MODULA-2
TABLE OF CONTENTS

Chapter 7. Modules 97

Chapter 8. Low-Level Facilities 111

Chapter 9. Turbo Modula-2 Extensions 121

Chapter 12. Turbo Modula-2 ·Reference Directory 271

Appendices

Appendix A. Turbo **Modula-2 and** Turbo **Pascal** 549

Appendix B. Installation Procedures 571

Appendix C. Summary of Compiler Directives 583

Appendix D. Error Diagnosis 585

Appendix E. BNF Syntax for Turbo Modula-2 597

Introduction

This book is a reference manual for the Turbo Modula-2 system, implemented for the CP/M-80 operating system running on Z80 computers. Although there are many examples throughout the book, this is not a tutorial for Modula-2 programming; a basic knowledge of Modula-2 or Pascal is assumed.

The Modula-2 language was designed in 1980 by Niklaus Wirth, who also ored Pascal. While Modula-2 provides the powerful data and statement structures of Pascal, it also incorporates a modular structure as well as basic facilities for multiprogramming applications. Turbo Modula-2 is Borland's practical implementation of Modula-2. It follows closely the definition of standard Modula-2 as defined by N. Wirth in his book, *Programming in Modula-2* (3rd Ed. New York: Springer Verlag, 1984). However, it differs from Modula-2 in two main areas: It has an easy-to-use I/0 library and optional extensions.

Because of the wide range of possible external devices, and in order to make the language truly machine-independent, the strict definition of Modula-2 does not include any input/output or low-level facilities. Turbo Modula-2, however, provides an extensive and flexible library of these facilities, which for the most part follow Wirth's suggestions.

Turbo Modula-2 also includes several constructs not mentioned in the original definition; for example, general-purpose *READ* and *WRITE* statements, string comparison and assignment, multidimensional open arrays and exception handling.

Structure of This Manual

This manual is divided into four main areas: an introduction, the module library, a look-up section, and appendices.

Chapter 1 will get you started using the Turbo Modula-2 system, walking you through such operations as copying your distribution disk and running your first program.

Chapters 2 through 7 provide a thorough description of Turbo Modula-2 programming, with a general discussion of the Modula-2 language elements and a more detailed explanation of data, statement, and program structures, as well as local and library modules. Chapters 5 and 6 serve as your guide to writing simple Modula-2 programs.

Chapters 8 and 9 cover the more advanced topics in Turbo Modula-2, including system-specific functions and extensions and low-level facilities.

Chapter 10 covers system operations, including discussion of the editor, the compiler, the linker, the librarian, and other operational features.

Chapter 11 contains the library modules, which consist of 14 modules containing over 100 procedures. You may want to read the short description of each library module now, and go back to study the specifics when you need to use a particular module.

Chapter 12 comprises Turbo Modula-2's extensive alphabetical look-up section, which contains entries of reserved words standard identifiers, and library identifiers.

The five appendices provide information on the operation of Modula-2, while also presenting some comparisons between Turbo Pascal and Turbo Modula-2. Appendix A provides the language comparisons. Appendix B details the instructions for installing Turbo Modula-2 on your system. Appendix C summarizes the compiler directives discussed in Chapter 9. Appendix D lists error messages and their definitions/diagnoses. Appendix E provides the BNF syntax diagrams of Turbo Modula-2.

Typography

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

Alternate   Alternate characters are used to illustrate program examples and
               screen displays.

Italics   Italics are used to emphasize certain concepts and first-mentioned
               terms.

Boldface   Boldface is used to mark **reserved words** in **text as** well as in programming examples.

Acknowledgements

Several programs, languages, and operating systems are referenced in this manual; the following lists them and their respective companies.

☐ Turbo Pascal is a registered trademark of Borland **International.**

(WordStar is a registered trademark of MicroPro International.

☐ CP/M is a registered trademark of Digital Research Inc.

☐ Microsoft is a registered trademark of Microsoft Corp.

Chapter 1
Getting Started

Before Use

When you receive your Turbo Modula-2 disk, complete and mail in the license agreement at the front of this manual. This agreement allows you to make as many (.es as you need for your personal use and backup purposes only.

For your own protection, make a backup copy of the distribution disk with your file-copy or disk-copy program before you start using Turbo Modula-2. Make certain all files have transferred successfully, then store the original disk in a safe place. If anything happens to the backup copy, you can make a new backup copy from the original.

Files on Disk

The files you have just copied from the distribution disk to your backup disk are described here.

System Files

M2.COM   The Turbo Modula-2 system file contains the M-code inter-
              preter, the Overlay Manager, and the runtime system. Enter-
              ing the command M2 on your terminal will load this file and
              get Turbo Modula-2 up and running.

M2.OVR   Overlay file for M2.COM.

SHELL.MCD   The Turbo Modula-2 menu shell.

COMPILE.MCD   The Turbo Modula-2 M-code compiler.

GENZ80.MCD   The Turbo Modula-2 compiler's optional second-pass used
              for generating native Z80 code.

ERRMSGS.OVR   Text file containing error messages.

LIBRARY.MCD   A library manager that combines several compiled modules
                     into a single .LIB file.

SYSLIB.LIB   The standard modules supplied with the Turbo Modula-2
                     system.

LINK.MCD   The Turbo Modula-2 static overlay linker that is used to pro-
                     duce stand-alone (.COM) files.

Installation Files

INSTM2.COM   This installation program allows you to install the Turbo
                     Modula-2 compiler (M2.COM), as well as other .COM pro-
                     grams produced by the linker.

INSTM2.OVR   Installation program overlay file.

INSTM2.DTA   Installation data file compatible with Turbo Pascal's
                     TINST.DTA file (see Appendix B, »Installation Pro-
                     cedures«, for more details).

Example Files

*.MOD   Sample Modula-2 programs.

Other Files

READ.ME   If present, this file contains the latest corrections or sugges-
                     tions on the use of the system.

Utility Files

REL.MCD   A Turbo Modula-2 utility that converts Microsoft REL
                     (relocatable) object files into the .MCD format used by Tur-
                     bo Modula-2.

PROFILE.MCD   A Turbo Modula-2 utility that counts instructions in the
                various procedures of ar. M-code program. This utility can
                help you improve the efficiency of your Turbo Modula-2
                programs.

Preparing a Work Disk

The most effective way to use Turbo Modula-2 is to have the system files on your
boot disk, allowing you to use other drives for programs and data. To make your
working disk, follow these steps:

1. Prepare a system boot disk, which will become your work disk; also include
   a file-copy program such as PIP. For more information on this step, consult
   your operating system's user manual.

2. Now boot your system using the work disk, and place the Turbo Modula-2
   backup disk (the one you made after reading the first paragraph of this
   chapter) into a free disk drive.

3. Copy the following files from your **Turbo Modula-2 backup** disk to the **work**
   disk:

   M2.OVR
   M2.COM
   SHELL.MCD
   INSTM2.COM
   INSTM2.MSG
   INSTM2.DTA

   Now run the installation program by typing INSTM2 and pressing [RET] at
   the system prompt. This message will appear:

   ```
   Install program ([RETURN] for M2.COM):
   ```

   Press [RET] again and the following screen will appear:

```
              Modula-M2 system installation menu.
           Choose installation item from the following:


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

  Enter S, C, M, or Q:
```

5. Press ⌈ s ⌉ for Screen installation and pick the appropriate terminal type. For
   more details on installing your system, see Appendix B.

   Note: For Turbo Pascal owners with custom terminals, you may substitute
   your Turbo Pascal terminal data file for INSTM2.DTA by copying the Pascal
   data file to your work disk and renaming it from TINST.DTA to
   INSTM2.DTA.

6. Now press ⌈ M ⌉ for Miscellaneous. Tell the system which drives to search in
   for library and work files (generally, all your drives will be included).

   Note: Do not try to install the keyboard commands until you become more
   familiar with the editor.

7. Now select ⌈ Q ⌉ for Quit and delete the installation files from your work disk
   (INSTM2.*). Then delete the file-copy program we had you install in Step 1.
   Your bootable work disk should contain an installed copy of M2.COM,
   — M2.OVR, and SHELL.MCD.

8. With your work disk in drive A and your backup disk in drive B, type M2
   at the system prompt. The Turbo Modula-2 main menu will appear.

   Selected drive: A

   Work file:

   Edit   Compile   Run   eXecute

   Link   Options   Quit   liBrarian

   Dir   Filecopy Kill   reName   **Type**

   >

N te: Do not attempt to execute any menu items (except Filecopy) **until all** necessary files have been copied to the work disk.

9. To finish preparing your work disk, you must copy some files using Turbo Modula-2. To begin, press ⬚F⬚ for Filecopy at the main menu. Then at the "Copy from:" prompt, type in the name of the file you want to copy; for example:

Copy from: B:*.MCD [RET]

Press ⬚RET⬚ to get the "Copy to:" **prompt:**

Copy to: A: [RET]

Or do it in one step, like **so:**

Copy from: B:ERRMSGS.OVR A: [RET]

or

Copy from: B:SYSLIB.LIB A: [RET]

Copy the following files using any of the preceding procedures:

COMPILE.MCD
SYSLIB.LIB
ERRMSGS.OVR

And optionally, if there is room left on your disk, copy these files as well:

GENZ80.MCD
LIBRARY.MCD
LINK.MCD
REL.MCD
PROFILE.MCD

(The files M2.COM, **M2.OVR, and SHELL.MCD should already be on your** work disk.)

