compiler directive is:

```
{$I filename}
```

Where *filename* is any legal file name. Leading spaces are ignored and lower case letters are translated to upper case. If no file type is specified, the default type **.PAS** is assumed. This directive must be specified on a line by itself.

Examples:

```
{$Ifirst.pas}
{$I COMPUTE.MOD}
{$iStdProc }
```

A space must be left between the file name and the closing brace if the file does not have a three-letter extension; otherwise the brace will be taken as part of the name.

To demonstrate the use of the include facility, let us assume that in your "library" of commonly used procedures and functions you have a file called STUPCASE.FUN. It contains the file *StUpCase* which is called with a character or a string as parameter and returns the value of this parameter with any lower case letters set to upper case.

File STUPCASE.FUN:

```
function StUpCase(St: AnyString): AnyString;
var I: Integer;
begin
```

```
  for I := 1 to Length(St) do
    St[I] := UpCase(St[I]);
  StUpCase := St
end;
```

In any future program you write which requires this function to convert strings to upper case letters, you need only include the file at compilation time instead of duplicating it into the source code:

```
program Include Demo;
type
  InData = string[80];
  AnyString = string[255];
var
  Answer: InData;
{$I STUPCASE.FUN}
begin
  ReadLn(Answer);
  Writeln(StUpCase(Answer));
end.
```

This method is not only easier and saves space; it also makes program updating quicker and safer, as any change to a "library" routine will automatically affect all programs including this routine.

Since TURBO Pascal allows free ordering, and even multiple occurrences, of the individual sections of the declaration part, you may have, for example, a number of files containing various commonly used **type** definitions in your "library" and include the ones required by different programs.

All compiler directives except **B** and **C** are local to the file in which they appear. That is, if a compiler directive is set to a different value in an included file, it is reset to its original value upon return to the including file. **B** and **C** directives are always global. Compiler directives are described in Appendix C.

Include files cannot be nested, i.e., one include file cannot include yet another file and then continue processing.

Chapter 18
OVERLAY SYSTEM

The overlay system lets you create programs much larger than can be accommodated by the computer's memory. The technique is to collect a number of subprograms (procedures and functions) in one or more files separate from the main program file, which will then be loaded automatically one at a time into the **same** area in memory.

The following drawing shows a program using one overlay file with five overlay subprograms collected into one **overlay group**, thus sharing the same memory space in the main program:

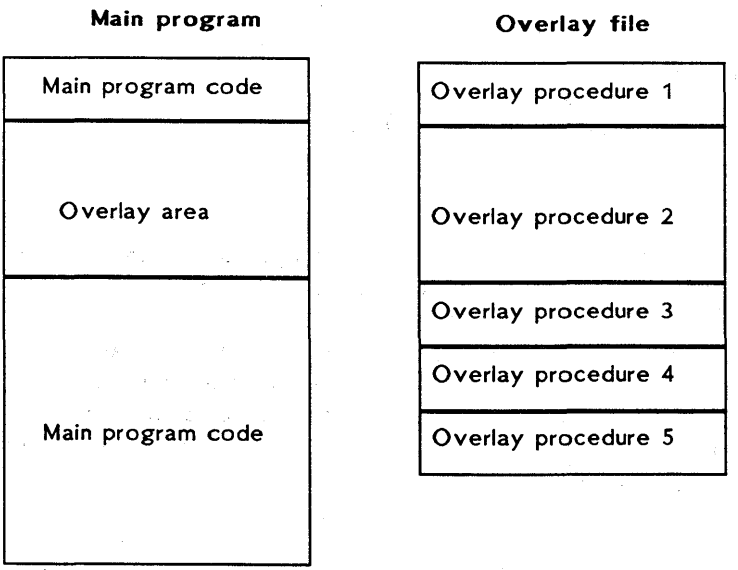


Figure 18-1: Principle of Overlay System

When an overlay procedure is called, it is automatically loaded into the overlay area reserved in the main program. This "gap" is large enough to accommodate the largest of the overlays in the group. The space required by the main program is thus reduced by roughly the sum of all subprograms in the group less the largest of them.

In the example above, overlay procedure 2 is the largest of the five procedures and thus determines the size of the overlay area in the main code. When it is loaded into memory, it occupies the entire overlay area:

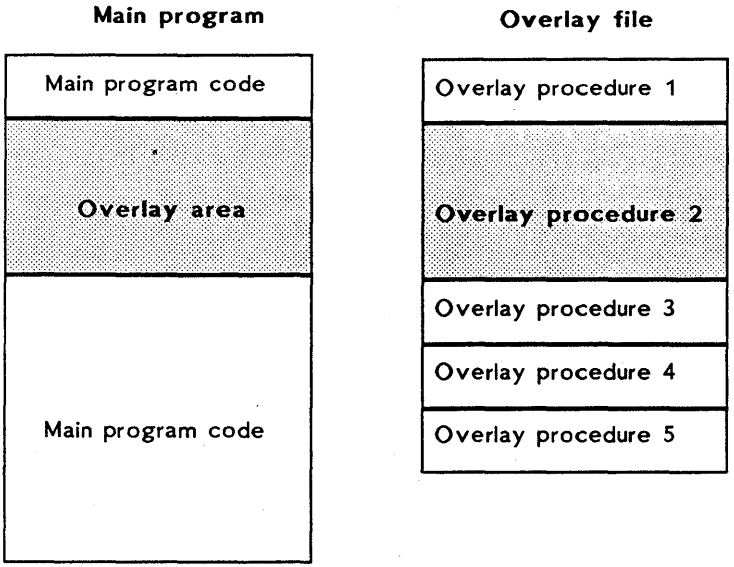


Figure 18-2: Largest Overlay Subprogram Loaded

The smaller subprograms are loaded into the same area of memory, each starting at the first address of the overlay area. Obviously they occupy only part of the overlay area; the remainder is unused:

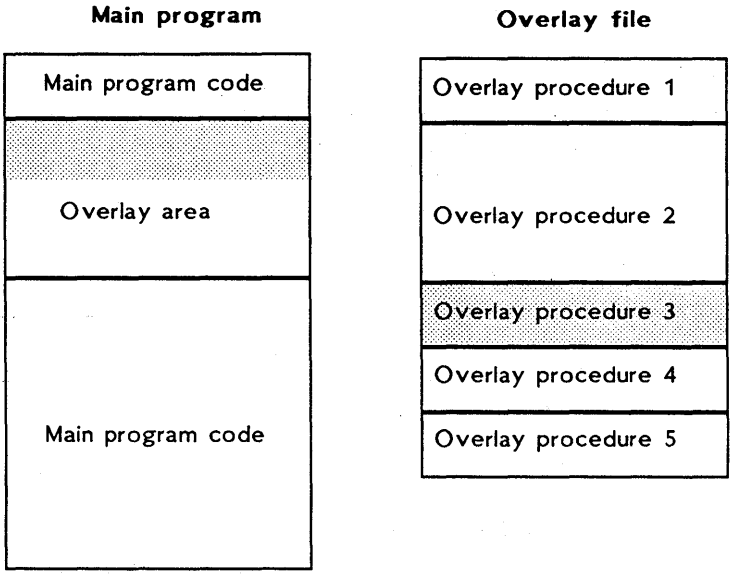


Figure 18-3: Smaller Overlay Subprogram Loaded

As procedures 1, 3, 4, and 5 execute in the same space as used by procedure 2, it is clear that they require no additional space in the main program. It is also clear that none of these procedures must ever call each other, as they are never present in memory simultaneously.

There could be many more overlay procedures in this group of overlays; in fact the total size of the overlay procedures could substantially exceed the size of the main program. And they would still require only the space occupied by the largest of them.

The tradeoff for this extra room for program code is the addition of disk access time each time a procedure is read in from the disk. With good planning, as discussed on page 141, this time is negligible.

18.1. CREATING OVERLAYS

Overlay subprograms are created automatically, simply by adding the reserved word **overlay** to the declaration of any procedure or file:

```
overlay procedure Initalize;
and
overlay function TimeOfDay: Time;
```

When the compiler meets such a declaration, code is no longer output to the main program file, but to a separate overlay file. The name of this file will be the same as that of the main program, and the type will be a number designating the overlay group, ranging from 000 through 099.

Consecutive overlay subprograms will be grouped together, i.e., as long as overlay subprograms are not separated by any other declaration, they belong to the same group and are placed in the same overlay file.

```
Example 1:
overlay procedure One;
begin
  :
end;

overlay procedure Two;
begin
  :
end;

overlay procedure Three;
begin
  :
end;
```

These three overlay procedures will be grouped together and placed in the same overlay file. If they are the first group of overlay subprograms in a program, the overlay file will be number *000*.

The three overlay procedures in the following example will be placed in consecutive overlay files, *.000* and *.001,* because of the declaration of a non-overlay procedure *Count* separating overlay procedures *Two* and *Three.*

The separating declaration may be any declaration, for example a dummy **type** declaration, if you want to force a separation of overlay areas.

Example 2:

```
overlay procedure One;
begin
  :
end;

overlay procedure Two;
begin
  :
end;

procedure Count;
begin
  :
end;

overlay procedure Three;
begin
  :
end;
```

A separate overlay area is reserved in the main program code for each overlay group, and the following files will be created:

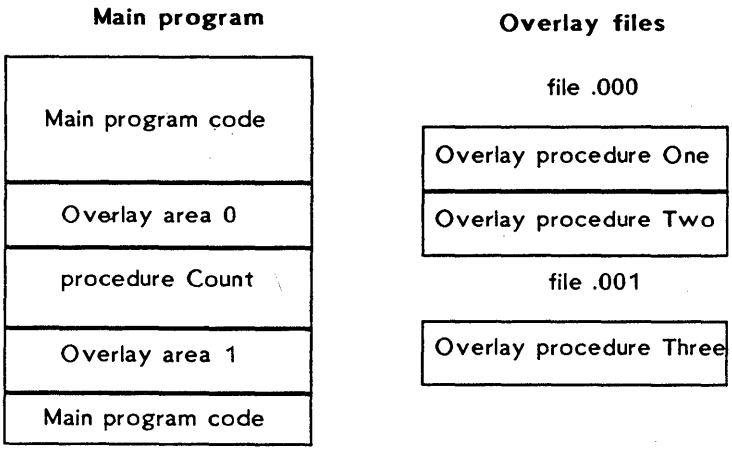


Figure 18-4: Multiple Overlay Files

18.2. NESTED OVERLAYS

Overlay subprograms may be nested, i.e., an overlay subprogram may itself contain overlay subprograms which may contain overlay subprograms, etc.

```
Example 3:
program OverlayDemo;
  :
  :
overlay procedure One;
begin
  :
end;
```

```
overlay procedure Two;
  overlay procedure Three;
  begin
    :
  end;
begin
  :
end;
  :
```

In this example, two overlay files will be created. File *.000* contains overlay procedures *One* and *Two*, and an overlay area is reserved in the main program to accommodate the larger of these. Overlay file *.001* contains overlay procedure *Three*, which is local to overlay procedure *Two*, and an overlay area is created in the code of overlay procedure *Two*:

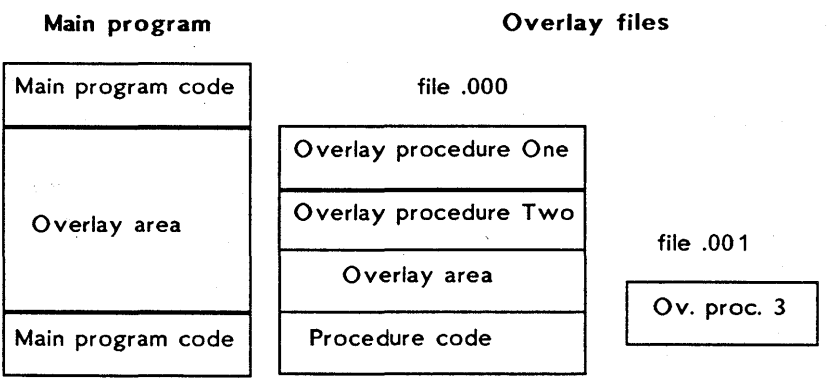


Figure 18-5: Nested Overlay Files

18.3. AUTOMATIC OVERLAY MANAGEMENT

An overlay subprogram is loaded into memory only when called. On each call to an overlay subprogram, a check is first made to see whether that subprogram is already present in the overlay area. If not, it will automatically be read in from the appropriate overlay file.

18.4. PLACING OVERLAY FILES

During compilation, overlay files will be placed on the logged drive, i.e., on the same drive as the main program file (.COM file).

During execution, the system normally expects to find its overlay files on the logged drive. This may be changed as described on page 149.

18.5. EFFICIENT USE OF OVERLAYS

The overlay technique adds overhead to a program by adding some extra code to manage the overlays, and by requiring disk reads during execution. Overlays, therefore, should be carefully planned.

To avoid slowing down execution too much, an overlay subprogram should not be called too often. If one is called often, it should at least be called without intervening calls to other subprograms in the same overlay file, to keep disk access to a minimum. The added time will of course vary greatly, depending on the actual disk configuration. A 5¼" floppy will add much to the running time, a hard disk much less, and a RAM disk, as used by many, very little.

To save as much space as possible in the main program, one group of overlays should contain as many individual subprograms as possible. Purely from the point of view of saving space, the more subprograms you can put into a single overlay file, the better. The overlay space used in the main program need only accommodate the largest of these subprograms; the rest have a free ride in the same area of memory. This must be weighed against the time considerations discussed above.

18.6. RESTRICTIONS IMPOSED ON OVERLAYS

18.6.1. DATA AREA

Overlay subprograms in the same group share the same area in memory and thus **cannot** be present simultaneously. Therefore, they must **not** call each other. Consequently, they may share the same data area which further adds to the space saved when using overlays.

In example 1 on page 134, none of the procedures may call each other. In example 2, however, overlay procedures *One* and *Two* may

call overlay procedure *Three*, and overlay procedure *Three* may call each of the other two, because they are in separate files and consequently in separate overlay areas in the main program.

18.6.2. FORWARD DECLARATIONS

Overlay subprograms may not be **forward** declared. This restriction is easily circumvented, however, by **forward** declaring an ordinary subprogram which then in turn calls the overlay subprogram.

18.6.3. RECURSION

Overlay subprograms cannot be recursive. This restriction may be circumvented by declaring an ordinary recursive subprogram which then calls the overlay subprogram.

18.6.4. RUN-TIME ERRORS

Run-time errors occurring in overlays are found as usual, and an address is issued by the error handling system. This address, however, is an address within the overlay area, and there is no way of knowing which overlay subprogram was actually active when the error occurred.

For this reason, run-time errors in overlays can't be found readily with the Options menu's "Find run-time error" facility. What "Find run-time error" will point out is the first occurrence of code at the specified address. This **may** be the place of the error, but the error might just as easily occur in a subsequent subprogram within the same overlay group.

This is not a serious limitation, however, because the type of error and the way it occurs will most often indicate in which subprogram the error happened. The way to locate the error precisely is then to place the suspected subprogram as the first subprogram of the overlay group. "Find run-time error" will then work.

The best thing to do is not to place subprograms in overlays until they have been fully debugged!