At this point, it would be wise to make a backup of your work disk to avoid having to reinstall the system. Now you're ready to try out the Turbo Modula-2 compiler.

Running Turbo Modula-2

Now that you've prepared a work disk and logged onto the drive containing it, you are ready to load Turbo Modula-2 into memory. After the system prompt, type M2, press ⌈RET⌋, and the following message will appear (but your terminal will be listed):

```
----------------------------------------------------------------
Turbo Modula-2 System                                Version 1.00
                                                     CP/M-80, Z80

Copyright(C) 1984, 1985,1986               Borland International
----------------------------------------------------------------


Terminal: No Terminal Selected

This will be followed by the main menu:

Selected drive: A

Work file:

Edit      Compile  Run        eXecute

Link      Options  Quit       liBrarian

Dir       Filecopy Kill       reName    Type

>
```

To run a program, place the backup disk containing the sample programs (*.MOD) in the B drive. At the main menu, press ⌈ c ⌋. You will see this message:

```
Workfile name:
```

Type Hello and press [RET]. Turbo Modula-2 will now compile the sample program. The compilation terminates with the following message:

```
Compiled bytes:  32
M-code file A00:HELLO.MCD produced.
```

Now press [R] and then [RET] to run the program. The screen will look like this:

```
(  ? Run MCD-file: A00:HELLO

 Hello World!

 >
```

After the prompt (>) appears in the main menu, quit Turbo Modula-2 by pressing [Q]; this will return you to the operating system.

Chapter 2
A Brief Tour of Turbo Modula-2

For those of you who want to learn Turbo Modula-2 quickly, or are already acquainted with some aspects of it, this chapter gets you into the thick of things in a hurry. Only a minimum of explanation is given here; to use Turbo Modula-2 to its full potential, you should refer to Chapter 10. This chapter will take you thr͡ ·h the menu system and briefly explain how to use each feature.

The Menu System

To start, place your work disk in the logged drive (to make a work disk, refer to Chapter 1), then type M2 and press ⟨RET⟩ to bring up the following screen:

```
--------------------------------------------------------------------------------
Turbo Modula-2 System                                        Version 1.00
                                                             CP/M-80, Z80


Copyright(C) 1984,1985, 1986                        Borland International
--------------------------------------------------------------------------------
```

Terminal: No terminal selected

This screen will be quickly followed by the main menu:
Selected drive: A

⟨ ᴜrk file:

Edit	Compile	Run	eXecute	
Link	Options	Quit	liBrarian	
Dir	Filecopy	Kill	reName	Type

>

The main menu contains all of the major functions performed by Turbo Modula-2. The following section describes each element of the main menu and displays a sample screen where appropriate. (Note: The italicized items in the example indicate that you must type in your own data.) To select each item, you press the letter highlighted on your screen.

Selected drive. Use this to change the default drive; press ⌐ S ⌐ and type the letter of the drive that is to become the new default drive. You can also use this function to reset the system or to log the disk; this is handy when you need to insert a new disk to copy files to.

```
>S
```

```
New drive: a: ___
```

Work file. Sets a default file name for other menu commands to use, such as Edit, Compile, Link, and so on. You can override this default by pressing the space bar before selecting the desired menu item (see the section »Avoiding the Menus« in Chapter 10).

```
>W
```

```
Workfile name: Myfile
```

Edit. Invokes the Turbo Editor, a WordStar-like editor. This editor is similar to the Turbo Pascal editor; however, it is a »one-pass« virtual editor that limits file sizes to the disk space available rather than internal memory.

```
>E
```

```
Edit file: COO:MYFILE.MOD
```

Compile. Turbo Modula-2 is an incremental compiler, which means it saves its state when a compile-time error is found. You can then enter the editor, correct the error and exit, at which point compilation continues at the closest block to where the error was corrected.

```
>C
```

```
Compile file: COO:MYFILE.MOD
```

.

Run. Runs compiled code without going through the linking step. The **Run** command does dynamic linking while it is reading in the support modules to run a program.

```
>R

Run MCD-file: COO:MYFILE
```

eXecute. Provides a way to run most external programs, such as **STAT** or even the Turbo Modula-2 installation program.

```
( ?ute COM-file: INSTM2.COM
```

Link. This command serves two purposes: It links separately compiled modules into a stand-alone .COM file and links specified modules so that they will load quickly when linked dynamically during the **Run** command. In addition, the linker provides the facility to generate overlays.

```
>L

Link main module: COO:MYFILE
```

Options. This command invokes the following Options menu:

```
compiler options:
```

List	(OFF)	Native	(OFF)	eXtensions	(ON)
Test	(OFF)	Overflow	(OFF)	Upper=lower	(OFF)

```
Path to search: SYSLIB

( .id run-time error

Save current selection        Quit

>
```

The compiler options are global; that is, they influence the entire compilation unless overridden with internal switches. You can toggle the compiler options by pressing the key for the capital (highlighted) letter in each option (for example, L for List). The following describes the function of each compiler option:

☐ List determines if source output is displayed on the monitor during compilation.

☐ Native determines which type of code is produced, M-code or Native.

☐ *e*Xtensions tells the compiler to issue warning messages if a program is using any of the Turbo Modula-2 extensions.

☐ Test determines if the compiler generates test code for array bounds and subrange checking.

☐ Overflow determines if the compiler generates code to check for integer overflow.

☐ Upper=lower tells the compiler whether or not to be case-sensitive.

☐ Path to search tells the system which library files to search when looking for external modules.

```
>P
```

```
New search path: SYSLIB MYLIB
```

☐ Find run-time error helps you find the runtime error position.

```
>F
```

```
Module name: COO:MYFILE.MOD
```

```
Enter PC: 23                                    .
```

☐ Save current selection allows you to save the options and search path you find most comfortable.

☐ Quit returns you to the main menu.

Now we'll get back to describing the main menu options.

Quit. Returns you to the operating system.

li**B**rarian. This command prompts you for a library **name** and then displays the librarian menu, as shown in the following:

```
>B
```

```
Select library: SYSLIB
```

```
Selected library: A00:SYSLIB.LIB
```

```
Dir      Include  Copy
```

```
(  .1     Ompress  Quit
```

The library keeps many precompiled library modules in one file. The compiled versions of the definition module and the implementation module are contained in the library as .SYM files and .MCD files, respectively.

Notice that the librarian menu has a different prompt, distinguishing the following library management utilities from the main menu's file- management routines. Again, initiate each option by pressing the capital (or highlighted) letter of its name.

☐ *Selected library* prompts for a name prior to displaying the librarian menu. Press ⟨ **S** ⟩ to change the selected library without returning to the main menu.

```
*S
```

```
Select library:
```

⟨ *Dir* lists the .SYM and .MCD files, that are stored in the selected library. In addition, it lists the size of each file and the cumulative size of all the files.
```
*D
```

```
Directory of library: C00.SYSLIB
    1: COMLINE     .MCD    0.5K .SYM    0.5K
    2: CONVERT     .MCD    2.0K .SYM    1.0K
    3: DOUBLES     .MCD    3.5K .SYM    1.0K
    4: FILES       .MCD    3.0K .SYM    1.5K
    5: INOUT       .MCD    1.0K .SYM    1.0K
    6: LOADER      .MCD    2.5K .SYM    1.0K
    7: LONGMATH    .MCD    2.0K .SYM    0.5K
    8: MATHLIB     .MCD    1.5K .SYM    0.5K
    9: PROCESSE    .MCD    0.5K .SYM    0.5K
   10: STRINGS     .MCD    0.5K .SYM    1.0K
   11: TERMINAL    .MCD    1.0K .SYM    1.0K
   12: TEXTS       .MCD    2.0K .SYM    2.0K
       Total size:        39.5K Unused: 8.0K
```

☐ *Include* takes external **files (either .SYM** files or .MCD) **and** places them in the library.

```
*I
```

```
   Include file:
```

☐ *Copy* copies files from the selected library to either another library or to a stand-alone .SYM or .MCD file.

```
*C
```

```
Copy module:
```

☐ Kill erases a specified library module.

```
   *K            -
```

```
Kill module:
```

☐ *c*Ompress eliminates unnecessary space in the library files. Since this command can take some time, we recommend using it only on stable (debugged) library files.

```
*O
```

Compressing library, please wait.

```
*
```

□ Quit returns you to the main menu.

The remainder of the main menu options described here are file-management commands.