Chapter 22
CP/M-80

This chapter describes features of TURBO Pascal specific to the 8-bit CP/M-80 implementation. It presents two kinds of information:

1) Things you should know to make efficient use of TURBO Pascal. Pages 143 through 155.

2) The rest of the chapter describes things which are only of interest to experienced programmers, such as machine language routines, technical aspects of the compiler, etc.

22.1. EXECUTE COMMAND

You will find an additional command on the main TURBO menu in the CP/M-80 version: e**X**ecute. It lets you run other programs from within TURBO Pascal, for example copying programs, word processors – in fact anything that you can run from your operating system. When entering **X**, you are prompted:

Command: ■

You may now enter the name of any program, which will then load and run normally. Upon exit from the program, control is transferred back to TURBO Pascal, and you return to the TURBO prompt > .

22.2. COMPILER OPTIONS

The **O** command selects the following menu, on which you may view and change some default values of the compiler. It also provides a helpful function to find run-time errors in programs compiled into object code files.

```
compile Memory
        Com-file
        cHn-file

command line Parameter:

Find run-time error    Quit
```

Figure 22-1: Options Menu

22.2.1. MEMORY/COM FILE/CHN FILE

The three commands **M**, **C**, and **H** select the compiler mode, i.e., where to put the code which results from the compilation.

Memory is the default mode. When active, code is produced in memory and resides there ready to be activated by a **R**un command.

Com-file is selected by pressing **C**. The arrow moves to point to this line. When active, code is written to a file with the same name as the Work file (or Main file, if specified) and the file type .COM. This file contains the program code and Pascal run-time library, and may be activated by typing its name at the console. Programs compiled this way may be larger than programs compiled in memory, as the program code itself does not take up memory during compilation, and as program code starts at a lower address.

c**H**ain-file is selected by pressing **H**. The arrow moves to point to this line. When active, code is written to a file with the same name as the Work file (or Main file, if specified) and the file type .CHN. This file contains the program code but no Pascal library and must be activated from another TURBO Pascal program with the *Chain* procedure (see page 146).

When **C**om or c**H**n mode is selected, the menu is expanded with the following two lines:

```
Start address:  XXXX  (min YYYY)
End   address:  XXXX  (max YYYY)
```

Figure 22-2: Start and End Addresses

22.2.2. START ADDRESS

The Start address specifies the address (in hexadecimal) of the first byte of the code. This is normally the end address of the Pascal library plus one, but may be changed to a higher address if you want to set space aside, e.g., for absolute variables to be shared by a series of chained programs.

When you enter an **S**, you are prompted to enter a new Start address. If you just hit <RETURN>, the minimum value is assumed. Don't set the Start address to anything less than the minimum value, as the code will then overwrite part of the Pascal library.

22.2.3. END ADDRESS

The End address specifies the highest address available to the program (in hexadecimal). The value in parentheses indicates the top of the TPA on your computer, i.e., BDOS minus one. The default setting is 700 to 1000 bytes less to allow space for the loader, which resides just below the BDOS when executing programs from TURBO.

If compiled programs are to run in a different environment, the **E**nd address may be changed to suit the TPA size of the system. If you anticipate your programs to run on a range of different computers, it will be wise to set this value relatively low, e.g., C100 (48K), or even A100 (40K) if the program is to run under MP/M.

When you enter an **E**, you are prompted to enter an End address. If you just hit <RETURN>, the default value is assumed (i.e., top of TPA less 700 to 1000 bytes). If you set the **E**nd address higher than this, the resulting programs cannot be executed from TURBO, as they will overwrite the TURBO loader; and if you set it higher than the top of TPA, the resulting programs will overwrite part of your BDOS if run on yur machine.

22.2.4. COMMAND LINE PARAMETERS

The **P** command lets you enter one or more parameters which are passed to your program when running it in **M**emory mode, just as if they had been entered on the command line. These parameters may be accessed through the *ParamCount* and *ParamStr* functions.

22.2.5. FIND RUNTIME ERROR

When you run a program compiled in memory, and a run-time error occurs, the editor is invoked, and the error is automatically pointed out. This, of course, is not possible if the program is in a .COM file or a .CHN file. When a run-time error is found, TURBO prints out the error code and the value of the program counter at the time of the error, e.g.:

```
Run-time error 01, PC=1B56
Program aborted
```

Figure 22-3: Run-time Error Message

To find the place in the source text where the error occurred, enter the **F** command on the **O**ptions menu. When prompted for the address, enter the address given by the error message:

```
Enter PC: 1B56
```

Figure 22-4: Find Run-time Error

The place in the source text is now found and pointed out exactly as if the error had occurred while running the program in memory.

22.3. STANDARD IDENTIFIERS

The following standard identifiers are unique to the CP/M-80 implementation:

Bios	Bdos	RecurPtr
BiosHL	BdosHL	StackPtr

22.4. CHAIN AND EXECUTE

TURBO Pascal provides two standard procedures, *Chain* and *Execute*, which allow you to activate other programs from a TURBO program. The syntax of these procedure calls is:

```
Chain(FilVar)
Execute(FilVar)
```

where *FilVar* is a file variable of any type, previously assigned to a disk file with the standard procedure *Assign*. If the file exists, it is loaded into memory and executed.

The *Chain* procedure is used only to activate special TURBO Pascal .CHN files, i.e., files compiled with the CHN-file option selected on the Options menu (see page 144). Such a file contains only program code, no Pascal library. It is loaded into memory and executed at the start address of the current program, i.e., the address specified when the current program was compiled. It then uses the Pascal library already present in memory. Thus, the current program and the chained program must use the same start address.

The *Execute* procedure may be used to execute any .COM file, i.e., any file containing executable code. This could be a file created by TURBO Pascal with the COM-option selected on the Options menu (see page 144). The file is loaded and executed at address $100, as specified by the CP/M standard.

If the disk file does not exist, an I/O error occurs. This error is treated as described on page 103. If the I compiler directive is passive ({$I-}), program execution continues with the statement following the failed *Chain* or *Execute* statement, and the *IOresult* function must be called prior to further I/O.

Data can be transferred from the current program to the chained program either by ***shared global variables*** or by ***absolute address variables***.

To ensure overlapping, shared global variables should be declared as the very first variables in both programs, and they must be listed in the same order in both declarations. Furthermore, both programs must be compiled to the same memory size (see page 145). When these conditions are satisfied, the variables will be placed at the same address in memory by both programs, and as TURBO Pascal does not automatically initialize its variables, they may be shared.

Example:
Program MAIN.COM:

```
program Main;
var
  Txt: string[80];
  CntPrg: file;
begin
  Write('Enter any text: '); Readln(Txt);
  Assign(CntPrg, 'ChrCount.Chn');
  Chain(CntPrg);
end.
```

Program CHRCOUNT.CHN:

```
program ChrCount;
var
  Txt: string[80];
  NoOfChar, NoOfUpc, I: Integer;
begin
  NoOfUpc := 0;
  NoOfChar := Length(Txt);
  for I := 1 to length(Txt) do
    if Txt[I] in ['A'..'Z'] then NoOfUpc :=
Succ(NoOfUpc);
  Write('No of characters in entry: '; NoOfChar);
  Writeln('. No of upper-case characters: ', NoOfUpc, '.');
end.
```

If you want a TURBO program to determine whether it was invoked by eXecute or directly from the command line, you should use an **absolute** variable at address *$80*. This is the command line length byte, and when a program is called from CP/M, it contains a value between 0 and 127. When eXecuting a program, therefore, the calling program should set this variable to something higher than 127. When you then check the variable in the called program, a value between 0 and 127 indicates that the program was called from CP/M, a higher value that it was called from another TURBO program.

Note: neither *Chain* nor *Execute* can be used in direct mode, i.e., from a program run with the compiler options switch in position **M**emory (page 144).

22.5. OVERLAYS

During execution, the system normally expects to find its overlay files on the logged drive. The *OvrDrive* procedure may be used to change this default value.

22.5.1. OVRDRIVE PROCEDURE

Syntax OvrDrive(*Drive*)

where *Drive* is an integer expression specifying a drive (0 = the logged drive, 1 = A:, 2 = B:, etc.). On subsequent calls to overlay files, the files will be expected on the specified drive. Once an overlay file has been opened on one drive, future calls to the same file will look the same drive.

```
Example:
program  OvrTest;

overlay procedure ProcA;
begin
  Writeln('Overlay A');
end;

overlay procedure ProcB;
begin
  Writeln('Overlay B');
end;

procedure Dummy;
begin
  {Dummy procedure to separate the overlays
   into two groups}
end;

overlay procedure ProcC;
begin
  Writeln('Overlay C');
end;
```

```
begin
   OvrDrive(2);
   ProcA;
   OvrDrive(0);
   ProcC;
   OvrDrive(2);
   ProcB;
end;
```

The first call to *OvrDrive* specifies overlays to be sought on the B: drive. The call to *ProcA* therefore causes the first overlay file (containing the two overlay porcedures *ProcA* and *ProcB*) to be opened here.

Next, the *OvrDrive(0)* statement specifies that following overlays are to be found on the logged drive. The call to *ProcC* opens the second overlay file here.

The following *ProcB* statement calls an overlay procedure in the first overlay file; and to ensure that it is sought on the B: drive, the *OvrDrive(2)* statement must be executed before the call.

22.6. FILES

22.6.1. FILE NAMES

A file name in CP/M consists of one through eight letters or digits, optionally followed by a period and a file type of one through three letters or digits:

Drive:Name.Type

22.6.2. TEXT FILES

The *Seek* and *Flush* procedures and the *FilePos* and *FileSize* functions are not applicable to CP/M text files.

22.7. ABSOLUTE VARIABLES

Variables may be declared to reside at specific memory addresses, and are then called **absolute**. This is done by adding the reserved

word **absolute** and an address expressed by an integer constant to the variable declaration.

```
Example:
var
IObyte: Byte absolute $0003;
CmdLine: string[127] absolute $80;
```

Absolute may also be used to declare a variable "on top" of another variable, i.e., that a variable should start at the same address as another variable. When **absolute** is followed by the variable (or parameter) identifier, the new variable will start at the address of that variable (or parameter).

```
Example:
var
Str: string[32];
StrLen: Byte absolute Str;
```

The above declaration specifies that the variable *StrLen* should start at the same address as the variable *Str*, and since the first byte of a string variable gives the length of the string, *StrLen* will contain the length of *Str*. Notice that only one identifier may be specified in an **absolute** declaration, so the construct:

```
Ident1, Ident2: Integer absolute $8000
```

is **illegal**. Further details on space allocation for variables are given on pages 159 and 169.

22.8. ADDR FUNCTION

Syntax: Addr(Name);

Returns the address in memory of the first byte of the type, varialbe, procedure, or function with the identifier *Name*. If *Name* is an array, it may be subscripted, and if *Name* is a record, specific fields may be selected. The value returned is of type *Integer*.

22.9. PREDEFINED ARRAYS

TURBO Pascal offers two predefined arrays of type *Byte*, called *Mem* and *Port*, which are used to access CPU memory and data ports directly.

22.9.1. MEM ARRAY

The predeclared array *Mem* is used to access memory. Each component of the array is a *Byte*, and indexes correspond to addresses in memory. The index type is *Integer*. When a value is assigned to a component of *Mem*, it is stored at the address given by the index expression. When the *Mem* array is used in an expression, the byte at the address specified by the index is used.

Examples:

```
Mem[WsCursor] := 2;
Mem[WsCursor+1] := $1B;
Mem[WsCursor+2] := Ord(' ');
IObyte := Mem[3];
Mem[Addr+Offset] := Mem[Addr];
```

22.9.2. PORT ARRAY

The *Port* array is used to access the data ports of the Z-80 CPU. Each element of the array represents a data port with indexes corresponding to port numbers. As data ports are selected by 8-bit addresses, the index type is *Byte*. When a value is assigned to a component of *Port*, it is output to the port specified. When a component of *Port* is referenced in an expression, its value is input from the port specified.

The use of the port array is restricted to assignment and reference in expressions only, i.e., components of *Port* cannot function as variable parameters to procedures and functions. Furthermore, operations referring to the entire *Port* array (reference without index) are not allowed.

22.10. ARRAY SUBSCRIPT OPTIMIZATION

The **X** compiler directive allows the programmer to select whether array subscription should be optimized with regard to execution speed or to code size. The default mode is active, i.e., {$X+}, which causes execution speed optimization. When passive, i.e., {$X-}, the code size is minimized.

22.11. WITH STATEMENTS

The default "depth" of nesting of *With* statements is 2, but the **W** directive may be used to change this value to between 0 and 9. For each block, *With* statements require two bytes of storage for each nesting level allowed. Keeping the nesting to a minimum may thus greatly affect the size of the data area in programs with many subprograms.

22.12. POINTER-RELATED ITEMS

22.12.1. MEMAVAIL

The standard function *MemAvail* is available to determine the available space on the heap at any given time. The result is an *Integer*, and if more than 32767 bytes are available, *MemAvail* returns a negative number. The correct number of free bytes is then calculated as 65536.0 + *MemAvail*. Notice the use of a real constant to generate a *Real* result, as the result is greater than *MaxInt*. Memory management is discussed in futher detail on page 169.

22.12.2. POINTERS AND INTEGERS

The standard functions *Ord* and *Ptr* provide direct control of the address contained in a pointer. *Ord* returns the address contained in its pointer argument as an *Integer*, and *Ptr* converts its *Integer* argument into a pointer which is compatible with all pointer types.

These functions are extremely valuable in the hands of an experienced programmer as they allow a pointer to point to anywhere in memory. If used carelessly, however, they are very dangerous, as a dynamic variable may be made to overwrite other variables, or even program code.

22.13. CP/M FUNCTION CALLS

For the purpose of calling CP/M BDOS and BIOS routines, TURBO Pascal introduces two standard procedures, *Bdos* and *Bios*, and four standard functions *Bdos, BdosHL, Bios,* and *BiosHL.*

Details on BDOS and BIOS routines are found in the *CP/M Operating System Manual* published by Digital Research.

22.13.1. BDOS PROCEDURE AND FUNCTION

Syntax: Bdos(*Func*[,*Param*]);

The *Bdos* **procedure** is used to invoke CP/M BDOS routines. *Func* and *Param* are integer expressions. *Func* denotes the number of the called routine, and is loaded into the C register. *Param* is optional, and denotes a parameter which is loaded into the DE register pair. A call to address 5 then invokes the BDOS.

The *Bdos* **function** is called like the procedure, and returns an *Integer* result which is the value returned by the BDOS in the A register.

22.13.2. BDOSHL FUNCTION

Syntax: BdosHL(*Func*[,*Param*]);

This function is exactly similar to the *Bdos* function above, except that the result is the value returned in the HL register pair.

22.13.3. BIOS PROCEDURE AND FUNCTION

Syntax: Bios(*Func*[,*Param*]);

The *Bios* **procedure** is used to invoke BIOS routines. *Func* and *Param* are integer expressions. *Func* denotes the number of the called routine, with 0 meaning the WBOOT routine, 1 the CONST routine, etc. That is, the address of the called routine is *Func**3 plus the WBOOT address contained in addresses 1 and 2. *Param* is optional and denotes a parameter which is loaded into the BC register pair before the call.

The *Bios* **function** is called like the procedure and returns an integer result which is the value returned by the BIOS in the A register.