Dir. The directory command allows you to display a full or partial directory of any disk. It accepts drive name, user areas, and wildcard file names (such as *.MOD or *.*) to build the directory listing. When a directory listing is displayed, it is shown with a number before each file name. These numbers may be used in subsequent file-management commands, such as Filecopy and Kill.

```
>D
```

Directory mask:

1:	COMPILE	.MCD	4:	LIBRARY	.MCD	7:	M2	.OVR	10:	SHELL	.MCD
2:	ERRMSGS	.OVR	5:	LINK	.MCD	8:	MYFILE	.BAK	11:	SYSLIB	.LIB
3:	GENZ80	.MCD	6:	M2	.COM	9:	MYFILE	.MOD			

Bytes Remaining on A: 96K

□ **Filecopy.** This command will accept a drive name, a user number, and a wildcard file name as the source file. In addition, the source file may be a list or range of numbers that reference the last directory command.

```
>F
```

```
Copy from  :  1 3-5
Copy to    :  b:
```

□ **Kill.** This command allows you to delete disk files without leaving the shell. It also accepts drive names, user numbers, and wildcard file names, as well as a list or range of numbers referencing the last directory command.

```
>K
Kill file: *.bak
Deleting D00:MYFILE.BAK
```

☐ reName. This command accepts one new file name (with optional drive and user area) for the old name and one file specification for the new name.

```
>N

Rename from :  myfile.mod
Rename to   :  myfile.def
```

☐ Type This command displays the specified file on the screen. You can pause output by pressing Ctrl S and terminate output by pressing Ctrl C .

```
>T

Type file: myfile.def

DEFINITIONMODULE MyFile;
  PROCEDUREMyProc;
ENDMyFile.
```

Chapter 3
Language Elements

Language elements are the building blocks that form a program and are the fundamental units recognized by the compiler. Each element is used to build a different level of abstraction. For example, a digit is an element that can be combined with other digits to form a number that is an element at a different level of abstraction. Thus, we can continue abstracting until a level is reached where an entire program is represented by one word, such as MyProgram.

This chapter describes the elements used to assemble meaningful program statements in Turbo Modula-2. In the following section, we will look at the first level of abstraction used by Turbo Modula-2; namely, the characters that comprise symbols.

Character Set

The Turbo Modula-2 character set includes all characters that are legal in Modula-2 declarations, expressions, and statements. This set is made up of alphabetic, numeric, and special characters.

Alphabetic (uppercase and lowercase)

   A to *Z* and *a* to *z*

Numeric

   0 1 2 3 4 5 6 7 8 9

Special Characters

   ~ + - * / = ^ < > ( ) [ ] { } . , : ; & # ' " |

Not all characters are available on all terminals; thus Turbo Modula-2 recognizes the following synonyms:

(.	for	[	Left parentheses and period for left index bracket
.)	for	]	Period and right parentheses for right index bracket
(:	for	{	Left parentheses and colon for left brace
:)	for	}	Colon and right parentheses for right brace
!	for	I	Exclamation mark for vertical bar

Using these available symbols, it is possible to create a great number of Modula-2 programs.

Vocabulary

The Modula-2 compiler recognizes certain groups of characters as symbols themselves, including numbers, characters, strings, delimiters, operators, and reserved words and identifiers. These symbols (or characters) comprise the second level of abstraction, providing you with the means to form sentences and thus programs in Modula-2.

The Turbo Modula-2 vocabulary can be divided into five classes: numbers, strings and characters, delimiters and comments, operators, and reserved words and identifiers (user-defined, standard, and library).

Numbers

There are two types of numbers defined in Modula-2: whole numbers and real numbers. Turbo Modula-2 recognizes three subtypes of whole numbers and two subtypes of real numbers.

Whole Numbers

Turbo's three types of whole numbers may be further defined as two signed numbers and one unsigned number. The two signed whole numbers are single precision and double precision, which are INTEGER and LONGINT, respectively. The unsigned whole number is single precision and is called a CARDINAL.

Following are the ranges for whole numbers:

Type		Range		
INTEGER	(single-precision, signed whole number)	-32,768	to	32,767
LONGINT	(double-precision, signed whole numbers)	-2,147,483,648	to	2,147,483,647
CARDINAL	(single-precision, unsigned whole numbers)	0	to	65,535

Single-precision integers can be formed in any of these three bases: decimal, octal, or hexadecimal (hex). Decimal numbers consist of the digits 0 to 9. Octal numbers comprise the digits 0 to 7, followed by the letter *B*. Hex numbers consist of the digits 0 to 9 and the letters *A* to *F*. The first character in a hexadecimal number must be a digit and the hexadecimal number must end with the letter *H*. The following are examples of legal and illegal single-precision whole numbers:

Legal (base)

Single	Precision	Illegal	Single Precision
1986	(Decimal)	12.34	Decimal point illegal.
-10	(Decimal)	FH	First character must be a digit.
10B	(Octal)	08B	8 is not a legal octal digit.
0FFFFH	(Hex)	0F	Requires *H* for hexadecimal.
1AH	(Hex)	0GH	G is not a legal hexadecimal digit.
62345	(Decimal)	-60000	Too small for single-precision integer.

Double-precision integers, or long integers, can only be formed in decimal notation, followed by the letter *L*. Following are legal and illegal long integers:

Legal Double Precision	Illegal Double Precision	
123L	123	This is a normal integer.
456L	45.6	This is a real number.
120392237L	0FFFFFFFFH	Only decimal base allowed.

Real Numbers

Single-precision and double-precision numbers are the two types of real numbers recognized by Turbo Modula-2. They are called REAL and LONGREAL, respectively.

Single-precision real numbers can be accurate up to 6 decimal points, while double-precision reals can be accurate up to 14 decimal points. The ranges of REAL and LONGREAL are as follows:

Type		Range	
REAL	-6.80565E+38	to	6.80565E+38
LONGREAL	-3.5953862697246D+308	to	3.5953862697246D+308

As shown in the previous ranges, a real number contains a sequence of numeric characters, containing a decimal point with an optional *scale factor*. The scale factor is specified by the letter E or D (E for single precision and D for double precision) and an integer, beginning with an optional plus (+) or minus (-) sign. The following are samples of legal and illegal real numbers:

Legal Real Numbers		Illegal Real Numbers	
12.34	Single precision	1 453	Blank space illegal
0.1E3	Equals 100.0	12	Needs decimal point
3.6E-5	Equals 0.000036	3.6E-99	Exponent too large
12.34D0	Double precision	1,423.0	Comma is illegal
0.1D3	Equals 100.0	12d-10	D and E must be uppercase
3.6D-5	Equals 0.000036	12.0D-999	Exponent too small

Strings and Characters

Strings
A *string* is a portion of text that can be handled as a single unit, such as a message to be written on the screen. Strings provide a means of manipulating text within a program. Constant strings are formed by enclosing a sequence of printable characters in either single or double quotation marks. The opening and closing quotation marks must be of the same kind, and that kind cannot occur within the string. Here are some examples of legal and illegal constant strings:

Legal Constant Strings	Illegal Constant Strings	
'My name is:'	"Hello'	Quotes must match
'He said, "What is the time?"'	"She said, "The	Opening quote type
"12 34"	time is 4 p.m."""	cannot occur within
		the string

Characters

A *character* is a single, printable ASCII symbol enclosed in matching quotes (single or double), or an octal integer (up to 377) followed by the letter *C*. In theory, characters occupy 1 byte of storage, and thus may take on any value between 0 and 255 decimal (or 377 octal). In practice, characters may occupy an entire word (2 bytes) in memory. The only time a character actually takes 1 byte is when it is declared as part of an array (arrays pack characters together). Any other declaration involving a character always results in the character occupying 2 bytes. Following are legal and illegal characters:

Legal Characters		Illegal Characters	
"A"	The letter A	"AB"	Only one character allowed
'*'	Asterisk	"Q'	Must have matching quotes
15C	Carriage return	8C	8 is not an octal digit
377C	Largest character	777C	Octal number too large for character

When a string contains only one character enclosed in quotes, the string is considered a special case and is compatible with a character. This only applies to constant strings like "A" or '*', not character variables.

Delimiters and Comments

Delimiters

A *delimiter* is one or more characters that separate other syntactic entities. For example:

Blanks separate identifiers and reserved words.

The vertical bar separates CASE, exception, and variant record statements.

,	The comma separates items in parameter lists and declarations.
;	The semicolon separates program statements and declarations.
*, +, -, /	The mathematical symbols (or operators) also serve as delimiters between operands.
Comments	A comment may appear anywhere one of the preceding delimiters is allowed. In addition, a comment may serve as a substitute for a blank space between symbols.

Comments

Comments contain descriptive or explanatory text about a program. Since, for the most part, they are ignored by the compiler (except for compiler switches, see Chapter 10), they can contain any characters enclosed by a set of parentheses and asterisks.

   Unlike Pascal, which allows either the symbol sets (* and *) or { and } to delimit comments, Modula-2 uses only (* and *) to delimit comments and uses { and } exclusively to delimit sets. Comments may be nested to any depth (limited only by memory space). They may occur anywhere a delimiter is allowed, and are treated as blanks. The following is a sample of legal and illegal comments:

Legal Comments	Illegal Comments	
`(* This is a comment *)`	`{This isn't a comment}`	`Braces not allowed`
`(*Comments can be nested (* like this *) *)`	`(* Hi (* there *)`	`Missing second closing comment`

Operators

Operators in Modula-2 are made up of either special symbols or reserved words. In the case of reserved words, they are always printed in uppercase letters (see "Compiler Switches," in Chapter 10) and cannot be used as identifiers. The following lists the special symbol and function of each operator:

+	Addition and set union
-	Subtraction and set difference
*	Multiplication and set intersection
/	Division and symmetric set difference
&	Logical AND
~	Logical NOT
^	Dereferencing
:=	Assignment
=	Equality
# < >	Inequality
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal
( )	Parentheses
[ ]	Index brackets
{ }	Set braces
. , : ; .. : ' "	Punctuation

In addition to these special symbols, the following reserved words are also operators:

AND	Logical AND
OR	Logical OR
NOT	Logical NOT
IN	Set membership
DIV	Integer division
MOD	Integer modulus

Operators are closely related to the operands they work on. In most cases, an operator will only work with certain kinds of operands; for example, to divide real numbers you must use the real division operator: 2.8/2.9. In addition, the result of an operation could be of a different type from the original operands; for example, comparing two integers results in a Boolean value: 1 = 2. (Chapter 4 discusses operators and operands in more depth.)

Reserved Words

Reserved words in Modula-2 are symbols with a predefined and static meaning;

they cannot be used for any other purpose. They are always written in uppercase letters.

AND	ELSIF	LOOP	REPEAT
ARRAY	END	MOD	RETURN
BEGIN	EXIT	MODULE	SET
BY	EXPORT	NOT	THEN
CASE	FOR	OF	TO
CONST	FROM	OR	TYPE
DEFINITION	IF	POINTER	UNTIL
DIV	IMPLEMENTATION	PROCEDURE	VAR
DO	IMPORT	QUALIFIED	WHILE
ELSE	IN	RECORD	WITH

The following Turbo Modula-2 reserved words provide useful extensions to the Modula-2 standard:

```
EXCEPTION   FORWARD   RAISE
```

In this manual (and other Borland manuals), all reserved words are in boldface type; however, they will not appear on your screen this way.

The preceding symbols and reserved words will be discussed in the next few chapters. For an immediate discussion of reserved words, refer to Chapter 12, "Turbo Modula-2 Reference Directory."

Identifiers

Identifiers are unique names given to constants, types, variables, procedures, and modules. Identifiers are sequences of alphabetic and numeric characters; the first character must always be a letter. Note that Turbo Modula-2 distinguishes between uppercase and lowercase letters; thus, unlike in Pascal, in Turbo Modula-2 *Var1* and *VAR1* are considered two unique identifiers.

Standard Identifiers

Turbo Modula-2 has a number of predefined identifiers for special purposes. Standard identifiers are "visible" and available in all modules without explicitly importing them. The standard identifiers can be redefined, but doing so will cost you the function offered by the predefined identifier. Also, a redefined standard

identifier only affects the module where it is redefined, not other imported modules. (Again, note that the use of uppercase and lowercase is significant.)

ABS	DISPOSE	INT	ODD
BITSET	DOUBLE	INTEGER	ORD
BOOLEAN	EXCL	LONG	PROC
CAP	FALSE	LONGINT	REAL
CARD	FLOAT	LONGREAL	SIZE
CARDINAL	HALT	MAX	TRUE
CHAR	HIGH	MIN	TRUNC
CHR	INC	NEW	VAL
DEC	INCL	NIL	

The **following standard** procedures provide extensions to the Modula-2 standard:

READ　　　　**READLN**　　　**WRITE**　　　**WRITELN**

User-Defined Identifiers

Users may declare their own identifiers; however, there are two rules to follow: Identifiers must begin with an alphabetic character and may consist only of alphabetic and numeric characters; spaces, underscores, and other special characters are not allowed. The following is a sampling of legal and illegal user-defined identifiers:

Legal Identifiers	Illegal Identifiers	
MyName	My Name	
		Blank space illegal
Tr612	Public-Transit	Hyphen illegal
hello	1st	First character must **be a letter**
Modula2	LAST__PRIME	Underscore illegal

Library Identifiers

Turbo Modula-2 contains an extensive library (in SYSLIB.LIB) with a number of predefined modules. The modules and their identifiers can be redefined, but doing

so will cause you to lose the function offered by them, or at least make using them more awkward.

In one sense, library identifiers are no different than user-defined identifiers since you can create libraries with new identifiers, thus adding to the list of available library identifiers. However, there are two differences.

One is that Turbo Modula-2 provides a set of standard identifiers that are necessary for programming in Modula-2. We presume that some of these standard identifiers are in every implementation of Modula-2, while we have added others for machine-specific reasons. The second difference is that Turbo Modula-2 "comprehends" the functions of system-dependent library modules, such as *SYSTEM* and *STORAGE*. Thus, when an item is used from the machine-dependent *SYSTEM* module, the compiler already knows about it and has no need to look at additional symbol files to determine usage. Table 3-1 is a list of library identifiers.

Table 3-1

ADDRESS	EndError	Open	SpecialOps
ADR	Entier	OpenInput	Sqrt
ALLOCATE	EOF	OpenOutput	StartProcess
Append	EOL	OpenText	StatusError
Arctan	EOLN	OpSet	STORAGE
Argument **Error**	EOT	OUT	String
available	ExitScreen	outName	StringError
Awaited	Exp	OUTOFMEMORY	Strings
BDOS	FILE	output	StrToCard
BIOS	Files	OVERFLOW	StrToDouble
BusyRead	FileSize	Pos	StrToInt
BYTE	FILL	PROCESS	StrToLong
Call	firstDrive	Processes	StrToReal
CAPS	Flush	progName	SYSTEM
CardToStr	FREEMEM	PromptFor	termCH
clearEol	GetName	Random	Terminal
ClearScreen	GotoXY	Randomize	TEXT
ClearToEOL	haltOnControlC	READ	TextDriver
Close	Highlight	ReadAgain	TextFile
CloseInput	highlightNormal	ReadByte	TextNotOpen
CloseOutput	HI.RESULT	ReadBytes	Texts
CloseText	Init	ReadCard	TooLarge
CODE	InitScreen	ReadChar	TooManyTexts

Col	inName	ReadDouble	TRANSFER
ComLine	InOut	ReadInt	TSIZE
commandLine	INP	ReadLine	UseError
ConnectDriver	Insert	ReadLn	WAIT
console	insertDelete	READLN	WORD
Convert	InsertLine	ReadLong	Write
Copy	IntToStr	REALOVERFLOW	WRITEBYTE
Cos	IORESULT	ReadReal	WRITEBYTES
Create	IOTRANSFER	ReadRec	WriteCard
CreateText	legal	ReadString	WriteChar
DeadLock	Length	ReadWord	WriteDouble
DEALLOCATE	Ln	RealToStr	WriteHex
Delete	Loader	RedirectInput	WriteInt
DeleteLine	LoadError	RedirectOutput	WriteLn
DeviceError	LongMath	RELEASE	WRITELN
DiskFull	LongToStr	Rename	WriteLong
DISPOSE	MARK	ResetSys	WriteOct
Done	MathLib	SEND	WriteReal
Doubles	NEW	SetCol	WriteRec
DoubleToStr	NEWPROCESS	SetPos	WriteString
	NextPos	SIGNAL	WriteWord
	Normal	Sin	
	NoTrailer	SIZE	
	numCols		
	numRows		

For more detailed information regarding library modules, refer to Chapter 11, "The Standard Library." For information regarding the preceding library identifiers, refer to Chapter 12, "Turbo Modula-2 Reference Directory."

Chapter 4
Expressions

An *expression* is a sequence of language elements that combine to form a temporary data object with a possibly different value and type than either original element. An expression consists of operators and operands that perform certain operations when an expression is "evaluated."

In the previous chapter, we looked at the character set and vocabulary used to build Modula-2 programs. Here we'll take the next step by showing you how to form expressions with language elements, while also providing the foundation for our discussion of abstract data types in the next chapter.

Properties of an Expression

To explain expressions, let's look at a simple example from mathematics:

```
1 + 2
```

1. The expression consists of operands and an operator: The two operands are 1 and 2, and the operator is +.

2. Expressions have a result value that can be assigned or used in further expression evaluation. The preceding example has a result value of 3.

3. addition, the expression is considered to have a type that is determined by the result. In this example, the result type is CARDINAL.

Expressions are evaluated by applying each operator to its operands. In general, expressions are evaluated from left to right. Operators that take precedence over others are executed first, despite the position of the operator within the expression (see the later section, "Operator Precedence").