22.13.4. BIOSHL FUNCTION

Syntax: BiosHL(*Func*[,*Param*]);

This function is exactly similar to the *Bios* function above, except that the result is the value returned in the HL register pair.

22.14. USER-WRITTEN I/O DRIVERS

For some applications it is practical for a programmer to define his own input and output drivers, i.e., routines which perform input and output of characters to and from external devices. The following drivers are part of the TURBO environment, and used by the standard I/O drivers (although they are not available as standard procedures or functions):

function	*ConSt:* boolean;
function	*ConIn:* Char;
procedure	*ConOut*(Ch: Char);
procedure	*LstOut*(Ch: Char);
procedure	*AuxOut*(Ch: Char);
function	*AuxIn:* Char;
procedure	*UsrOut*(Ch: Char);
function	*UsrIn:* Char;

The *ConSt* routine is called by the function *KeyPressed*, the *ConIn* and *ConOut* routines are used by the CON:, TRM:, and KBD: devices, the *LstOut* routine is used by the LST: device, the *AuxOut* and *AuxIn* routines are used by the AUX: device, and the *UsrOut* and *UsrIn* routines are used by the USR: device.

By default, these drivers use the corresponding BIOS entry points of the CP/M operating system, i.e., *ConSt* uses CONST, *ConIn* uses CONIN, *ConOut* uses CONOUT, *LstOut* uses LIST, *AuxOut* uses PUNCH, *AuxIn* uses READER, *UsrOut* uses CONOUT, and *UsrIn* uses CONIN. This, however, may be changed by the programmer by assigning the address of a driver procedure or function defined by the programmer to one of the following standard variables:

Variable	Contains the address of the
ConStPtr	*ConSt* function
ConInPtr	*ConIn* function
ConOutPtr	*ConOut* procedure
LstOutPtr	*LstOut* procedure
AuxOutPtr	*AuxOut* procedure
AuxInPtr	*AuxIn* function
UsrOutPtr	*UsrOut* procedure
UsrInPtr	*UsrIn* function

A user-defined driver procedure or driver function must match the definitions given above. That is, a *ConSt* driver must be a *Boolean* function, a *ConIn* driver must be a *Char* function, etc.

22.15. EXTERNAL SUBPROGRAMS

The reserved word **external** is used to declare external procedures and functions, typically procedures and functions written in machine code.

An external subprogram has no *block*, i.e., no declaration part and no statement part. Only the subprogram heading is specified, immediately followed by the reserved word **external** and an integer constant defining the memory address of the subprogram:

```
procedure DiskReset; external $EC00;
function IOstatus: boolean; external $D123
```

Parameters may be passed to external subprograms, and the syntax is exactly the same as that of calls to ordinary procedures and functions:

```
procedure Plot(X,Y: Integer); external $F003;
procedure QuickSort(var List: PartNo); external $1C00;
```

Parameter passing to external subprograms is discussed further on page 165.

22.16. IN-LINE MACHINE CODE

TURBO Pascal features the **inline** statement as a very convenient way of inserting machine-code instructions directly into the program text. An inline statement consists of the reserved word **inline** followed by one or more *code elements* separated by slashes and enclosed in parentheses.

A code element is built from one or more data elements, separated by plus (+) or minus (-) signs. A data element is either an integer constant, a variable identifier, a procedure identifier, a function identifier, or a location counter reference. A location counter reference is written as an asterisk (*).

Example:
inline (10/$2345/count+1/sort-*+2);

Each code element generates one byte or one word (two bytes) of code. The value of the byte or the word is calculated by adding or subtracting the values of the data elements according to the signs that separate them. The value of a variable identifier is the address (or offset) of the variable. The value of a procedure or function identifier is the address (or offset) of the procedure or function. The value of a location counter reference is the address (or offset) of the location counter, i.e., the address at which to generate the next byte of code.

A code element will generate one byte of code if it consists of integer constants only, and if its value is within the 8-bit range (0..255). If the value is outside the 8-bit range, or if the code element refers to variable, procedure, or function identifiers, or if the code element contains a location counter reference, one word of code is generated (least significant byte first).

The '<' and '>' characters may be used to override the automatic size selection described above. If a code element starts with a '<' character, only the least significant byte of the value is coded, even if it is a 16-bit value. If a code element starts with a '>' character, a word is always coded, even though the most significant byte is zero.

Example:
inline (<$1234/>$44);

This **inline** statement generates three bytes of code: $34, $44, $00.

The following example of an inline statement generates machine code that will convert all characters in its string arugment to upper case.

```
procedure UpperCase (var Strg: Str); {Str is type string[255]}
{$A+}
begin
  inline (  $2A/Strg/  {      LD    HL,(Strg)   }
            $46/       {      LD    B,(HL)      }
            $04/       {      INC   B           }
            $05/       { L1:  DEC   B           }
            $CA/*+20/  {      JP    Z, L2       }
            $23/       {      INC   HL          }
            $7E/       {      LD    A,(HL)      }
            $FE/$61/   {      CP    'a'         }
            $DA/*-9/   {      JP    C,L1        }
            $FE/$7B/   {      CP    'z'+1       }
            $D2/*-14/  {      JP    NC,L1       }
            $D6/$20/   {      SUB   20H         }
            $77/       {      LD    (HL),A      }
            $C3/*-20); {      JP    L1          }
                       { L2:  EQU   $           }
end;
```

Inline statements may be freely mixed with other statements throughout the statement part of a block, and **inline** statements may use all CPU registers. **Note**, however, that the contents of the stack pointer register (SP) must be the same on exit as on entry.

22.17. INTERRUPT HANDLING

The TURBO Pascal run-time package and the code generated by the compiler are both fully interruptable. Interrupt service routines must preserve all registers used.

If required, interrupt service procedures may be written in Pascal. Such procedures should always be compiled with the **A** compiler directive active ({$A+}), they must not have parameters, and they must themselves insure that all registers used are preserved. This is done by placing an **inline** statement with the necessary PUSH

instructions at the very beginning of the procedure, and another **inline** statement with the corresponding POP instructions at the very end of the procedure. The last instruction of the ending **inline** statement should be an EI instruction ($FB) to enable further interrupts. If daisy-chained interrupts are used, the **inline** statement may also specify a RETI instruction ($ED, $4D), which will override the RET instruction generated by the compiler.

The general rules for register usage are that integer operations use only the AF, BC, DE, and HL registers (other operations may use IX and IY), and real operations use the alternate registers.

An interrupt service procedure should not employ any I/O operations using the standard procedures and functions of TURBO Pascal, as these routines are not re-entrant. Also note that BDOS calls (and in some instances BIOS calls, depending on the specific CP/M implementaion) should not be performed from interrupt handlers, as these routines are not re-entrant.

The programmer may disable and enable interrupts throughout a program using DI and EI instructions generated by **inline** statements.

If mode 0 (IM 0) or mode 1 (IM 1) interrupts are employed, it is the responsibility of the programmer to initialize the restart locations in the base page (note that RST 0 cannot be used, as CP/M uses locations 0 through 7).

If mode 2 (IM 2) interrupts are employed, the programmer should generate an initialized jump table (an array of integers) at an absolute address, and initialize the I register through a **inline** statement at the beginning of the program.

22.18. INTERNAL DATA FORMATS

In the following descriptions, the symbol @ denotes the address of the first byte occupied by a variable of the given type. The standard function *Addr* may be used to obtain this value for any variable.

22.18.1. BASIC DATA TYPES

The basic data types may be grouped into structures (arrays, records, and disk files), but this structuring will not affect their internal formats.

22.18.1.1. SCALARS

The following scalars are all stored in a single byte: *Integer* subranges with both bounds in the range 0..255, *Booleans*, *Chars*, and declared scalars with less than 256 possible values. This byte contains the ordinal value of the variable.

The following scalars are all stored in two bytes: *Integers*, *Integer* subranges with one or both bounds not within the range 0..255, and declared scalars with more than 256 possible values. These bytes contain a 2's complement 16-bit value with the least significant byte stored first.

22.18.1.2. REALS

Reals occupy 6 bytes, giving a floating point value with a 40-bit mantissa and an 8-bit 2's exponent. The exponent is stored in the first byte and the mantissa in the next five bytes with the least significant byte first:

```
@       Exponent
@ + 1   LSB of mantissa
:
@ + 5   MSB of mantissa
```

The exponent uses binary format with an offset of $80. Hence, an exponent of $84 indicates that the value of the mantissa is to be multiplied by 2^($84-$80) = 2^4 = 16. If the exponent is zero, the floating point value is considered to be zero.

The value of the mantissa is obtained by dividing the 40-bit unsigned integer by 2^40. The mantissa is always normalized, i.e., the most significant bit (bit 7 of the fifth byte) should be interpreted as a 1. The sign of the mantissa is stored in this bit, a 1 indicating that the number is negative, and a 0 indicating that the number is positive.

22.18.1.3. STRINGS

A string occupies the number of bytes corresponding to one plus the maximum length of the string. The first byte contains the current length of the string. The following bytes contain the actual characters, with the first character stored at the lowest address. In the table shown below, **L** denotes the current length of the string, and *Max* denotes the maximum length:

@	Current length(L)
@ + 1	First character
@ + 2	Second character
:	
@ + L	Last character
@ + L + 1	Unused
:	
@ + *Max*	Unused

22.18.1.4. SETS

An element in a **set** occupies one bit, and as the maximum number of elements in a set is 256, a set variable will never occupy more than 32 bytes (256/8).

If a set contains less than 256 elements, some of the bits are bound to be zero at all times and need therefore not be stored. In terms of memory efficiency, the best way to store a set variable of a given type would then be to "cut off" all insignificant bits, and rotate the remaining bits so that the first element of the set would occupy the first bit of the first byte. Such rotate operations, however, are quite slow, and TURBO therefore employs a compromise: only bytes which are statically zero (i.e., bytes of which no bits are used) are not stored. This method of compression is very fast and in most cases as memory-efficient as the rotation method.

The number of bytes occupied by a set variable is calculated as (*Max* **div** 8) - (*Min* **div** 8) + 1, where *Max* and *Min* are the upper and lower bounds of the base type of that set. The memory address of a specific element *E* is:

MemAddress = @ + (*E* div 8) - (*Min* div 8)

and the bit address within the byte at *MemAddress* is:

BitAddress = *E* **mod** 8

where *E* denotes the ordinal value of the element.

22.18.1.5. FILE INTERFACE BLOCKS

The table below shows the format of a file interface block:

@ + 0	Flags byte.
@ + 1	Character buffer.
@ + 2	Sector buffer pointer (LSB).
@ + 3	Sector buffer pointer (MSB).
@ + 4	Number of records (LSB).
@ + 5	Number of records (MSB).
@ + 6	Record length (LSB).
@ + 7	Record length (MSB).
@ + 8	Current record (LSB).
@ + 9	Current record (MSB).
@ + 10	Unused.
@ + 11	Unused.
@ + 12	First byte of CP/M FCB.
:	
@ + 47	Last byte of CP/M FCB.
@ + 48	First byte of sector buffer.
:	
@ + 175	Last byte of sector buffer.

The format of the flags byte at @ + 0 is:

Bit 0..3	File type.
Bit 4	Read semaphore.
Bit 5	Write semaphore.
Bit 6	Output flag.
Bit 7	Input flag.

File type 0 denotes a disk file, and 1 through 5 denote the TURBO Pascal logical I/O devices (CON:, KBD:, LST:, AUX:, and USR:). For

typed files, bit 4 is set if the contents of the sector buffer are undefined, and bit 5 is set if data has been written to the sector buffer. For text files, bit 5 is set if the character buffer contains a pre-read character. Bit 6 is set if output is allowed, and bit 7 is set if input is allowed.

The sector buffer pointer stores an offset (0..127) in the sector buffer at @ + 48. For typed and untyped files, the three words from @ + 4 to @ + 9 store the number of records in the file, the record length in bytes, and the current record number. The FIB of an untyped file has no sector buffer, and so the sector buffer ponter is not used.

When a text file is assigned to a logical device, only the flag's byte and the character buffer are used.

22.18.1.6. POINTERS

A pointer consists of two bytes containing a 16-bit memory address, and it is stored in memory using the byte-reversed format, i.e., the least significant byte is stored first. The value **nil** corresponds to a zero word.

22.18.2. DATA STRUCTURES

Data structures are built from the basic data types using various structuring methods. Three different structuring methods exist: arrays, records, and disk files. The structuring of data does not in any way affect the internal formats of the basic data types.

22.18.2.1. ARRAYS

The components with the lowest index values are stored at the lowest memory address. A multi-dimensional array is stored with the rightmost dimension increasing first, e.g., given the array

```
Board: array[1..8, 1..8] of Square
```

you have the following memory layout of its components:

```
Lowest address:    Board [1,1]
                   Board [1,2]
                     :
                   Board [1,8]
                   Board [2,1]
                   Board [2,2]
                     :
                     :
Highest address:   Board [8,8]
```

22.18.2.2. RECORDS

The first field of a record is stored at the lowest memory address. If the record contains no variant parts, the length is given by the sum of the lengths of the individual fields. If a record contains a variant, the total number of bytes occupied by the record is given by the length of the fixed part plus the length of the largest of its variant parts. Each variant starts at the same memory address.

22.18.2.3. DISK FILES

Disk files are different from other data structures in that data is not stored in internal memory but in a file on an external device. A disk file is controlled through a file interface block (FIB) as described on page 162. In general there are two different types of disk files: random-access files and text files.

22.18.2.3.1. RANDOM-ACCESS FILES

A random-access file consists of a sequence of records, all of the same length and same internal format. To optimize the file-storage capacity, the records of a file are totally contiguous. The first four bytes of the first sector of a file contains the number of records in the file and the length of each record in bytes. The record of the file is stored starting at the fourth byte.

```
sector 0, byte 0:  Number of records (LSB)
sector 0, byte 1:  Number of records (MSB)
sector 0, byte 2:  Record length (LSB)
sector 0, byte 3:  Record length (MSB)
```

22.18.2.3.2. TEXT FILES

The basic components of a text file are characters, but a text file is subdivided into *lines*. Each line consists of any number of characters ended by a CR/LF sequence (ASCII $0D/$0A). The file is terminated by a Ctrl-Z (ASCII $1A).

22.18.3. PARAMETERS

Parameters are transferred to procedures and functions via the Z-80 stack. Normally, this is of no interest to the programmer, as the machine code generated by TURBO Pascal will automatically PUSH parameters onto the stack before a call, and POP them at the beginning of the subprogram. However, if the programmer wishes to use **external** subprograms, these must POP the parameters from the stack themselves.

On entry to an **external** subroutine, the top of the stack always contains the return address (a word). The parameters, if any, are located below the return address, i.e., at higher addresses on the stack. Therefore, to access the parameters, the subroutine must first POP off the return address, then all the parameters, and finally it must restore the return address by PUSHing it back onto the stack.

22.18.3.1. VARIABLE PARAMETERS

With a variable (**var**) parameter, a word is transferred on the stack giving the absolute memory address of the first byte occupied by the actual parameter.

22.18.3.2. VALUE PARAMETERS

With value parameters, the data transferred on the stack depends upon the type of the parameter, as described in the following sections.