Operands

In Turbo Modula-2, the operands of an expression can be any *elementary type* (such as INTEGER or REAL), including function procedures that return an elementary type (such as standard function procedures *ABS*, *CHR*, or *SIZE*). However, operands within a particular expression must be of the same type; for instance, in the previous example both operands are integers. The following shows an illegal expression, one that mixes integers and reals:

```
3.4 + 30
```

The purpose of this type-checking is to make explicit the types of operands involved in the expression. Unlike Pascal and some other languages, Modula-2 in general has no implicit type conversions; however, there are ways to override this strict type-checking. The explicit conversion in the following example makes the preceding example legal:

```
3.4 + FLOAT(30)
```

The result of this expression is 33.4 and is of type REAL. Note that it is possible for the result type to be different from its operands. Consider the following:

```
1 = 2
```

Here is an expression containing integer operands and one operator. When the operator is applied to the operands, the resulting value is FALSE and the result's type is BOOLEAN.

Operators

The most distinguishing feature of an **operator** is the type (or types) it works on. There are four classes of operator:

Arithmetic   Performs normal mathematical operations. Operations **are** performed on numbers, such as INTEGER and REAL.

Relational   Does a comparison of like items to obtain a BOOLEAN result of TRUE or FALSE. The items compared can be of any elementary type; for example, INTEGER, POINTER. string, and so on.

Logical   Performs the combination and the negation of BOOLEAN
            expressions. For example, the expression, **NOT** Raining
            AND (Today=Friday) combines the negation of *Raining* and
            the truth of (*Today=Friday*).

Set   Performs bit operations on operands of type BITSET or
            logical set operations on user-defined **SET** types; for exam-
            ple, you may wish to mask the high bit of bytes in a WordStar
            document file, or you may want to include the enumerated
            value *Red* in a user-defined set of colors.

Since the selection of operators is dependent on the type they're operating on,
we'll defer further description of each operator until its type is introduced in
Chapter 5.

Operator Precedence

When more than one operator appears in an expression, its meaning or result is
dependent on the order of evaluation. Operations of highest precedence are
resolved first; operations of equal precedence proceed from left to right. Opera-
tions within parentheses are of the highest priority, and thus are evaluated first.
Operator precedence is defined as follows:

lst (Highest)	priority:	**NOT, ~**
2nd	priority:	***, /, DIV, MOD, AND, &**
3rd	priority:	**+, -, OR**
4th (Lowest)	priority:	**=, #, <, >, >=, >=, <>, IN**

As an example, let's look at a simple mathematical expression:
(
· 1.0 + 2.0 * 3.0

Evaluation begins on the left with 1.0 + 2.0; however, this subexpression is not
fully evaluated until the priority level of the next operator is determined. The
previous table, indicates that multiplication has a higher priority than addition;
thus the subexpression 2.0 * 3.0 is evaluated first, with its product added to 1.0
for a result of 7.0 (type REAL).

Parentheses can alter the interpretation of operator precedence. Whene'
parantheses are found in an expression, the contents within the parantheses ;
evaluated before being combined with any other item in the expression.

If the prior example is changed to include parantheses, as shown in the follc
ing:

```
(1.0 + 2.0) * 3.0
```

then evaluation will start with the contents of the parantheses, and the sum v
be multiplied by 3.0. This time the result will be 9.0 (type REAL).

Boolean expresssions are evaluated almost the same as other expressio
However, in Boolean expressions, it is not always necessary to evaluate the en
expression in order to determine that expression's result. This is called *sh(*
circuit evaluation. For example, consider the following Boolean expression:

```
(2+4=4) AND (3+4=7)
```

To begin evaluation, we would first resolve the contents of the first parenthe
(2+4=4); its result is FALSE, which is saved. The AND operator is next,
because both operands to an AND must be TRUE for the entire expression tc
TRUE, we need only evaluate the contents of the second set of parentheses if
first operand is TRUE. In this case, the first operand of the AND is FALSE
the second operand to AND is not evaluated (the expression is short-circuit(

· This order of evaluation is useful for certain algorithms that need to acces:
element of an array onl' if the index is not a number that would cause an ai
bounds error. Thus, assuming you understand arrays, the following expres:
would *not* cause a bounds error if I is 0 and the array a is undefined when
equal to 0:

```
(I # 0) AND (a[I]=100)
```

Other Boolean operators, such as OR, are evaluated in a similar manner. T|
if the first operand of an OR expression is TRUE, the second operand is|
evaluated.

Chapter 5
Data Structure

Modula-2 is often described as a language that encourages structured programming. It permits a large program to be broken down into smaller, more manageable sections that can be separately compiled into object code. Within a program, objects are structured into logical units that make them easier to manipulate.

Modula-2 has, as you might expect, many components that join to form a modern block-structured language. As Wirth has pointed out, programs are a combination of algorithmic structures and data structures. Modula-2 has well-defined constructs for expressing both structures. In this chapter, we will look at object manipulation in programs and the data structures used to describe these objects.

Data Objects

Data objects are the information a program processes. This information can be numbers, characters, or anything the program problem requires. Every data object in your Modula-2 program has the following properties:

Identifier   A unique name distinguishing one data item from another. An identifier is given to most variables, any defined types, many constants, and program parts. Identifiers (except for standard identifiers) must be defined in a declaration.

Value   Each data item has a value. This value may be constant and unchangeable throughout the program or the data item may take on different values at different times during the course of the program.

Type   Each data item has a type that determines its use. The *type* of a data item dictates the values it can be assigned and the operations that can be performed on it.

As an example, suppose your program includes the variable *Count*, which can take any whole number as a value. The variable *Count* has the following properties:

☐ *Count* is the *identifier*, or unique name, of this data item.

☐ The *value* of the variable *Count* is determined at runtime by an assignment and may be changed at any point.

☐ The *type* of *Count* is INTEGER, which means that it may receive only the values in the range -32K to 32K and that only integer math operations may be performed on it.

There are two categories of types: *unstructured* and *structured*. First, we will look at the elementary data types in Modula-2, which are predefined unstructured types. Then we will see how declarations define a data item's type, identifier, and value. Next, we will discuss user- defined unstructured types. And lastly, for more complicated objects, we will examine the building blocks Modula-2 provides to define larger structures--structured types.

Elementary Data Types

In the previous chapter, we described some of the objects that might be manipulated in a program (numbers, strings, and so on). In addition, we showed that an object, such as a number, may have different representations called types. For simple items, Modula-2 has predefined names that represent these types; for example, INTEGER, LONGINT, REAL, and so forth. These are called elementary data types.

The following eight types are predefined in Modula-2. They are considered basic types and are always available. Generally, variables are declared as one of these types; however, you can also build new types from these basic types.

CHAR type: Range [0C to 377C]. Takes a single character as a value. A CHAR constant is denoted by enclosing a printable ASCII character in single or double quotation marks, or by specifying the octal value of the ASCII character followed by a *C* (as in the range given here). Sample CHAR values include the following

"A" '2' "#" ' " ' 1C 377C

BOOLEAN type: Range [FALSE to TRUE]. Assumes one of two logical truth states denoted by the standard identifiers TRUE and FALSE. Use BOOLEAN-type variables when you expect a yes/no- or on/off-type answer. For example, "Is it raining?" has a yes/no answer that could be represented as

```
Raining = TRUE
```

  or

```
Raining = FALSE
```

CARDINAL type: Range [0 to 65535]. Use CARDINAL-type variables anytime values are limited to positive whole numbers or zero, such as for a person's age or address, or lengths and distances. The type CARDINAL is used more frequently than the type INTEGER, and a CARDINAL's range is also double that of an INTEGER in the positive direction. Sample CARDINAL values:

```
8088   2132   0   64000
```

INTEGER type: Range [-32768 to 32767]. Use the INTEGER type when values are expected to drop below zero (for instance, in temperature ranges). In practice, this type is not used often since most values in programs are non-negative. Examples of INTEGER values:

```
8088   -2132  0
```

BITSET type: Range [0..15] (bit-wise). BITSET is a predefined set type (see "Set Types" later in this chapter), whose primary use is in performing bit-wise operations (such as masking) on word-length variables. BITSETs (and sets in general) has several predefined operations. Operators provided are *union*(+), *intersection*(-), *difference*(*), and *symmetrical difference*(/). In addition, there are standard procedures (*INCL* and *EXCL*) that are used to include and exclude any element from a set variable, and a reserved word (IN) to test whether a particular element is a member of a set element.

Sets are designated by enclosing the members in curly brackets, like this: {3,4,6}. You can visualize the type BITSET by comparing the set notation to its binary representation:

```
                                  { }  =  0000000000000000
                 {4,5,6,7,12,13,14,15}  =  1111000011110000
 {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15}  =  1111111111111111
```

In the first example, all the bits are clear which means there are no members in the set. In the second example, bits 4 through 7 and bits 12 through 15 are set (the right-most bit is bit 0). In the last example, all bits are set.

REAL type: Range [-6.80565E+38 to 6.80565E+38]. Use REAL numbers to represent objects with a fractional part or to represent numbers that are very large or very small, including money, the distance to the sun (93,000,000 miles), or measurements of temperature and time. Real numbers have 6 digits of precision; for example:

```
3.0E+8 98.6   0.0 2.3E-11
```

Turbo Modula-2 supports two types of double-precision numbers, LONGINT and LONGREAL. Double precision means the number is stored in twice the storage space as its single-precision counterpart, resulting in greater accuracy since the number is represented with more bits. Compare the ranges and number of significant digits in the single- and double-precision numbers.