22.18.3.2.1. SCALARS

Integers, Booleans, Chars and declared scalars are transferred on the stack as a word. If the variable occupies only one byte when it is stored, the most significant byte of the parameter is zero. Normally, a word is popped from the stack using an instruction like POP HL.

22.18.3.2.2. REALS

A real is transferred on the stack using six bytes. If these bytes are popped using the instruction sequence:

```
POP     HL
POP     DE
POP     BC
```

then L will contain the exponent, H the fifth (least significant) byte of the mantissa, E the fourth byte, D the third byte, C the second byte, and B the first (most significant) byte.

22.18.3.2.3. STRINGS

When a string is at the top of the stack, the byte pointed to by SP contains the length of the string. The bytes at addresses SP + 1 through SP + n (where n is the length of the string) contain the string, with the first character stored at the lowest address. The following machine-code instructions may be used to pop the string at the top of the stack and store it in *StrBuf:*

```
LD      DE,StrBuf
LD      HL,0
LD      B,H
ADD     HL,SP
LD      C,(HL)
INC     BC
LDIR
LD      SP,HL
```

22.18.3.2.4. SETS

A set always occupies 32 bytes on the stack (set compression only applies to the loading and storing of sets). The following machine code instruction may be used to pop the set at the top of the stack and store it in *SetBuf:*

```
LD      DE,SetBuf
LD      HL,0
ADD     HL,SP
LD      BC,32
LDIR
LD      SP,HL
```

This will store the least significant byte of the set at the lowest address in *SetBuf*.

22.18.3.2.5. POINTERS

A pointer value is transferred on the stack as a word containing the memory address of a dynamic variable. The value **nil** corresponds to a zero word.

22.18.3.2.6. ARRAYS AND RECORDS

Even when used as value parameters, *Array* and *Record* parameters are not actually pushed onto the stack. Instead, a word containing the address of the first byte of the parameter is transferred. It is then the responsibility of the subroutine to pop this word, and use it as the source address in a block copy operation.

22.18.4. FUNCTION RESULTS

User-written **external** functions must return their results exactly as specified in the following:

Values of scalar types must be returned in the HL register pair. If the type of the result is expressed in one byte, then it must be returned in L, and H must be zero .

Reals must be returned in the BC, DE, and HL register pairs. B, C, D, E, and H must contain the mantissa (most significant byte in B), and L must contain the exponent.

Strings and **sets** must be returned on the top of the stack in the formats described on page 166.

Pointer values must be returned in the HL register pair.

22.18.5. THE HEAP AND THE STACKS

As indicated by the memory maps in previous sections, three stack-like structures are maintained during execution of a program: The *heap*, the *CPU stack*, and the *recursion stack*.

The heap is used to store dynamic variables, and is controlled with the standard procedures *New*, *Mark*, and *Release*. At the beginning of a program, the heap pointer *HeapPtr* is set to the address of the bottom of free memory, i.e., the first free byte after the object code.

The CPU stack is used to store intermediate results during evaluation of expressions and to transfer parameters to procedures and functions. An active **for** statement also uses the CPU stack, and occupies one word. At the beginning of a program, the CPU-stack pointer *StackPtr* is set to the address of the top of free memory.

The recursion stack is used only by recursive procedures and function, i.e., procedures and functions compiled with the passive ({$A-}) **A** compiler directive. On entry to a recursive subprogram it copies its work space onto the recursion stack, and on exit the entire work space is restored to its original state. The default initial value of *RecurPtr* at the beginning of a program is 1K ($400) bytes below the CPU-stack pointer.

Because of this technique, variables local to a subprogram must not be used as **var** parameters in recursive calls.

The pre-defined variables

HeapPtr:	The heap pointer,
RecurPtr:	The recursion-stack pointer, and
StackPtr:	The CPU-stack pointer

allow the programmer to control the position of the heap and the stacks.

The type of these variables is *Integer*. *HeapPtr* and *RecurPtr* may be used in the same context as any other *Integer* variable, but *StackPtr* may only be used in assignments and expressions.

When these variables are manipulated, always make sure that they point to addresses within free memory, and that:

HeapPtr ‹ *RecurPtr* ‹ *StackPtr*

Failure to adhere to these rules will cause unpredictable, perhaps fatal, results.

Needless to say, assignments to the heap and stack pointers must never occur once the stacks or the heap are in use.

On each call to the procedure *New*, and on entering a recursive procedure or function, the system checks for collision between the heap and the recursion stack, i.e., checks whether *HeapPtr* is less than *RecurPtr*. If not, a collision has occurred, which results in an execution error.

Note that **no** checks are made at any time to insure that the CPU stack does not overflow into the bottom of the recursion stack. For this to happen, a recursive subroutine must call itself some 300-400 times, which must be considered a rare situation. If, however, a program requires such nesting, the following statement executed at the beginning of the program block will move the recursion stack pointer downwards to create a larger CPU stack:

```
RecurPtr := StackPtr - 2 * MaxDepth - 512;
```

where *MaxDepth* is the maximum required depth of calls to the recursive subprogram(s). The extra approximately 512 bytes are needed as a margin to make room for parameter transfers and intermediate results during the evaluation of expressions.

22.19. MEMORY MANAGEMENT

22.19.1. MEMORY MAPS

The following diagrams illustrate the contents of memory at different stages of working with the TURBO system. Solid lines indicate fixed boundaries (i.e., determined by amount of memory, size of your CP/M, version of TURBO, etc.), whereas dotted lines indicate boundaries which are determined at run-time (e.g., by the size of the source text,

and by possible user manipulation of various pointers, etc.). The sizes of the segments in the diagrams do not necessarily reflect the amounts of memory actually consumed.

22.19.1.1. COMPILATION IN MEMORY

During compilation of a program in memory (**M**emory-mode on compiler **O**ptions menu, see page 143), the memory is mapped as follows:

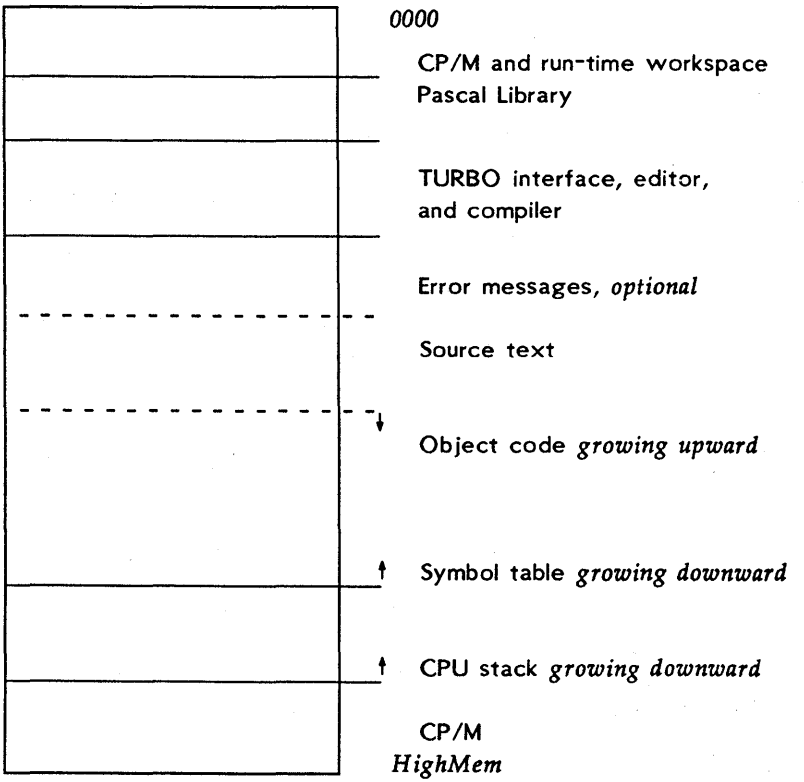


Figure 22-5: Memory map during compilation in memory

If the error-message file is not loaded when starting TURBO, the source text starts that much lower in memory. When the compiler is invoked, it generates object code working upwards from the end of the source text.

The CPU stack works downwards from the logical top of memory, and the compiler's symbol table works downwards from an address 1K ($400 bytes) below the logical top of memory.

22.19.1.2. COMPILATION TO DISK

During compilation to a .COM or .CHN file (Com-mode or cHn-mode on compiler Options menu, see page 143), the memory looks much as during compilation in memory (see preceding section) ***except*** that generated object code does not reside in memory but is written to a disk file. Also, the code starts at a higher address (right after the Pascal library instead of after the source text). Compilation of much larger programs is thus possible in this mode.

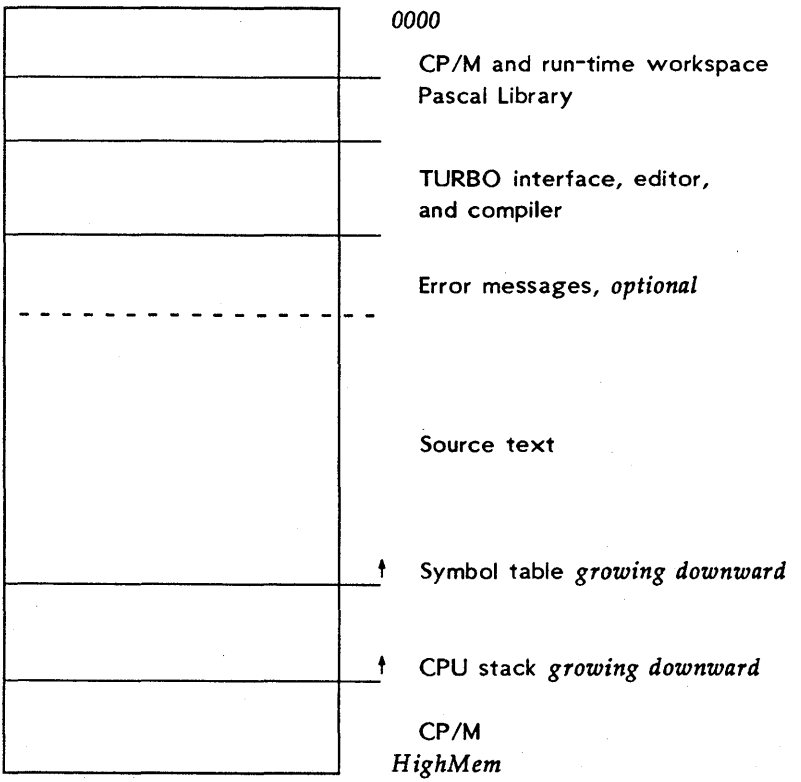


Figure 22-6: Memory map during compilation to a file

22.19.1.3. EXECUTION IN MEMORY

When a program is executed in direct, or memory mode (i.e., the **M**emory-mode on compiler **O**ptions menu is selected, see page 143), the memory is mapped as follows:

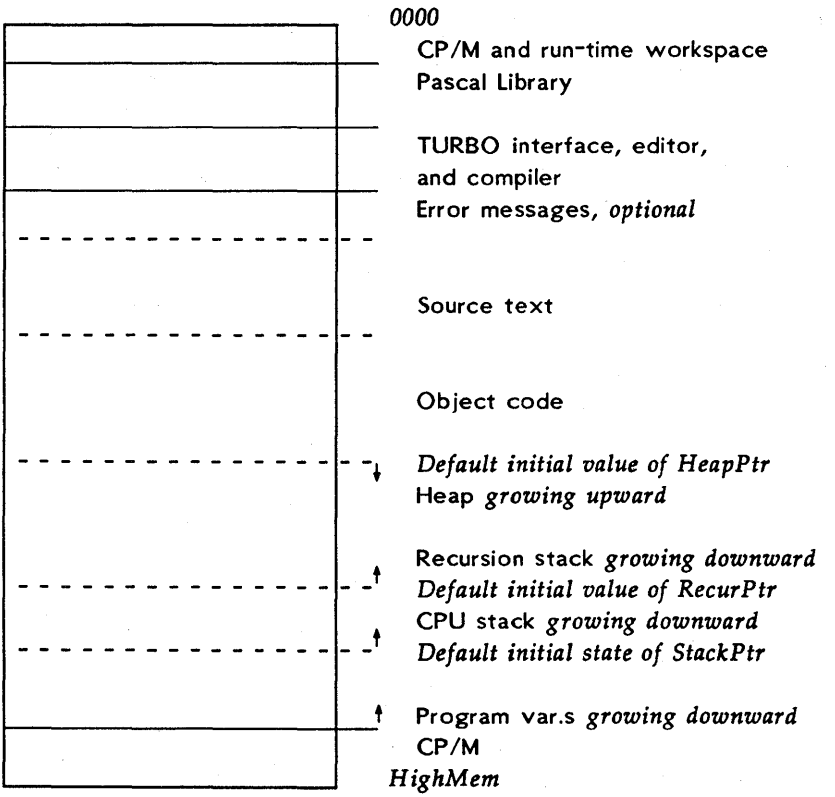


Figure 22-7: Memory map during execution in direct mode

When a program is compiled, the end of the object code is known. The heap pointer *HeapPtr* is set to this value by default, and the heap grows from here and upwards in memory towards the recursion stack. The maximum memory size is BDOS minus one (indicated on the compiler **O**ptions menu). Program variables are stored from this address and downwards.

The end of the variables is the "top of free memory", which is the initial value of the CPU stack pointer *StackPtr*. The CPU stack grows downwards from here towards the position of the recursion stack pointer *RecurPtr*, $400 bytes lower than *StackPtr*. The recursion stack grows from here downwards towards the heap.

22.19.1.4. EXECUTION OF A PROGRAM FILE

When a program file is executed (either by the **R**un command with the **M**emory mode on the compiler **O**ptions menu selected, by an e**X**ecute command, or directly from CP/M), the memory is mapped as follows:

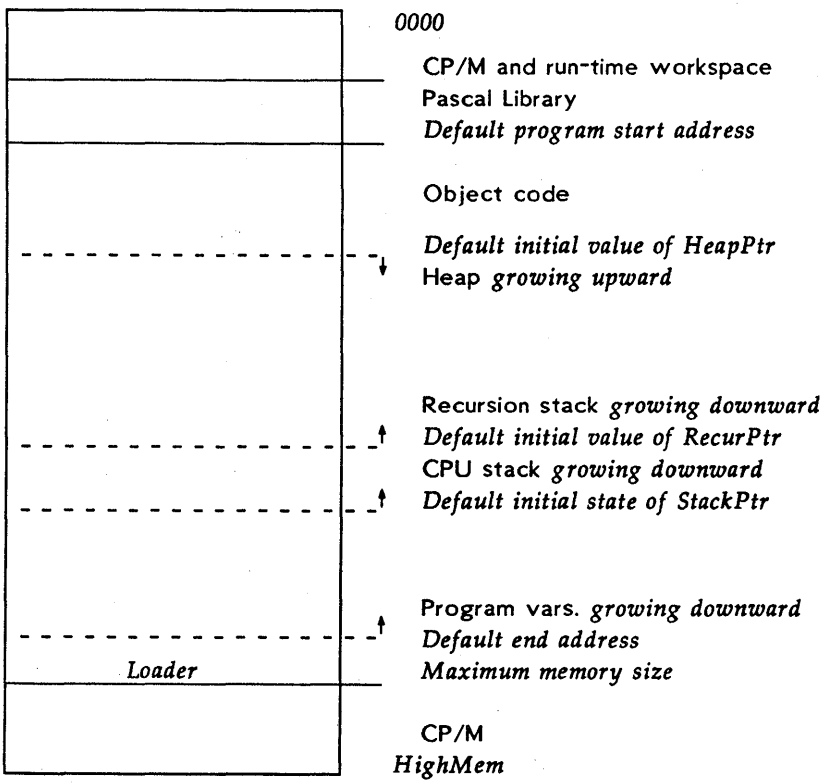


Figure 22-8: Memory map during execution of a program file

This map resembles the previous, exept for the absence of the TURBO interface, editor, and compiler (and possible error messages) and of the source text. The *default program start address* (shown on the compiler **O**ptions menu) is the first free byte after the Pascal runtime library. This value may be manipulated with the **S**tart address command of the compiler **O**ptions menu, e.g., to create space for **absolute** variables and/or external procedures between the library and the code. The *maximum memory size* is BDOS minus one, and the default value is determined by the BDOS location on the computer in use.

If programs are to translated for other systems, care should be taken to avoid collision with the BDOS. The maximum memory may be manipulated with the **E**nd address command of the compiler **O**ptions menu. Notice that the *default end address* setting is approximately 700 to 1000 bytes lower than maximum memory. This is to allow space for the loader, which resides just below BDOS when .COM files are **R**un or e**X**ecuted from the TURBO system. This loader restores the TURBO editor, compiler, and possible error messages when the program finishes, and thus returns control to the TURBO system.

Appendix A
STANDARD PROCEDURES AND FUNCTIONS

This appendix lists all standard procedures and functions available in TURBO Pascal and describes their application, syntax, parameters, and type. The following symbols are used to denote elements of various types:

type	any type
string	any string type
file	any file type
scalar	any scalar type
pointer	any pointer type

Where parameter type specification is not present, it means that the procedure or function accepts variable parameters of any type.

A.1. INPUT/OUTPUT PROCEDURES AND FUNCTIONS

The following procedures use a non-standard syntax in their parameter list:

procedure

Read (**var** *F*: **file of** *type*; **var** V: *type*);
Read (**var** *F*: text; **var** *I*: Integer);
Read (**var** *F*: text; **var** *R*: Real);
Read (**var** *F*: text; **var** *C*: Char);
Read (**var** *F*: text; **var** *S*: *string*);
Readln (**var** *F*: text);
Write (**var** *F*: **file of** *type*; **var** V: *type*);
Write (**var** *F*: text; *I*: Integer);
Write (**var** *F*: text; *R*: Real);
Write (**var** *F*: text; *B*: Boolean);
Write (**var** *F*: text; *C*: Char);
Write (**var** *F*: text; *S*: *string*);
Writeln (**var** *F*: text);

A.2. ARITHMETIC FUNCTIONS

function

```
Abs (I: Integer): Integer;
Abs (R: Real): Real;
ArcTan (R: Real): Real;
Cos (R: Real): Real;
Exp (R: Real): Real;
Frac (R: Real): Real;
Int (R: Real): Real;
Ln (R: Real): Real;
Sin (R: Real): Real;
Sqr (I: Integer): Integer;
Sqr (R: Real): Real;
Sqrt (R: Real): Real;
```

A.3. SCALAR FUNCTIONS

function

```
Odd (I: Integer): Boolean;
Pred (X: scalar): scalar;
Succ (X: scalar): scalar;
```

A.4. TRANSFER FUNCTIONS

function

```
Chr (I: Integer): Char;
Ord (X: scalar): Integer;
Round (R: Real): Integer;
Trunc (R: Real): Integer;
```

A.5. STRING PROCEDURES AND FUNCTIONS

The *Str* procedure uses a non-standard syntax for its numeric parameter.

procedure

```
Delete (var S: string; Pos, Len: Integer);
Insert (S: string; var D: string; Pos: Integer);
Str (I: Integer; var S: string);
Str (R: Real; var S: string);
```

```
    Val (S: string; var R: Real; var P: Integer);
    Val (S: string; var I, P: Integer);

function
    Concat (S1, S2, ..., Sn: string): string;
    Copy (S: string; Pos, Len: Integer): string;
    Length (S: string): Integer;
    Pos (Pattern, Source: string): Integer;
```

A.6. FILE-HANDLING ROUTINES

```
procedure
    Append (var F: file; Name: string);
    Assign (var F: file; Name: string);
    BlockRead (var F: file; var Dest: Type; Num: Integer);
    BlockWrite (var F: file; var Dest: Type; Num: Integer);
    Chain (var F: file);
    Close (var F: file);
    Erase (var F: file);
    Execute (var F: file);
    Rename (var F: file; Name: string);
    Reset (var F: file);
    Rewrite (var F: file);
    Seek (var F: file of type; Pos: Integer);

function
    Eof (var F: file): Boolean;
    Eoln (var F: Text): Boolean;
    FilePos (var F: file of type): Integer;
    FilePos (var F: file): Integer;
    FileSize (var F: file of type): Integer;
    FileSize (var F: file): Integer;
    SeekEof (var F: file): Boolean;
    SeekEoln (var F: Text): Boolean;
```

A.7. HEAP-CONTROL PROCEDURES AND FUNCTIONS

```
procedure
    Dispose (var P: pointer);
    FreeMem (var P: pointer, I: Integer);
    GetMem (var P: pointer, I: Integer);
```

```
    Mark (var P: pointer);
    New (var P: pointer);
    Release (var P: pointer);

function
    MaxAvail: Integer;
    MemAvail: Integer;
    Ord (P: pointer): Integer;
    Ptr (I: Integer): pointer;
```

A.8. SCREEN-RELATED PROCEDURES AND FUNCTIONS

```
procedure
    CrtExit;
    CrtInit;
    ClrEol;
    ClrScr;
    DelLine;
    GotoXY (X, Y: Integer);
    InsLine;
    LowVideo;
    NormVideo;
```

A.9. MISCELLANEOUS PROCEDURES AND FUNCTIONS

```
procedure
    Bdos (Func,Param: Integer);
    Bios (Func,Param: Integer);
    Delay (mS: Integer);
    FillChar (var Dest,Length: Integer; Data: Char);
    FillChar (var Dest,Length: Integer; Data: Byte);
    Halt;
    Move (var Source,Dest: type; Length: Integer);
    Randomize;

function
    Addr (var Variable): Integer;
    Addr (<function identifier>): Integer;
    Addr (<procedure identifier>): Integer;
    Bdos (Func, Param: Integer): Byte;
    BdosHL (Func, Param: Integer): Integer;
```

```
Bios (Func,Param: Integer): Byte;
BiosHL (Func,Param: Integer): Integer;
Hi (I: Integer): Integer;
IOresult: Boolean;
KeyPressed: Boolean;
Lo (I: Integer): Integer;
ParamCount: Integer;
ParamStr (N: Integer): string;
Random (Range: Integer): Integer;
Random: Real;
SizeOf (var Variable): Integer;
SizeOf (<type identifier>): Integer;
Swap (I: Integer): Integer;
UpCase (Ch: Char): Char;
```

Appendix B
SUMMARY OF OPERATORS

The following table summarizes all operators of TURBO Pascal. The operators are grouped in order of descending precedence. Where *Type of operand(s)* is indicated as *Integer, Real,* the result is as follows:

Operands	**Result**
Integer, Integer	Integer
Real, Real	Real
Real, Integer	Real

Operator	**Operation**	**Type of operand(s)**	**Type of result**
+ unary	sign identity	Integer, Real	as operand
- unary	sign inversion	Integer, Real	as operand
not	negation	Integer, Boolean	as operand
*	multiplication	Integer, Real	Integer, Real
	set intersection	any set type	as operand
/	division	Integer, Real	Real
div	Integer division	Integer	Integer
mod	modulus	Integer	Integer
and	arithmetical **and**	Integer	Integer
	logical **and**	Boolean	Boolean
shl	shift left	Integer	Integer
shr	shift right	Integer	Integer
+	addition	Integer, Real	Integer,Real
	concatenation	string	string
	set union	any set type	as operand
-	subtraction	Integer, Real	Integer, Real
	set difference	any set type	as operand
or	arithmetical **or**	Integer	Integer
	logical **or**	Boolean	Boolean
xor	arithmetical **xor**	Integer	Integer
	logical **xor**	Boolean	Boolean

Operator	Operation	Type of operand(s)	Type of result
=	equality	any scalar type	Boolean
	equality	string	Boolean
	equality	any set type	Boolean
	equality	any pointer type	Boolean
<>	inequality	any scalar type	Boolean
	inequality	string	Boolean
	inequality	any set type	Boolean
	inequality	any pointer type	Boolean
>=	greater or equal	any scalar type	Boolean
	greater or equal	string	Boolean
	set inclusion	any set type	Boolean
<=	less or equal	any scalar type	Boolean
	less or equal	string	Boolean
	set inclusion	any set type	Boolean
>	greater than	any scalar type	Boolean
	greater than	string	Boolean
<	less than	any scalar type	Boolean
	less than	string	Boolean
in	set membership	see below	Boolean

The first operand of the **in** operator may be of any scalar type, and the second operand must be a set of that type.

Appendix C
SUMMARY OF COMPILER DIRECTIVES

A number of features of the TURBO Pascal compiler are controlled through compiler directives. A compiler directive is introduced as a comment with a special syntax. Whenever a comment is allowed in a program, a compiler directive is also allowed.

A compiler directive consists of an opening bracket immediately followed by a dollar sign immediately followed by one compiler directive letter or a list of compiler directive letters separated by commas, ultimately terminated by a closing bracket.

Examples:
```
{$I-}
{$I INCLUDE.FIL}
{$B-,R+,V-}
(*$U+*)
```

Notice that no spaces are allowed before or after the dollar sign. A plus sign after a directive indicates that the associated compiler feature is enabled (active), and a minus sign indicates that is disabled (passive).

C.1. IMPORTANT NOTICE

All compiler directives have default values. These have been chosen to optimize execution speed and minimize code size. For example, code generation for recursive procedures and index checking has been disabled. Check below to make sure that your programs include the required compiler directive settings!

C.2. A – ABSOLUTE CODE

Default: A+

The **A** directive controls generation of absolute, i.e., non-recursive, code. When active, {$A+}, absolute code is generated. When passive, {$A-}, the compiler generates code which allows recursive calls. This code requires more memory and executes more slowly.

C.3. B – I/O MODE SELECTION

Default: B+

The **B** directive controls input/output mode selection. When active, {$B+}, the CON: device is assigned to the standard files *Input* and *Output*, i.e., the default input/output channel. The TRM: device is used when the directive is passive, {$B-}. **This directive is global to an entire program block** and cannot be re-defined locally. See pages 92 and 95 for further details.

C.4. C – CTRL-C AND CTRL-S

Default: C+

The **C** directive controls interpretation of control characters during console I/O. When active, {$C+}, a Ctrl-C entered in response to a *Read* or *Readln* statement will interrupt program execution, and a Ctrl-S will toggle screen output off and on. When passive, {$C-}, control characters are not interpreted. The active state slows screen output somewhat, so if screen output speed is imperative, you should switch off this directive. **This directive is global to an entire progam block** and cannot be redefined locally.

C.5. I – I/O ERROR HANDLING

Default: I+

The **I** directive controls I/O error handling. When active, {$I+}, all I/O operations are checked for errors. When passive, {$I-}, it is the responsibility of the programmer to check I/O errors through the standard function *IOresult*. See page 103 for further details.

C.6. I – INCLUDE FILES

The **I** directive followed by a file name instructs the compiler to include the file with the specified name in the compilation. Include files are discussed in detail in Chapter 17.

C.7. R – INDEX RANGE CHECK

Default: R-

The **R** directive controls run-time index checks. When active, {$R+}, all array-indexing operations are checked to be within the defined bounds, and all assignments to scalar and subrange variables are checked to be within range. When passive, {$R-}, no checks are performed, and index errors may well cause a program to go haywire. It is a good idea to activate this directive while developing a program. Once debugged, execution will be speeded up by setting it passive (the default state).

C.8. U – USER INTERRUPT

Default: U-

The **U** directive controls user interrupts. When active, {$U+}, the user may interrupt the progam any time during execution by entering a Ctrl-C. When passive, {$U-}, this has no effect. Activating this directive will slow down execution speed significantly.

C.9. V – VAR-PARAMETER TYPE CHECKING

Default: V+

The **V** compiler directive controls type checking on strings passed as **var** parameters. When active, {$V+}, strict type checking is performed, i.e., the lengths of actual and formal parameters must match. When passive, {$V-}, the compiler allows passing of actual parameters which do not match the length of the formal parameters. See page 150 for further details.

C.10. W – NESTING OF WITH STATEMENTS

Default: W2

The **W** directive controls the level of nesting of *With* statements, i.e., the number of records which may be "opened" within one block. The **W** must be immediately followed by a digit between 1 and 9. For further details, please refer to page 70.

C.11. X – ARRAY OPTIMIZATION

Default: X+

The **X** directive controls array optimization. When active, {$X+}, code generation for arrays is optimized for maximum speed. When passive, {$X-}, the compiler minimizes the code size instead. This is discussed further on page 65.

Appendix D
TURBO VS. STANDARD PASCAL

The TURBO Pascal language follows the Standard Pascal defined by Jensen and Wirth in their **User Manual and Report**, with only minor differences introduced for the sheer purpose of efficiency. These differences are described in the following. Notice that the *extensions* offered by TURBO Pascal are discussed.

D.1. DYNAMIC VARIABLES

The procedure *New* will not accept variant record specifications. This restriction, however, is easily circumvented by using the standard procedure *GetMem*.

D.2. RECURSION

Because of the way local variables are handled during recursion, a variable local to a subprogram must not be passed as a **var** parameter in recursive calls.

D.3. GET AND PUT

The standard procedures *Get* and *Put* are not implemented. Instead, the *Read* and *Write* procedures have been extended to handle all I/O needs. There are three reasons for this: First, *Read* and *Write* give much faster I/O; second, variable-space overhead is reduced as file-buffer variables are not required; and third, the *Read* and *Write* procedures are far more versatile and easier to understand than *Get* and *Put*.

D.4. GOTO STATEMENTS

A **goto** statement must not leave the current block.

D.5. PAGE PROCEDURE

The standard procedure *Page* is not implemented, as the CP/M operating system does not define a form-feed character.

D.6. PACKED VARIABLES

The reserved word **packed** has no effect in TURBO Pascal, but it is still allowed. This is because packing occurs automatically whenever possible. For the same reason, standard procedures ***Pack*** and ***Unpack*** are not implemented.

D.7. PROCEDURAL PARAMETERS

Procedures and functions cannot be passed as parameters.

Appendix E
COMPILER ERROR MESSAGES

The following is a listing of error messages you may get from the compiler. When encountering an error, the compiler will always print the error number on the screen. Explanatory texts will only be issued if you have included error messages (answer **Y** to the first question when you start TURBO).

Many error messages are totally self-explanatory, but some need a little elaboration as provided in the following.