LONGINT type: Range [-2147483648 to 2147483647]. Double-precision integers are used when a whole number greater than 65535 or less than -32768 is needed. For example, the size of a file may be expressed with a LONGINT as 100000L bytes. Other examples of long integers include the following:

```
0L 1234567889L -20000000L
```

LONGREAL type: Range [-3.5953862697246D+308 to 3.5953862697246D+308]. These are double-precision reals. Variables of this type have a precision of 14 digits; for example, multiplication is accurate to 14 decimal points. Use LONGREALs when numbers must be very large or require great accuracy.

Declarations

In general, Modula-2 programs have a declaration part and a program part. The declaration part must contain all constant, type, and data declarations necessary

to describe the data objects used in the **program** part. Data items must be defined before they are used.

Constant Declarations

Constants benefit programs in two ways: (1) They place a descriptive name where otherwise obscure numbers may be. (2) They simplify future modifications by limiting certain changes to those constant definitions that have a global effect.

A *constant declaration* associates an identifier with a value and a type. Constant declarations start with the reserved word **CONST** and are followed by any number of declarations terminated by semicolons. Each declaration statement contains an identifier followed by an equal sign, followed by a constant expression. A constant expression is an expression containing only constants.

In the following declaration, the identifier *Code* represents the value *A* and *Rate* represents the value 1.20:

```
CONST
  Code = "A";
  Rate = 1.20;
  Amount = 350.0 * Rate;
```

These values determine the type of the constant, which for *Code* is CHAR and for *Rate* is REAL. In the third expression, the identifier *Amount* represents the value obtained when the constant expression is evaluated. Notice that the identifier *Rate* must be a constant that has already been defined.

In some cases, the constant's type is not obvious; for example:

```
CONST
  N = 100;
```

Is the constant *N* of type INTEGER or of type CARDINAL? In this case, the answer is both. However, if *N* had been declared as

```
CONST
  N = 60000;
```

then its type would be only CARDINAL, since the value is out of the range for integers.

Type Declarations

Each data item has a type defining the possible values it can assume and the operations that can be performed on it. In addition, a item type implies its structure and how much storage the data item occupies. The type can be one of the predefined types (such as REAL or CARDINAL) or can be a user-defined type.

A *type declaration* begins with the reserved word **TYPE** and is followed by any number of declarations terminated by semicolons. Each declaration contains an identifier, followed by an equal sign, followed by a type identifier or statement. For example:

```
TYPE
  Ages = CARDINAL;
  Time = INTEGER;
```

Here two new types are introduced into the program. These types can now be used to declare data items with descriptive type names. This method has the same advantage as that of naming constants: By using a descriptive name for a defined type that is to be used several times in the variable declaration, the declaration becomes more readable and easier to modify. For instance, in the previous example, you can change all variables declared as type *Time* from INTEGER to REAL by simply changing the word INTEGER to REAL.

Variable Declarations

Two things are required to describe a variable: a unique identifier and a type. The value of the variable is left undefined so it may assume different values at runtime. Modula-2 has no provision for initialization of variables at load time; all variables must be initialized at runtime.

The *variable declaration* defines the variable's type and identifier and optionally the location in memory where the variable will exist. The declaration begins with the reserved word VAR and is followed by any number of variable declarations terminated by semicolons. Each declaration starts with an identifier list, optionally followed by an absolute location, then by a colon, and lastly by the type

of the variable. The absolute location is specified by a left square bracket ([ ), followed by an address, and a right square bracket ( ]). For example:

```
VAR
Count: CARDINAL;
SwitchOn: BOOLEAN;
Screen[0C000H] : ScreenType;
```

This example defines three variables: a CARDINAL called *Count*, a BOOLEAN called *SwitchOn*, and an absolute variable called *Screen* residing at memory location 0C000H (the location must be a constant). This latter variable (of type *ScreenType*. These three variables are completely defined and ready to use in a program (assuming the type *ScreenType* has been defined previously).

User-Defined Unstructured Types

We have already seen some predefined unstructured types; however, there are also user-defined unstructured types, which can be used to define variables that can hold one value at a time.

Users can define three unstructured types: *enumeration, subrange,* and *pointer.* In general, user-defined types may be declared directly in the variable declaration, or they may be defined as types in the type definition section.

Enumeration Types

Often, predefined types such as INTEGER or REAL are not sufficiently descriptive for certain applications. For instance, suppose you are trying to describe the four points of a compass. One solution is to decide that North is equal to 1, East 2, and so on. However, when you are reading and writing the code, you must always remember which number stands for the compass point you need. Also, the variables you are working with are declared as CARDINAL or INTEGER, which tells you nothing about how the variables should be used. Enumeration types (with a user- defined range) provide an elegant solution to this problem. First, you define a new type:

```
TYPE
  CompassPoints = (North, East, South, West);
```

Then, you may declare **variables of this type, like so:**

VAR
```
Direction: CompassPoints;
```

When you use the variable, you simply assign or test for **the values** you **have** defined for that type:

```
Direction := North;  (* Assign the variable the value North *)
Direction >= East;   (* Compare the variable to the value East *)
```

Enumeration types can also be declared directly in the variable section. Once an enumeration type is defined, the names of its values are also declared and cannot be used in another enumeration type. Here the type *Weather* can assume the values *Clear*, *Rain*, *Wind*, or *Snow*. For example,

VAR
```
Weather   :(Clear,Rain,Wind,Snow);
Water     :(Clear,Murky,Opaque);
Color     :(Violet,Orange,Green,Black);
```

The use of the value *Clear* in defining *Water* would produce a compiler error because the value *Clear* has already been defined as a value assigned to *Weather*. If the second *Clear* were changed to *Transparent*, then the compiler would accept the declaration.

Enumeration types have an implied order by the way they are declared. In the first example, *Clear* is the first value of the type *Weather*, with an ordinal value of 0. The value *Snow* is greater than *Clear*, *Rain*, or *Wind*. Variables of enumeration types can be compared to find their relative rank. Thus a statement like *Clear<Snow* is TRUE and *Rain>Wind* is FALSE.

Subrange Types

The *subrange type* includes certain variables that can be included in one of the previously mentioned types but which never takes values outside of a restricted range. For example, today's date is of the type INTEGER, but is always in the range of 1 to 31. We can state this by declaring it a subrange of the CARDINAL type. You may define subranges of INTEGER, CARDINAL, or enumeration types. Sample subrange types include the following:

```
TYPE
Today        = [1..31];      (* CARDINAL type with range 1 to 31 *)
Unusual      = [Rain..Snow]; (* Subrange of Weather
```

To specify a subrange of positive integers, you may prefix the subrange with the word INTEGER.

```
VAR
 Cents: INTEGER[0..100];
```

However, in general, the compiler will know which type is specified. If the (ange contains a negative bound, then it assumes INTEGER; otherwise, it assumes CARDINAL.

Pointer Types

Pointers point to another variable that can be of any type, including another pointer. Pointer operations include comparison and assignment, plus a special *dereferencing operator* that allows you to access the object pointed to by the pointer. Pointers are used mainly for pointing to other structured data types such as records or arrays (see "Structured Types" later in this chapter). These pointers are used to build dynamic structures such as linked lists or trees. Sample pointer types include the following:

```
TYPE
CharPtr        = POINTER TO CHAR;
IntPtr         = POINTER TO INTEGER;
CardPtr        = POINTER TO CARDINAL;
RealPtr        = POINTER TO REAL;
SetPtr         = POINTER TO BITSET;
( ngPtr        = POINTER TO LONGINT;
 boolPtr       = POINTER TO BOOLEAN;
LRealPtr       = POINTER TO LONGREAL;
PtrPtr         = POINTER TO POINTER TO CHAR;
Person         = POINTER TO PERSONRECORD
```

The value of a pointer is an address in memory. Thus, you may assign a pointer value a 16-bit address. Assuming the following variable declarations:

VAR
```
cp1,cp2: CharPtr;
```

you may make the following assignments:

```
cp1 := ADDRESS(39283);(* Assign an absolute address to cp1 *)
cp2 := cp1;           (* Make cp2 point to where cp1 points *)
```

Notice that the ADDRESS operator converts the CARDINAL number to a generic pointer type, which can be assigned to any pointer. (See ADDRESS in Chapter 12, "Turbo Modula-2 Reference Directory," or in the module *SYSTEM*.)

Dereferencing is used to access the item pointed to by a pointer. The operator used is the caret (^). To dereference a pointer, you simply append the caret to the end of the pointer's identifier. For example, to assign a character to the location pointed to by the previous declaration, you would use the following statement:

```
cp1 ^ := 'A';
```

Structured Types

A structured type is a composite of more than one data object. A structured type is analogous to a group of compartments, where each compartment may hold another structured or unstructured type.

There are two types of structures in Modula-2: *arrays* and *records*. Arrays are structured types that contain a predefined number of variables of the *same* type. In contrast, a record contains a fixed number of *different* types.

Both structured types can be accessed as a whole, or the parts of the structure can be accessed individually.