01 **';' expected**
02 **':' expected**
03 **',' expected**
04 **'(' expected**
05 **')' expected**
06 **'=' expected**
07 **':=' expected**
08 **'[' expected**
09 **']' expected**
10 **'.' expected**
11 **'..' expected**
12 **BEGIN expected**
13 **DO expected**
14 **END expected**
15 **OF expected**
16 **PROCEDURE or FUNCTION expected**
17 **THEN expected**
18 **TO or DOWNTO expected**
20 **Boolean expression expected**
21 **File variable expected**
22 **Integer constant expected**
23 **Integer expression expected**
24 **Integer variable expected**
25 **Integer or real constant expected**
26 **Integer or real expression expected**
27 **Integer or real variable expected**
28 **Pointer variable expected**
29 **Record variable expected**

30 Simple type expected
Simple types are all scalar type, except real.

31 Simple expression expected

32 String constant expected

33 String expression expected

34 String variable expected

35 Textfile expected

36 Type identifier expected

37 Untyped file expected

40 Undefined label
A statement references an undefined label.

41 Unknown identifier or syntax error
Unknown label, constant, type, variable, or field identifier, or syntax error in statement.

42 Undefined pointer type in preceding type definitions
A preceding pointer type definition contains a reference to an unknown type identifier.

43 Duplicate identifier or label
This identifier or label has already been used within the current block.

44 Type mismatch
1) Incompatible types of the variable and the expression in an assignment statement. **2)** Incompatible types of the actual and the formal parameters in a call to a subprogram. **3)** Expression type incompatible with index type in array assignment. **4)** Types of operands in an expression are not compatible.

45 Constant out of range

46 Constant and CASE selector type does not match

47 Operand type(s) does not match operator
Example: 'A' div '2'

48 Invalid result type
Valid types are all scalar types, string types, and pointer types.

49 Invalid string length
The length of a string must be in the range 1..255.

50 String constant length does not match type

51 Invalid subrange base type
Valid base types are all scalar types, except real.

52 **Lower bound > upper bound**
The ordinal value of the upper bound must be greater than or equal to the ordinal value of the lower bound.

53 **Reserved word**
These may not be used as identifiers.

54 **Illegal assignment**

55 **String constant exceeds line**
String constants must not span lines.

56 **Error in integer constant**
An *Integer* constant does not conform to the syntax described in page 35, or it is not within the *Integer* range -32768..32767. Whole *Real* numbers should be followed by a decimal point and a zero, e.g., 123456789.0.

57 **Error in real constant**
The syntax of *Real* constants is defined on page 35.

58 **Illegal character in identifier**

60 **Constants are not allowed here**

61 **Files and pointers are not allowed here**

62 **Structured variables are not allowed here**

63 **Textfiles are not allowed here**

64 **Textfiles and untyped files are not allowed here**

65 **Untyped files are not allowed here**

66 **I/O not allowed here**
Variables of this type cannot be input or output.

67 **Files must be VAR parameters**

68 **File components may not be files**
file of file constructs are not allowed.

69 **Invalid ordering of fields**

70 **Set base type out of range**
The base type of a set must be a scalar with no more than 256 possible values or a subrange with bounds in the range 0..255.

71 **Invalid GOTO**
A **goto** cannot reference a label within a **for** loop from outside that **for** loop.

72 **Label not within current block**
A **goto** statement cannot reference a label outside the current block.

73 Undefined FORWARD procedure(s)

A subprogram has been **forward** declared, but the body never occurred.

74 INLINE error

75 Illegal use of ABSOLUTE

1) Only one identifier may appear before the colon in an **absolute** variable declaration. **2)** The **absolute** clause may not be used in a record.

76 Overlays can not be forwarded

The **forward** specification cannot not be used in connection with overlays.

77 Overlays not allowed in direct mode

Overlays can only be used from programs compiled to a file.

90 File not found

The specified include file does not exist.

91 Unexpected end of source

Your program cannot end the way it does. The program probably has more **begin**s than **end**s.

92 Unable to create overlay file

93 Invalid compiler directive

97 Too many nested WITHs

Use the W compiler directive to increase the maximum number of nested WITH statements. Default is 2.

98 Memory overflow

You are trying to allocate more storage for variables than is available.

99 Compiler overflow

There is not enough memory to compile the program. This error may occur even if free memory seems to exist; it is, however, used by the stack and the symbol table during compilation. Break your source text into smaller segments and use include files.

Appendix F
RUN-TIME ERROR MESSAGES

Fatal errors at run time halt a program and display:

```
Run-time error NN, PC=addr
    Program aborted
```

where *NN* is the run-time error number, and *addr* is the address in the program code where the error occured. In the following explanations of all run-time error numbers, notice that the numbers are hexadecimal!

01 **Floating point overflow.**

02 **Division by zero attempted.**

03 **Sqrt argument error.**
The argument passed to the Sqrt functiion was negative.

04 **Ln argument error.**
The argument passed to the Ln function was zero or negative.

10 **String length error.**
1) A string concatenation resulted in a string of more than 255 characters. **2)** Only strings of length 1 can be converted to a character.

11 **Invalid string index.**
Index expression is not within 1..255 with *Copy*, *Delete*, or *Insert* procedure calls.

90 **Index out of range.**
The index expression of an array subscript was out of range.

91 **Scalar or subrange out of range.**
The value assigned to a scalar or a subrange variable was out of range.

92 **Out of integer range.**
The real value passed to *Trunc* or *Round* was not within the *Integer* range -32768..32767.

F0 **Overlay file not found.**

FF **Heap/stack collision.**
A call was made to the standard procedure *New* or to a recursive subprogram, and there is insufficient free memory between the heap pointer (HeapPtr) and the recursion stack pointer (Recurptr).

Appendix G
I/O ERROR MESSAGES

An error in an input or output operation at run time causes in an I/O error. If I/O checking is active (**I** compiler directive active), an I/O error causes the program to halt and the following error message is displayed:

```
I/O error NN, PC=addr
Program aborted
```

Where *NN* is the I/O error number, and *addr* is the address in the program code where the error occured.

If I/O-error checking is passive ({$I-}), an I/O error will not cause the program to halt. Instead, all further I/O is suspended until the result of the I/O operation has been examined with the standard function *IOresult*. If I/O is attempted before *IOresult* is called after an error, a new error occurs, possibly hanging the program.

In the following explanations of all run-time error numbers, notice that the numbers are hexidecimal!

01 **File does not exist.**
The file name used with *Reset, Erase, Rename, Execute,* or *Chain* does not specify an existng file.

02 **File not open for input.**
1) You are trying to read (with *Read* or *Readln*) from a file without a previous *Reset* or *Rewrite.* **2)** You are trying to read from a text file which was prepared with *Rewrite* (and thus is empty). **3)** You are trying to read from the logical device LST:, which is only for output.

03 **File not open for output.**
1) You are trying to write (with *Write* or *Writeln*) to a file without a previous *Reset* or *Rewrite.* **2)** You are trying to read from a text file which was prepared with *Reset.* **3)** You are trying to write to the logical device KBD:, which is an input-only device.

04 **File not open.**
You are trying to access (with *BlockRead* or *BlockWrite*) a file without a previous *Reset* or *Rewrite.*

10 **Error in numeric format.**
The string read from a text file into a numeric variable does not conform to the proper numeric format (see page 35).

20 **Operation not allowed on a logical device.**
You are trying to *Erase, Rename, Execute,* or *Chain* a file assigned to a logical device.

21 **Not allowed in direct mode.**
Programs cannot be *Executed* or *Chained* from a program running in direct mode (i.e. a program activated with a **Run** command while the **Memory** compiler option is set).

22 **Assign to std files not allowed.**

90 **Record length mismatch.**
The record length of a file variable does not match the file you are trying to associate it with.

91 **Seek beyond end-of-file.**

99 **Unexpected end-of-file.**
1) Physical end-of-file encountered before EOF character (Ctrl-Z) when reading from a text file. **2)** An attempt was made to read beyond end-of-file on a defined file. **3)** A *Read* or *BlockRead* is unable to read the next sector of a defined file. Something may be wrong with the file, or (in the case of *BlockRead*) you may be trying to read past physical EOF.

F0 **Disk write error.**
Disk full while attempting to expand a file. This may occur with the output operations *Write, Writeln, BlockWrite,* and *Flush,* but also *Read, Readln,* and *Close* may cause this error, as they cause the write buffer to be flushed.

F1 **Directory is full.**
You are trying to *Rewrite* a file, and there is no more room in the disk directory.

F2 **File size overflow.**
You are trying to *Write* a record beyond 65535 to a defined file.

F3 **Too many open files.**

FF **File disappeared.**
An attempt was made to *Close* a file which was no longer present in the disk directory, e.g., because of an unexpected disk change.

Appendix H
TRANSLATING ERROR MESSAGES

The compiler error messages are collected in the file *TURBOMSG*. These messages are in English, but may be translated into any other language easily, as described in the following.

The first 24 lines of this file define a number of text constants for subsequent inclusion in the error-message lines; a technique which drastically reduces the disk and memory requirements of the error messages. Each constant is identified by a control character, denoted by a ^ character in the following listng. The value of each constant is anything that follows on the same line. All characters are significant, also leading and trailing blanks.

The remaining lines each contain one error message, starting with the error number and immediately followed by the message text. The message text may consist of any characters and may include previously defined constant identifiers (control character). Appendix E lists the resulting messages in full.

When you translate the error message, the relation between constants and error messages will probably be quite different from the English version listed here. Start therefore with writing each error message in full, disregarding the use of constants. You may use these error messages, but they will require excessive space. When all messages are translated, you should find as many common denominators as possible. Then define these as constants at the top of the file and include only the constant identifiers in subsequent message texts. You may define as few or as many constants as you need, the restriction being only the number of control characters.

As a good example of the use of constants, consider errors 25, 26, and 27. These are defined exclusively by constant identifiers, 15 total, but would require 101 characters if written in clear text.

The TURBO editor may be used to edit the *TURBOMSG* file. Control characters are entered with the Ctrl-P prefix, i.e., to enter a Ctrl-A (^A) into the file, hold down the <CONTROL> key and press first P, then A. Control characters appear dim on the TURBO editor screen (if your terminal has that video attribute).

Notice that the TURBO editor deletes all trailing blanks. The original message therefore does not use trailing blanks in any messages.

H.1. ERROR-MESSAGE FILE LISTING

```
^A are not allowed
^B can not be
^C constant
^D does not
^E expression
^F identifier
^G file
^H here
^KInteger
^LFile
^NIllegal
^O or
^PUndefined
^Q match
^R real
^SString
^TTextfile
^U out of range
^V variable
^W overflow
^X expected
^Y type
^[Invalid
^] pointer
01';'^X
02':'^X
03',^X
04'('^X
05')'^X
06'='^X
07':='^X
08'['^X
09']'^X
10'.'^X
11'..'^X
12BEGIN^X
```

```
13DO^X
14END^X
15OF^X
16PROCEDURE^O FUNCTION^X
17THEN^X
18TO^O DOWNTO^X
20Boolean^E^X
21^L^V^X
22^K^C^X
23^K^E^X
24^K^V^X
25^K^O^R^C^X
26^K^O^R^E^X
27^K^O^R^V^X
28Pointer^V^X
29Record^V^X
30Simple^Y^X
31Simple^E^X
32^S^C^X
33^S^E^X
34^S^V^X
35^T^X
36Type^F^X
37Untyped^G^X
40^P label
41Unknown^F^O syntax error
42^P^]^Y in preceding^Y definitions
43Duplicate^F^O label
44Type mismatch
45^C^U
46^C and CASE selector^Y^D^Q
47Operand^Y(s) ^D^Q operator
48^[ result^Y
49^[ ^S length
50^S^C length^D^Q^Y
51^[ subrange base^Y
52Lower bound > upper bound
53Reserved word
54^N assignment
55^S^C exceeds line
56Error in integer^C
```

57Error in^R^C
58^N character in^F
60^Cs^A^H
61^Ls and^]s^A^H
62Structured^Vs^A^H
63^Ts^A^H
64^Ts and untyped^Gs^A^H
65Untyped^Gs^A^H
661/O not allowed^H
67^Ls must be^V parameters
68^L components^B^Gs
69^[^Ordering of fields
70Set base^Y^U
71^[GOTO
72Label not within current block
73^P FORWARD procedure(s)
74INLINE error
75^N use of ABSOLUTE
76Overlays^B forwarded
77Overlays not allowed in direct mode
90^L not found
91Unexpected end of source
92Unable to create overlay file
93^[compiler directive
97Too many nested WITHs
98Memory^W
99Compiler^W

Appendix I
TURBO SYNTAX

The syntax of the TURBO Pascal language is presented here using the formalism known as the *Backus-Naur Form.* The following symbols are not symbols of the TURBO Pascal language, but meta-symbols belonging to the BNF formalism:

::=	Means "is defined as".
|	Means "or".
{}	Enclose items which may be repeated zero or more times.

In the following, symbols and words which are part of the language are printed in this typeface. Reserved words are printed in this typeface, but in **boldface**, e.g.: **array** and **for**. Syntactic constructs are printed in *italics*, e.g., *block* and *case-element*.

actual-parameter ::= *expression* | *variable*
adding-operator ::= + | - | **or** | **xor**
array-constant ::= (*structured-constant*{,*structured-constant*})
array-type ::= **array**[*index-type*{,*index-type*}] **of** *component-type*
array-variable ::= *variable*
assignment-statement ::= *variable* := *expression* |
function-identifier := *expression*
base-type ::= *simple-type*
block ::= *declaration-part statement-part*
case-element ::= *case-list*: *statement*
case-label ::= *constant*
case-label-list ::= *case-label*{,*case-label*}
case-list ::= *case-list-element*{,*case-list-element*}
case-list-element ::= *constant* | *constant* .. *constant*
case-statement ::= **case** *expression* **of** *case-*

element{;case-element} **end** |
case *expression* **of** *case-element{;case-element}*
else *statement{;statement}* **end**
complemented-factor ::= signed-factor | **not** *signed-factor*
component-type ::= type
component-variable ::= indexed-variable | *field-designator*
component-statement ::= **begin** *statement{;statement}* **end**
conditional-statement ::= if-statement | *case-statement*
constant ::= unsigned-number | *sign unsigned-number* |
constant-identifier | *sign constant-identifier* | *string*
constant-definition-part ::= **const** *constant-definition*
{;constant-definition};
constant-definition ::= untyped-constant-definition |
typed-constant-definition
constant-identifier ::= identifier
control-character ::= # unsigned-integer | ^*character*
control-variable ::= variable-identifier
declaration-part ::= {declaration-section}
declaration-section ::= label-declaration-part | *constant-definition-part* |
type-definition-part | *variable-declaration-part* |
procedure-and-function-declaration-part
digit ::= 0 | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9*
digit-sequence ::= digit {digit}
empty ::=
empty-statement ::= empty
entire-variable ::= variable-identifier | *typed-constant-identifier*
expression ::= simple-expression {relational-operator simple-expression}
factor ::= variable | *unsigned-constant* | *(expression)* |
function-designator | *set*
field-designator ::= record-variable . field-identifier
field-identifier ::= identifier
field-list ::= fixed part | *fixed-part; variant-part* | *variant-part*
file-identifier ::= identifier
file-identifier-list ::= empty | *(file-identifier {, file-identifier}*
filetype ::= **file of** *type*
final-value ::= expression
fixed-part ::= record-section {; record-section}
for-list ::= initial-value **to** *final-value* | *initial-value* **downto** *final-value*
for-statement ::= **for** *control-variable:= for-list* **do** *statement*
formal-parameter-section ::= parameter-group | **var** *parameter-*