Array Types

Arrays have a fixed number of elements of an identical type. The type of each individual element is called the *base type* of the array. Each element of the array is accessed via an index. Arrays are declared as follows:

VAR
```
IntegerList: ARRAY[1..5] OF INTEGER;
```

IntegerList is an array of five integers. The integer elements can be accessed as *IntegerList*[*1*], *IntegerList*[2], and so forth, where the number in the square bracket is the index.

Arrays can be very descriptive when used in combination with enumeration and subrange types. For example, we could use arrays to record the hours of sunshine in each day of the week by using the following declarations:

```
TYPE
Hours = [1..24];
ㄱays = (Sunday, Monday, Tuesday, Wednesday, Thursday, Friday,
      Saturday);
DaysOfTheWeek = [Sunday..Saturday];
WeeklySunshine = ARRAY DaysOfTheWeek OF Hours;

VAR
Sunshine: WeeklySunshine;
```

Then to store the fact that *Monday* was gloomy and *Friday* was sunny, we would use the following:

```
Sunshine[Monday] := 1;
Sunshine[Friday] := 12;
```

Notice that each element can be accessed by an identifying index. The second item in the *Sunshine* array is accessed with the value *Monday*.

It is also possible to treat the array as a unit:

```
ᵀAR
ThisWeek,LastWeek: WeeklySunshine;
```

With this declaration, and assuming the array *LastWeek* already has some values, we can make this assignment:

```
ThisWeek := LastWeek;     (* More of the same weather *)
```

Multidimensional arrays (such as matrixes) are also possible. Given our previous definition of *Weather*, we can build a structure that will store the weather conditions for each hour of every day for a week. The declaration is as follows:

```
VAR
WeeksWeather: ARRAY Days OF ARRAY Hours OF Weather;
```

An alternative and identical declaration is

```
VAR
WeeksWeather: ARRAY Days,Hours OF Weather;
```

We can access the elements in several ways, two of which are shown here:

```
WeeksWeather[Sunday],[12] := Clear;
```

assigns *Clear* weather to the 12th hour on *Sunday*; and

```
WeeksWeather[Monday] := WeeksWeather[Sunday];
```

assigns *Sunday*'s weather to *Monday*.

Record Types

While an array must consist of elements of identical type, a record may cons:
of elements or *fields* of different types. Each field can be accessed by a uniq
name. Records are used to group different types of information into a single da
type. Use record var:ble to handle such information as the attributes of a perso
for example, his or her name, age, address, phone number, or favorite color. Tl
is easily done with the following record declaration:

```
TYPE
People =    RECORD
            Name    : ARRAY[1..30] OF CHAR;
            Age     : CARDINAL;
            Address: ARRAY[1..40] OF CHAR;
            Phone   : ARRAY[1..14] OF CHAR;
            PreferredColor: Color;
            END;
```

This is a record containing five fields: *Name*, *Age*, *Address*, *Phone* and *Prej
redColor*. Each is a different type: three different arrays (*Name*, *Address*, *
Phone*), a CARDINAL, and the user-defined type *Color*. Each field in the rec
can be accessed with the name of the variable, followed by a period and the na

of the field to be accessed. For example, if the variable *Friend* is defined as type *People*, like so:

```
VAR
Friend : People;
```

then we can access each field as follows:

```
Friend.Name            := 'Jon';
Friend.Age             := 23;
Friend.Address         := '234 Anywhere Drive';
Friend.Phone           := '333-3456';
Friend.PreferredColor  := Blue;
```

We can also nest records two different ways. We can include the nested record directly in the record definition.

```
TYPE
 People = RECORD
          Name : RECORD
                 First,
                 Middle,
                 Last: ARRAY[1..15] OF CHAR;
                 END:
          Age  : CARDINAL;
        END;
```

Or we can declare the type separately and nest it using the new type's name:

```
TYPE
 Names  = RECORD
          First,
          Middle,
          Last: ARRAY[1..15] OF CHAR;
          END;

 People = RECORD
          Name : Names;
          Age :   CARDINAL;
         END;
```

```
VAR
  Person: People;
BEGIN
  Person.Name.First[1]  = 'L'; (* first initial *)
  Person.Name.Middle[1] = 'J'; (* middle initial *)
  Person.Name.Last[1]   = 'G'; (* last initial *)
END
```

Notice that we can now access the initials of a name by selecting the Name field with each of the nested fields (*First*, *Middle*, *Last*), and then index into the first character of each array.

Variant Records

A record type can also have one or **more** dynamic variant parts. The **variant is** selected by the value of a *tag field*.

```
Employee = RECORD
              Name: ARRAY[1..20] OFCHAR;
              Age:  CARDINAL;
              CASE OwnHome: BOOLEAN OF
               TRUE: Payment     : CARDINAL;
                     OwnedSince   : Date
               |
               FALSE:Rent: CARDINAL
              END
           END;
```

In the preceding record, there are two fixed fields, *Name* and *Age*, and a varian part consisting of one or two fields, depending on the value of the tag fiel *OwnHome*. If *OwnHome* is TRUE, then the fields *Payment* and *OwnedSince* ar presumed to exist and the *Rent* field should not be accessed. If *OwnHome* i FALSE, then only the *Rent* field should be accessed.

Note: It is possible to access any of the fields regardless of the value of the ta field; it is up to the programmer to ensure that variant fields are only accesse when the tag field has the correct value.

Sometimes you may not want to specify a tag field. In this case, **the tag fie**

can be omitted, producing a record with fields that overlap one another. The colon and type must be present to indicate that no tag field was chosen.

```
TYPE
 HiLo = RECORD
          CASE : BOOLEAN OF
           TRUE: byte: ARRAY[0..1] OFCHAR;
           |
           FALSE: All: CARDINAL;
          END
         END;
VAR
 MyWord : HiLo;
```

The variable *MyWord* of type *HiLo* can be accessed as follows:

```
MyWord.byte[1]:=  A'; (* Assign character 'A' to the low byte *)
MyWord.All     := 65;  (* Assign the CARDINAL 65 to the entire word *)
```

Set Types

A set type defines a collection of related members. The members may be of any scalar type. Within a particular set, all members are of the same type and must have an ordinal value between 0 and 15. For example,

```
TYPE
Color  = (Red, Green, Blue);
Digits = SET OF [0..9];
Colors = SET OF Color;
```

The set *Digits* has members that are a subrange of type CARDINAL. The members of *Digits* have ordinal values ranging from 0 to 9. *Colors* is a set consisting of members of an enumerated type. The members have ordinal values ranging from 0 to 2.

The value of a set variable is the collection of members that it contains at any point in time. Operations on sets include assignment and comparison, as well as some special set operations. If we define set variables like this:

VAR
```
ColSet1,ColSet2: Colors;
```

we can make the following statements:

```
ColSet1 := Colors{Red,Blue};
ColSet2 := Colors{Red,Green};
```

Thus, a variable of type *Colors* may take as a value any combination of the three members; for example, {Red,Blue}, {Blue}, {Green,Blue}, and { }, which is the empty set.

Notice that the type of the set must prefix the constant set. If no prefix is present, the type of the set is assumed to be BITSET. The same operations applicable to BITSET are valid for user-defined sets (see BITSET in Chapter 12). Thus, we have the following:

```
ColSet1 + ColSet2       = Colors{Red,Green,Blue}
ColSet1 - ColSet2       = Colors{Blue}
ColSet1 * ColSet2       = Colors{Red}
ColSet1 / ColSet2       = Colors{Green,Blue}
ColSet1 - ColSet1       = Colors{ }
ColSet1 / Colors{Green} = Colors{Red,Green,Blue}
```

Procedure Types

Procedure types are an advanced language feature included here for completeness. Thus, you may wish first to read more about procedures and parameters (in Chapter 6), and then come back to procedure types.

Procedures can be assigned to a variable in order to be passed as arguments to other procedures. A procedure-type declaration specifies the number and type of its parameters and an optional function result. The following is a sample procedure type:

TYPE
```
Proc1 = PROCEDURE (CARDINAL, VAR INTEGER);
```

This is a procedure type with two arguments. The first argument is a valu

parameter of type CARDINAL and the second is a variable parameter of type IN-TEGER. For example, suppose *P* is a procedure variable of type *Proc1*; declared as

```
VAR
 P: Proc1;
```

The procedure variable *P* can take two arguments: one a CARDINAL, the other an INTEGER variable, and only in that order. The variable *P* could then be passed as an argument to another procedure.

A procedure variable may only take on the value of a globally defined procedure (see "Procedure Declarations" in Chapter 6) with an identical parameter list. For example, the variable *P* may take the value of procedures declared as follows:

```
PROCEDURE Test(:CARDINAL; VAR i:INTEGER);
BEGIN
END Test;
```

or

```
PROCEDURE Fees(accountNo:CARDINAL; VAR Balance:INTEGER);
BEGIN
END Fees;
```

The assignment operation is the only operation allowed on procedure variables.

```
P := Test;
P := Fees;
```

Assuming that *Fees* is assigned to *P* and that *CurrentBalance* is declared as an INTEGER variable, the procedure *Fees* can be called as follows:

```
P(121,CurrentBalance);
```

The most useful aspect of procedure variables is passing them as parameters to algorithms accessing complicated data structures (such as tree traversal procedures). The benefit is that many different operations can be performed on each element of the data structure, but only one data- access routine need be written to do so.