group
function-declaration ::= function-heading block;
function-designator ::= function-identifier | function-identifier
(actual-parameter {, actual-parameter})
function-heading ::= **function** *identifier: result-type; |*
function *identifier*
(formal-parameter-section
{, formal-parameter-section}): result-type;
function-identifier ::= identifier
goto-statement ::= **goto** *label*
hexdigit ::= digit | A | B | C | D | E | F
hexdigit-sequence ::= hexdigit {hexdigit}
identifier ::= letter {letter-or-digit}
identifier-list ::= identifier {, identifier}
if-statement ::= **if** *expression* **then** *statement* {**else** *statement*}
index-type ::= simple-type
indexed-variable ::= array-variable [expression {, expression}]
initial-value ::= expression
inline-list-element ::= unsigned-integer | constant-identifier |
variable-identifier | location-counter-reference
inline-statement ::= **inline** *inline-list-element*
{,inline-list-element}
label ::= letter-or-digit {letter-or-digit}
label-declaration-part ::= **label** *label {, label};*
letter ::= A | B | C | D | E | F | G | H | I | J | K | L | M |
N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
a | b | c | d | e | f | g | h | i | j | k | l | m |
n | o | p | q | r | s | t | u | v | w | x | y | z | _
letter-or-digit ::= letter | digit
location-counter-reference ::= * | * *sign constant*
multiplying-operator ::= * | / | **div** | **mod** | **and** | **shl** | **shr**
parameter-group ::= identifier-list: type-identifier
pointer-type ::= ^ type-identifier
pointer-variable ::= variable
procedure-and-function-declaration-part ::=
{procedure-or-function-declaration}
procedure-declaration ::= procedure-heading block;
procedure-heading ::= **procedure** *identifier; |* **procedure** *identifier*
(formal-parameter-section
{, formal-parameter-section});
procedure-or-function-declaration ::= procedure-declaration |

function-declaration
procedure-statement ::= procedure-identifier | procedure-identifier
(actual-parameter {, actual-parameter})
program-heading ::= empty **program** *program-identifier*
file-identifier-list
program ::= program-heading block
program-identifier ::= identifier
record-constant ::= (record-constant-element
{; record-constant-element})
record-contant-element ::= field-identifier: structured-constant
record-type ::= **record** *field-list* **end**
record-variable ::= variable
record-variable-list ::= record-variable {, record-variable}
referenced-variable ::= pointer-variable ^
relational-operator ::= = | <> | <= | >= | < | > **in**
repeat-statement ::= **repeat** *statement {; statement}* **until** *expression*
repetitive-statement ::= while-statement | repeat-statement | for-statement
result-type ::= type-identifier
scalar-type ::= (identifier {, identifier})
scale-factor ::= digit-sequence | sign digit-sequence
set ::= [{set-element}]
set-constant ::= [{set-constant-element}]
set-constant-element ::= constant | constant .. constant
set-element ::= expression | expression .. expression
set-type ::= **set of** *base-type*
sign ::= + | -
signed-factor ::= factor | sign factor
simple-expression ::= term {adding-operator term}
simple-statement ::= assignment-statement | procedure-statement |
goto-statement | inline-statement | empty-statement
simple-type ::= scalar-type | subrange-type | type-identifier
statement ::= simple-statement | structured-statement
statement-part ::= compound-statement
string ::= {string-element}
string-element ::= text-string | control-character
string-type ::= **string** *[constant]*
structured-constant ::= constant | array-constant | record-constant |
set-constant
structured-constant-definition ::= identifier: type = structured-constant
structured-statement ::= compound-statement | conditional-statement |
repetitive-statement | with-statement

structured-type ::= unpacked-structured-type
packed *unpacked-structured-type*
subrange-type ::= constant .. constant
tag-field ::= empty | field-identifier:
term ::= complemented-factor {multiplying-operator complemented-factor}
text-string ::= '{character}'
type-definition ::= identifier = type
type-definition-part ::= **type** *type-definition{;type-definition};*
type-identifier ::= identifier
unpacked-structured-type ::= string-type | array-type | record-type |
set-type | file-type
unsigned-constant ::= unsigned-number | string | constant-identifier | **nil**
unsigned-integer ::= digit-sequence | $ hexdigit-sequence
unsigned-number ::= unsigned-integer | unsigned-real
unsigned-real ::= digit-sequence.digit-sequence |
digit-sequence.digit-sequence E scale-factor |
digit-sequence E scale-factor
untyped-constant-definition ::= identifier = constant
variable ::= entire-variable | component-variable | referenced-variable
variable-declaration ::= identifier-list: type |
identifier-list: type **absolute** *constant*
variable-declaration-part ::= **var** *variable-declaration*
{; variable-declaration};
variable-identifier ::= identifier
variant ::= empty | case-label list: (field-list)
variant-part ::= **case** *tag-field type-identifier* **of** *variant {; variant}*
while-statement ::= **while** *expression* **do** *statement*
with-statement ::= **with** *record-variable-list* **do** *statement*

Appendix J
ASCII TABLE

0	00	^@	NUL	32	20	SP	64	40	@	96	60	`
1	01	^A	SOH	33	21	!	65	41	A	97	61	a
2	02	^B	STX	34	22	"	66	42	B	98	62	b
3	03	^C	ETX	35	23	#	67	43	C	99	63	c
4	04	^D	EOT	36	24	$	68	44	D	100	64	d
5	05	^E	ENQ	37	25	%	69	45	E	101	65	e
6	06	^F	ACK	38	26	&	70	46	F	102	66	f
7	07	^G	BEL	39	27	'	71	47	G	103	67	g
8	08	^H	BS	40	28	(	72	48	H	104	68	h
9	09	^I	HT	41	29	)	73	49	I	105	69	i
10	0A	^J	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	^K	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	^L	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	^M	CR	45	2D	-	77	4D	M	109	6D	m
14	0E	^N	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	^O	SI	47	2F	/	79	4F	O	111	6F	o
16	10	^P	DLE	48	30	0	80	50	P	112	70	p
17	11	^Q	DC1	49	31	1	81	51	Q	113	71	q
18	12	^R	DC2	50	32	2	82	52	R	114	72	r
19	13	^S	DC3	51	33	3	83	53	S	115	73	s
20	14	^T	DC4	52	34	4	84	34	T	116	74	t
21	15	^U	NAK	53	35	5	85	55	U	117	75	u
22	16	^V	SYN	54	36	6	86	56	V	118	76	v
23	17	^W	ETB	55	37	7	87	57	W	119	77	w
24	18	^X	CAN	56	38	8	88	58	X	120	78	x
25	19	^Y	EM	57	39	9	89	59	Y	121	79	y
26	1A	^Z	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	^[	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	^\	FS	60	3C	<	92	5C	\	124	7C	|
29	1D	^]	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	^^	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	^_	US	63	3F	?	95	5F	_	127	7F	DEL

Appendix L
INSTALLATION

L.1. TERMINAL INSTALLATION

Before you use TURBO Pascal, it must be installed for your particular terminal, i.e., provided with information regarding control characters required for certain functions. This installation is easily performed using the program *TINST* which is described in this chapter.

After making a work copy, please store your distribution diskette safely away and work only on the copy.

Now start the installation by typing *TINST* at your terminal. Select Screen installation from the main menu. A menu listing a number of popular terminals will appear, inviting you to choose one by entering its number:

```
Choose one of the following terminals:

 1) ADDS 20/25/30          15) Lear-Siegler ADM-31
 2) ADDS 40/60             16) Liberty
 3) ADDS Viewpoint-1A      17) Morrow MDT-20
 4) ADM 3A                 18) Otrona Attache
 5) Ampex D80              19) Qume
 6) ANSI                   20) Soroc IQ-120
 7) Apple/graphics         21) Soroc new models
 8) Hazeltine 1500         22) Teletext 3000
 9) Hazeltine Esprit       23) Televideo 912/920/925
10) IBM PC CCP/M b/w       24) Visual 200
11) IBM PC CCP/M color     25) Wyse WY-100/200/300
12) Kaypro 10              26) Zenith
13) Kaypro II and 4        27) None of the above
14) Lear-Siegler ADM-20    28) Delete a definition

Which terminal? (Enter no. or ^X to exit):
```

Figure L-2: Terminal Installation Menu

If your terminal is mentioned, just enter the corresponding number, and the installation is complete. Before installation is actually performed, you are asked the question:

Do you want to modify the definition before installation?

This allows you to modify one or more of the values being installed as described in the following. If you do not want to modify the terminal definition, just type **N**, and the installation completes by asking you the operating frequency of your CPU (see last item in this appendix).

If your terminal is **not** on the menu, however, you must define the required values yourself. The values can most probably be found in the manual supplied with your terminal.

Enter the number corresponding to **None of the above** and answer the questions one by one as they appear on the screen.

In the following, each command you may install is described in detail. Your terminal may not support all the commands that can be installed. If so, just pass the command not needed by typing RETURN in response to the prompt. If *Delete line, Insert line,* or *Erase to end of line* is not installed, these functions will be emulated in software, slowing screen performance somewhat.

Commands may be entered either simply by pressing the appropriate keys or by entering the decimal or hexadecimal ASCII value of the command. If a commnd requires the two charaacters 'ESCAPE' and '=', you may:

either: press first the **Esc** key, then the =. The entry will be echoed with appropriate labels, i.e., <ESC> =.

or: enter the decimal or hexadecimal vallues separated by spaces. Hexadecimal values must be preceded by a dollar-sign. For example, you could enter 27 61, or $1B 61, or $1B $3D, which are all equivalent.

The two methods cannot be mixed. Once you have entered a non-numeric character, the rest of that command must be defined in that mode, and vice versa.

A hyphen entered as the very first character is used to delete a command, and echoes the next *Nothing*.

Terminal type:

Enter the name of the terminal you are about to install. When you complete *TINST*, the values will be stored, and the terminal name will appear on the initial list of terminals. If you later need to reinstall TURBO Pascal to this terminal, you can do that by choosing it from the list.

Send an initialization string to the terminal?

If you want to initialize your terminal when TURBO Pascal starts (e.g., to download commands to programmable function keys), you answer **Y** for yes to this question. If not, just hit RETURN.

Send a reset string to the terminal?

Define a string to be sent to the terminal when TURBO Pascal terminates. The description of the initialization command above applies here.

CURSOR LEAD-IN command:

Cursor Lead-in is a special sequence of characters which tells your terminal that the following characters are an address on the screen on which the cursor should be placed.

When you define this command, you are asked the following supplemental questions:

CURSOR POSITIONING COMMAND to send between line and column:

Some terminals need a command between the two numbers defining the row and column cursor address.

CURSOR POSITIONING COMMAND to send after line and column:

Some terminals need a command after the two numbers defining the row and column cursor address.

Column first?

Most terminals require the ROW of the address first, then the COLUMN. If this is the case on your terminal, answer **N**.

If your terminal wants COLUMN first, then ROW, then answer **Y**.

OFFSET to add to LINE
Enter the number to add to the LINE (ROW) address.

OFFSET to add to COLUMN
Enter the number to add to the COLUMN address.

Binary address?
Most terminals need the cursor address sent in binary form. If that is true for your terminal, enter **Y**. If your terminal expects the cursor address as ASCII digits, enter **N**. If so, you are asked the supplementary question:

2 or 3 ASCII digits?
Enter the number of digits in the cursor address for your terminal.

CLEAR SCREEN command:
Enter the command that will clear the entire contents of your screen, both foreground and background, if applicable.

Does CLEAR SCREEN also HOME cursor?
This is normally the case; if it is not so on your terminal, enter **N**, and define the cursor HOME command.

DELETE LINE command:
Enter the command that deletes the entire line at the cursor position.

INSERT LINE command:
Enter the command that inserts a line at the cursor position.

ERASE TO END OF LINE command:
Enter the command that erases the line at the cursor position from the cursor position through the right end of the line.

START OF 'LOW VIDEO' command:
If your terminal supports different video intensities, then define the command that initiates the **dim** video here. If this command is defined, the following question is asked:

START OF 'NORMAL VIDEO' command:
Define the command that sets the screen to show characters in 'normal' video.

Number of rows (lines) on your screen:
Enter the number of horizontal lines on your screen.

Number of columns on your screen:
Enter the number of vertical column positions on your screen.

Delay after CURSOR ADDRESS (0-255 ms):
Delay after CLEAR, DELETE, and INSERT (0-255 ms):
Delay after ERASE TO END OF LINE and HIGHLIGHT On/Off (0-255 ms):
Enter the delay in milliseconds required after the functions specified. RETURN means 0 (no delay).

Is this definition correct?
If you have made any errors in the definitions, enter **N**. You will then return to the terminal selection menu. The installation data you have just entered will be included in the installation data file and appear on the terminal selection menu, but installation will **not** be performed. When you enter **Y** in response to this question, you are asked:

Operating frequency of your microprocessor in MHz (for delays):
As the delays specified earlier depend on the operating frequency of your CPU, you must define this value.

The installation is finished, installation data is written to TURBO Pascal, and you returned to the outer menu. Installation data is also saved in the installation data file, and the new terminal will appear on the terminal selection list when you run *TINST* in future.

L.2. EDITING COMMAND INSTALLATION

The built-in editor responds to a number of commands which are used to move the cursor around on the screen, delete and insert text, move text, etc. Each of these functions may be activated by either a primary command or a secondary command. The secondary commands

are installed by Borland, and comply with the standard set by WordStar. The primary commands are undefined for most systems, and may easily be defined to fit your taste or your keyboard, using the installation program.

When you hit **C** for Command installation, the first command appears:

```
CURSOR MOVEMENTS:

1: Character left    Nothing -> ■
```

This means that no primary command has been installed to move the cursor one character left. If you want to install a primary command (in **addition** to the secondary WordStar-like Ctrl-S, which is not shown here), you may enter the desired command following the -> prompt in either of two ways:

1) Simply press the key you want to use. It could be a function key (for example a left-arrow-key, if you have it) or any other key or sequence of keys that you choose (to a maximum of 4). The installation program responds with a mnemonic for each character it receives. If you have a left-arrow-key that transmits an <ESCAPE> character followed by a lower case **a**, and you press this key in the situation above, your screen will look like this:

```
CURSOR MOVEMENTS:

1: Character left    Nothing -> <ESC> a ■
```

2) Instead of pressing the actual key you want to use, you may enter the ASCII value(s) of the character(s) in the command. The values of multiple characters are entered separated by spaces. Decimal values are just entered: 27; hexadecimal values are prefixed by a dollar-sign: $1B. This may be useful to install commands which are not presently available on your keyboard, for example if you want to install the values of a new terminal while still using the old one.

This facility has just been provided for very few and rare instances, because there is really no idea in defining a command that cannot be generated by pressing a key. But it's there for those who wish to use it.

In both cases, end your input by pressing <RETURN>. Notice that the two methods cannot be mixed within one command. If you have started defining a command sequence by pressing keys, you must define all characters in that command by pressing keys and vice versa.

You may enter a - (minus) to remove a command from the list, or a **B** to back through the list one item at a time.

The editor accepts a total of 45 commands, and they may all be installed to your specification. If you make an error in the installation, like defining the same command for two different purposes, a self-explanatory error message is issued, and you must correct the error before terminating the installation. A primary command, however, may conflict with one of the WordStar-like secondary commands; that will just render the secondary command inaccessible.

The following table lists the secondary commands, and allows you to mark any primary commands installed by yourself.

CURSOR MOVEMENTS:

1:	Character left	Ctrl-S	_____
2:	Alternative	Ctrl-H	_____
3:	Character right	Ctrl-D	_____
4:	Word left	Ctrl-A	_____
5:	Word right	Ctrl-F	_____
6:	Line up	Ctrl-E	_____
7:	Line down	Ctrl-X	_____
8:	Scroll up	Ctrl-W	_____
9:	Scroll down	Ctrl-Z	_____
10:	Page up	Ctrl-R	_____
11:	Page down	Ctrl-C	_____
12:	To left on line	Ctrl-Q Ctrl-S	_____
13:	To right of line	Ctrl-Q Ctrl-D	_____
14:	To top of page	Ctrl-Q Ctrl-E	_____
15:	To bottom of page	Ctrl-Q Ctrl-X	_____
16:	To top of file	Ctrl-Q Ctrl-R	_____

17:	To end of file	Ctrl-Q Ctrl-C	_____
18:	To beginning of block	Ctrl-Q Ctrl-B	_____
19:	To end of block	Ctrl-Q Ctrl-B	_____
20:	To last cursor position	Ctrl-Q Ctrl-P	_____
		INSERT & DELETE:	
21:	Insert mode on/off	Ctrl-V	_____
22:	Insert line	Ctrl-N	_____
23:	Delete line	Ctrl-Y	_____
24:	Delete to end of line	Ctrl-Q Ctrl-Y	_____
25:	Delete right word	Ctrl-T	_____
26:	Delete character under cursor	Ctrl-G	_____
27:	Delete left character	<DEL>	_____
28:	Alternative:	Nothing	_____
		BLOCK COMMANDS:	
29:	Mark block begin	Ctrl-K Ctrl-B	_____
30:	Mark block end	Ctrl-K Ctrl-K	_____
31:	Mark single word	Ctrl-K Ctrl-T	_____
32:	Hide/display block	Ctrl-K Ctrl-H	_____
33:	Copy block	Ctrl-K Ctrl-C	_____
34:	Move block	Ctrl-K Ctrl-V	_____
35:	Delete block	Ctrl-K Ctrl-Y	_____
36:	Read block from disk	Ctrl-K Ctrl-R	_____
37:	Write block to disk	Ctrl-K Ctrl-W	_____
		MISC. EDITING COMMANDS:	
38:	End edit	Ctrl-K Ctrl-D	_____
39:	Tab	Ctrl-I	_____
40:	Auto tab on/off	Ctrl-Q Ctrl-I	_____
41:	Restore line	Ctrl-Q Ctrl-L	_____
42:	Find	Ctrl-Q Ctrl-F	_____
43:	Find & replace	Ctrl-Q Ctrl-A	_____
44:	Repeat last find	Ctrl-L	_____
45:	Control character prefix	Ctrl-P	_____

Table L-1: Secondary Editing Commands

Items 2 and 28 let you define alternative commands to *Character Left* and *Delete left Character* commands. Normally <BS> is the alternative to Ctrl-S, and there is no defined alternative to <DEL>. You may install primary commands to suit your keyboard, for example to use the <BS> as an alternative to <DEL> if the <BS> key is more conveniently located. Of course, the two alternative commands must be unambiguous like all other commands.

Appendix M
CP/M PRIMER

M.1. HOW TO USE TURBO ON A CP/M SYSTEM

When you turn on your computer, it reads the first couple of tracks on your CP/M diskette and loads a copy of the CP/M operating system into memory. Each time you re-boot your computer, CP/M also creates a list of the disk space available for each disk drive. Whenever you try to save a file to the disk, CP/M checks to make sure that the diskettes have not been changed. If you have changed the diskette in Drive A without rebooting, for example, CP/M will generate the following error message when a disk write is attempted:

BDOS ERROR ON A: R/O

Control will return to the operating system and your work was NOT saved! This can make copying a diskette a little confusing for the beginner. If you are new to CP/M, follow these instructions:

M.2. COPYING YOUR TURBO DISK

To make a working copy of your TURBO MASTER DISK, do the following:

1. Make a blank diskette and put a copy of CP/M on it (see your CP/M manual for details). This will be your TURBO **work disk**.

2. Place this disk in Drive A:. Place a CP/M diskette with a copy of PIP.COM in Drive B (PIP.COM is CP/M's file copy program, and should be on your CP/M diskette. See your CP/M manual for details).

3. Re-boot the computer. Type B:PIP and then press <RETURN>

4. Remove the diskette from Drive B: and insert your TURBO MASTER DISK.

5. Now type: A:=B:*.*[V] and then press <RETURN>

You have instructed PIP to copy all the files from the diskette in Drive B: onto the diskette in Drive A:. Consult your CP/M manual if any errors occur.

The last few lines on your screen should look like this:

```
A> B:PIP

*A:=B:*.*[V]

COPYING -
FIRSTFILE
:
:
LASTFILE
*
```

6. Press <RETURN>, and the PIP program will end.

M.3. USING YOUR TURBO DISK

Store your TURBO MASTER DISK in a safe place. To use TURBO Pascal, place your new TURBO **work disk** in drive A: and re-boot the system. Unless your TURBO came pre-installed for your computer and terminal, you should install TURBO (see Chapter 1). When done, type

TURBO

and TURBO Pascal will start.

If you have trouble copying your diskette, please consult your CP/M user manual or contact your hardware vendor for CP/M support.

Appendix N
HELP!!!

This appendix lists a number of most commonly asked questions and their answers. If you don't find the answer to *your* question here, you can either call Alpha's technical support staff, or you can access CompuServe's Consumer Information 24 hours a day and "talk" to the Borland Special Interest Group.

Q: How do I use the system?

A: Please read the manual, specifically Chapter 1. If you must get started immediately, do the following:

1) Boot up your operating system.
2) Run *TINST* to install TURBO for your equipment.
3) Run TURBO.
4) Start programming!

Q: I am having trouble installing my terminal!

A: If your terminal is not on the installation menu, you must define it to TINST. All terminals come with a manual containing information on codes that control video I/O. You must answer the questions in the installation program according to the information in your hardware manual. The terminology we use is the closest we could find to a standard. Note: most terminals do not require an initialization string or reset string. These are usually used to access enhanced features of a particular terminal; for example, on some terminals you can send an initialization string to make the keypad act as a cursor pad. You can put up to 13 characters into the initialization or reset strings.

Q: I am having disk problems. How do I copy my disks?

A: Most disk problems do not mean you have a defective disk. Specifically, if you are using a CP/M-80 system you may want to refer to Appendix M (Z-System computers don't issue a BDOS error in that situation). If you can get a directory of your distribution disk, then chances are that it is a good disk.

Q: Do I need an 8087 chip to use TURBO-87?

A: Yes, and in addition, TURBO-87 only works on CP/M-incompatible 16-bit systems running MS-DOS, PC-DOS, or CP/M-86.

Q: Do I need any special equipment to use TURBO-BCD?
A: No, but the BCD reals package only works on 16-bit implementations of TURBO that are not compatible with CP/M or the Z-System.

Q: Do I need TURBO to run programs I developed in TURBO?
A: No, TURBO can make .COM files.

Q: How do I make .COM files?
A: Type **O** from the main menu for compiler **O**ptions. In the compiler **O**ptions menu, select **C** for .COM files.

Q: What are the limits on the compiler as far as code and data?
A: The compiler can handle up to 64K of code, 64K of data, 64K of stack and unlimited heap. The object code, however, cannot exceed 64K.

Q: What are the limits of the editor as far as space?
A: The editor can edit as much as 64K at a time. If this is not enough, you can split your source into more than one file using the *$I* compiler directive. This is explained in Chapter 17.

Q: What do I do when I get error 99 (Compiler overflow)?
A: You can do two things: break your code into smaller segments and use the *$I* compiler directive (explained in Chapter 17) or compile to a .COM file.

Q: What do I do if my object code is going to be larger than 64K?
A: Either use the chain facility (see Chapter 22) or use overlays (Chapters 18 and 22).

Q: How do I read from the keyboard without having to hit return (duplicate BASIC's INKEY$ function)?
A: Like this: *Read(Kbd,Ch)* where *Ch:Char*.

Q: How can I get output to go to the printer?
A: You can use the following program. If you wish to have a listing that underlines or highlights reserved words, puts in page breaks, and lists all include files, there is one included free (including source) on the TURBO Tutor diskette.

```
program TextFileDemo;

var
  TextFile : Text;
  Scratch  : String[128];

begin
  Write('File to print: ');                { Get file name        }
  Readln(Scratch);
  Assign(TextFile, Scratch);               { Open the file        }
  {$I-}
  Reset(TextFile);
  {$I+}
  if IOresult <> 0 then
    Writeln('Cannot find ', Scratch);      { File not found       }
  else                                     { Print the file..     }
  begin
    while not Eof(TextFile) do
    begin
      Readln(TextFile, Scratch);           { Read a line          }
      Writeln(Lst, Scratch)                { Print a line         }
    end; { while }
      Writeln(Lst)                   { Flush printer buffer       }
  end { else }
end.
```

Q: How do I get output and input from COM1:?

A: Try: Writeln(AUX, ...) after setting up the port using an ASSIGN-type program from CP/M. To read try: Read(AUX, ...). You must remember that there is no buffer set up automatically when reading from AUX.

Q: How do I read a function key?

A: Function keys generate escape sequences (ESCAPE followed by one or more characters) or control characters, depending on what terminal you are using. There is no way to tell whether a given escape sequence or control character was generated by pressing a function key or was typed by the user.

Programs that depend on a particular set of function keys tend to make a program less general. It is better to define functions

in your program that are independent of a particular input, and then provide a separate program that prompts the user for the input he wants to stand for that function. In that way the user can easily customize your program for his preferences and his equipment. TURBO's *TINST* program is an example of this approach (see Appendix L).

Q: I am having trouble with file handling. What is the correct order of instructions to open a file?

A: The correct manner to handle files is as follows:

To create a new file:

```
Assign(FileVar,'NameOf.Fil');
Rewrite(FileVar);
:
:
Close(FileVar);
```

To open an existing file:

```
Assign(FileVar,'NameOf.Fil');
Reset(FileVar);
:
:
Close(FileVar);
```

Q: Why don't my recursive procedures work?

A: Set the A compiler directive off: {$A-}

Q: How can I use EOF and EOLN without a file variable as a parameter?

A: Turn off buffered input: {$B-}

Q: How do I find if a file exists on the disk?

A: Use {$I-} and {I+}. The following function returns *True* if the file name passed as a parameter exists, otherwise it returns *False*:

```
type
Name = string[66];
:
:
function Exist(FileName: Name): Boolean;
var
  Fil: file;
begin
  Assign(Fil,FileName);
  {$I-}
  Reset(Fil);
  {$I+}
  Exist := (IOresult = 0)
end;
```

Q: How do I disable CTRL-C?

A: Set its compiler directive off: {$C-}.

Q: I get a "type mismatch" error when passing a string to a function or procedure as a parameter.

A: Turn off type checking of variable parameters: {$V-}.

Q: I get a "file not found" error on my include file when I compile my program, even though the file is in the directory.

A: When using the include compiler directive {$I filename.ext} there must be a space separating the file name from the terminating brace, if the extension is not three letters long: {$ISample.F }. Otherwise the brace will be interpreted as part of the file name.

Q: Why does my program behave differently when I run it several times in a row?

A: If you are running programs in **M**emory mode and use typed constants as initialized variables, these constants will only be initialized right after a compilation, not each time you run the program, because they reside in the code segment. With .COM files, this problem does not exist, but if you still experience different results when using arrays and sets, turn on range checking {$R+}.

Q: I don't get the results I think I should when using *Reals* and *Integers* in the same expression.

A: When assigning an Integer expression to a Real variable, the expression is converted to Real. However, the expression itself is calculated as an integer, and you should therefore be aware of possible integer overflow in the expression. This can lead to surprising results. Take for instance:

```
RealVar := 40 * 1000;
```

First, the compiler multiplies integers 40 and 1000, resulting in 40,000 which gives integer overflow. It will actually come out to -25536 as Integers wrap around. Now it will be assigned to the RealVar as -25536. To prevent this, use either:

```
RealVar := 40.0 * 1000;
```

or

```
RealVar := 1.0 * IntVar1 * IntVar2;
```

to insure that the expression is calculated as a Real.

Q: How do I get disk directory from my TURBO program?

A: Sample procedures for accessing the directory are included in the TURBO Tutor package (order the TURBO Tutor from Borland).

INDEX

- C -

- D -

- E -

- F -

- G -

- H -

- I -

- N -

- O -

- P -

- Q -

- R -

- S -

- T -

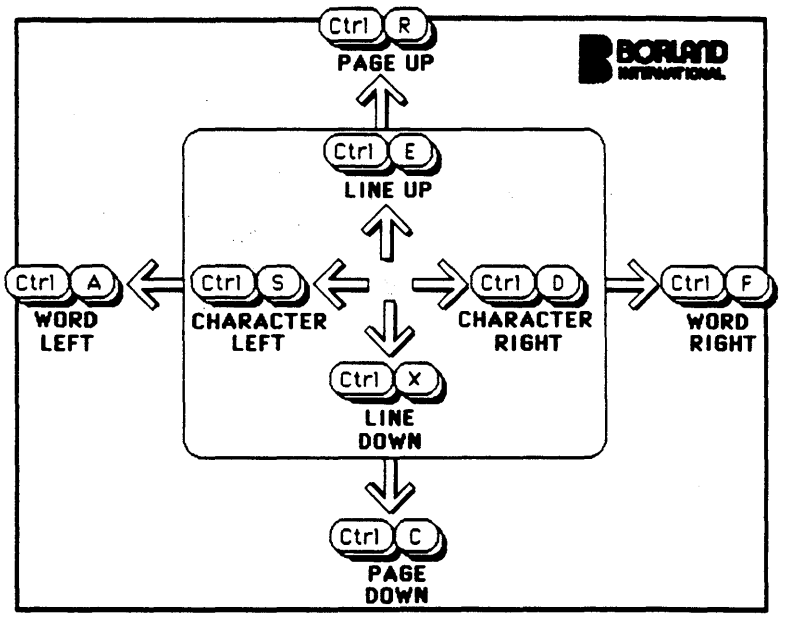
- U -

- V -

- W -

- X -

EDITOR QUICK REFERENCE



DELETE

Ctrl G — DELETE CHARACTER
Ctrl T — DELETE WORD
Ctrl Y — DELETE LINE

FIND

Ctrl Q F — FIND
Ctrl Q A — FIND & CHANGE
Ctrl L — REPEAT LAST FIND

OPTIONS: U =UPPER/LOWER CASE
W=WHOLE WORDS ONLY
B =BACKWARDS
G =GLOBAL
N =NO QUESTION

BLOCK

Ctrl K B ⟷ Ctrl K K
MARK BEGINNING — MARK END
Ctrl K T — MARK WORD
Ctrl K C — COPY BLOCK
Ctrl K V — MOVE BLOCK
Ctrl K Y — DELETE BLOCK

Ctrl K D — END EDIT