Chapter 6
Statements

In the previous chapter we discussed how to define data with constant, variable, and type declarations. Now we'll provide you with the remaining item you'll need to write your program. the statement.

Statements provide a means to handle data and define the flow of control. Modula-2 program statements can be divided into four classes: *assignment, conditional, repetitive,* and *procedural* (which includes procedure calls and a special flow of control statements). Each statement type will be examined in detail, but first we'll talk about programs in general in Modula-2.

In Modula-2, programs are called modules. Every main program begins with the reserved word **MODULE**, followed by the name of the program and a semicolon. The module statement is followed by a series of declaration statements that include the data declarations discussed in the last chapter (namely constant, type, and variable declarations), and possibly external and procedure declarations (discussed in the following chapter and at the end of this chapter, respectively).

The following sample program outlines the general form of a Modula-2 program:

```
MODULE Skeleton;
(* External Declarations (Next Chapter) *)

CONST
  (* Constant Declarations (Last Chapter) *)
TYPE
  (* Type Declarations (Last Chapter) *)
VAR
  (* Variable Declarations (Last Chapter) *)

  (* Procedure Declarations (This Chapter) *)
```

```
BEGIN
 (* Program statements (This Chapter) *)
END Skeleton.
```

The program body follows the declaration section, and contains statements separated by semicolons. The statements are within a block enclosed by the reserved words **BEGIN** and **END**, followed by the name of the program and a period. Modula-2 statements can be divided into the following classes:

Assignment   Assignments are made to variables, resulting in a change in the value that the variable holds. The source of the assignment may be any expression that has the same resulting type as the destination. A *simple assignment* may look like this: $X := 4$.

Conditional   This class groups statements together so that they are executed only if a specified condition is TRUE; for example, »If it's raining, then....« This group includes the IF and CASE statements.

Repetitive   Looping statements group one or more statements that are to be executed a number of times. For example, a loop may »Switch a light on and off ten times.« This group includes the **FOR**, **WHILE**, **REPEAT**, and **LOOP** statements.

Procedural   These statements call and control subroutines. They include procedure calls and the **RETURN** statement, which affects the flow of control in procedures.

Assignment Statements

A simple assignment statement replaces the current value of a variable with the result of an expression. The expression may be as simple as a constant or another variable, or it may be a complicated mathematical expression containing constants, variables, and functions. For example:

```
MODULE Assignment;
CONST
 Pi = 3.14159;
```

```
VAR
 X,Y: CARDINAL;
 CircleArea,Radius,Z : REAL;

BEGIN
 (* simple *)
 X := 2;
 Y := X + 1;
   Z           := 3.23;

 (* complex *)
 CircleArea := 2.0 * Pi * Radius;
 Z := (CircleArea*(45.2/29.3)+Z*(Z+2.0))/Pi;
END Assignment;
```

The result of the expression on the right must be assignment compatible with the variable on the left. In the first two examples, the expressions result in type CARDINAL; thus *X* and *Y* must be of type CARDINAL. The second set of expressions have REAL results; thus *CircleArea* and *Z* are of type REAL.

Note that := is the *assignment operator* and should be read as »becomes« (or gets), as in *I* becomes *I+1*, or *I* gets *I+2*.

WITH Statements

The WITH statement is used to alter the scope of identifiers, and is often used with assignment statements (though it is not itself an assignment statement). WITH is used with record variables, eliminating the need to name the record identifier for each field accessed within the WITH block. The WITH statement takes the following form:

```
WITH record identifier DO
 Statement sequence
END
```

This statement improves code readability when you are assigning values to fields of record variables, because the code reflects the immediate action of assignment to individual fields without the clutter of additional identifiers. For example:

```
MODULE WithStatement;
TYPE
 Months = (Jan,Feb,Mar,Apr,May,Jun,July,Aug,Sep,Oct,Nov,Dec);
 Date   = RECORD
              Day:[1..31];
              Month:Months;
Year:CARDINAL
            END;
 Person= RECORD
              Name       : ARRAY[0..30] OFCHAR;
              Age        : CARDINAL;
              Height     : REAL;
              BirthDay   : Date;
            END;
VAR
 Friend: ARRAY [1..2] OF Person;

BEGIN
 Friend[1].Name := 'Rodney';
 Friend[1].Age := 28;
 Friend[1].Height := 5.5;
 Friend[1].BirthDay.Day:= 3;
 Friend[1].BirthDay.Month:= Dec;
 Friend[1].BirthDay.Year:= 1958;
 WITHFriend[2] DO
  Name := 'Judith';
  Age := 32;
  Height := 5.4;
  WITH BirthDay DO
    Day:= 27;
    Month:= Oct;
    Year:= 1954;
  END
 END
ENDWithStatement.
```

The first record, *Friend [1]*, receives values by simple assignment. Note that for every field accessed the name of the record is present. The second record is accessed using the WITH statement. Notice that the structure of the code reflects the structure of the data. The change in scope is accentuated by the indentation used to code the WITH statement.

Conditional Statements

IF Statements

The IF statement executes a different *statement sequence* depending on the result of a Boolean expression. The IF statement has the form

```
IF BOOLEAN expression THEN
  Statement sequence
{ELSIF BOOLEAN expression THEN
  Statement sequence}
[ELSE
  Statement sequence]
END
```

The ELSIF (else if) and ELSE parts are optional. If the Boolean expression following the IF is TRUE, the first statement sequence is executed and control continues with the first statement after the END. If the first expression is FALSE and the ELSIF part is present, then its Boolean expression is evaluated. If it is TRUE, then its statement sequence is executed; otherwise, the next ELSIF is evaluated. There may be any number of ELSIF parts after the IF THEN, including none. If all expressions are FALSE, then the ELSE part, if present, is executed. For example:

```
MODULE IfThenElsifElse;

VAR
  SwitchOn,PowerOn: BOOLEAN;
  Status : (OK, Danger, Emergency);

BEGIN
  (* Statements that set Boolean flags *)
  IF SwitchOn AND PowerOn THEN
    Status := OK
  ELSIF NOT SwitchOn THEN
    Status := Danger
  ELSE
    Status := Emergency
  END;
    (* Other statements *)
END IfThenElsifElse.
```

After the Boolean flags have been set, the expression following the **IF** i:
evaluated. If it is TRUE, then *Status* gets *OK* and execution continues after th
END; otherwise, the **ELSIF** expression is evaluated. If it is TRUE, then *Statu.*
gets *Danger*. However, if the **ELSIF** expression is FALSE, the **ELSE** is execute(
and *Status* is set to *Emergency*. After either the **IF** part, the **ELSIF** part, or th
ELSE part is executed, control continues with the first statement after the **END**

CASE Statements

The CASE statement executes a different statement sequence according to th
value of an expression. The result of the expression is compared against value
in a constant list (a list of constants separated by commas) until a match is foun
or there are no constants left. A CASE statement may have several constant list:
each associated with a statement sequence and separated by vertical bars.

The elements of the constant lists must be of the same type as the expressio
following the **CASE**. The CASE statement takes the form

```
CASE expression OF
  Constant List : statement sequence
  |
  Constant List : statement sequence
         .
         .
         .
ELSE
  statement sequence
END
```

where a particular statement sequence is executed if the expression results
a value contained in the constant list. The optional ELSE part is executed if noi
of the constants match the expression. If none of the constants match the expre
sion and there is no ELSE part, the behavior of the code is dependent on the s(
ting of the TEST switch when the source was compiled (see »Compiler Optio
and Switches,« in Chapter 10). If TEST is ON, then the exception CaseSelectE
ror is generated. If TEST is OFF, then execution continues with the instructi(
after the END that corresponds to the CASE. For example:

```
MODULE Cases;
 VAR
  Temp   : (Freezing, Cold, Cool, Perfect, Warm, Hot, Searing);
  HeaterOn,
  AirConditionerON : BOOLEAN;
 BEGIN
  (* Statements which set Temp *)
  CASE Temp OF
  Freezing..Cool   : HeaterOn := TRUE
  |
   Perfect, Warm, Hot : HeaterOn := FALSE
  ELSE(* must be Searing *)
   HeaterOn := FALSE;
   AirConditionerON := TRUE;
  END
 END Cases.
```

This example executes one of three statement sequences depending on the value of *Temp*. If *Temp* has a value between *Freezing* and *Cool*, then the heater is turned on. If the value is between *Perfect* and *Hot*, then the heater is turned off. However, *Temp* may take on a value that is in neither constant list. In that case, the ELSE part of the CASE statement is executed and the heater is turned off and the air conditioner is turned on.

Note: The CASE statement is for situations where the lists of constants have adjacent ordinal values. When conditional execution is dependent on nonadjacent values, the IF THEN ELSIF statement should be used; otherwise, nonadjacent values used in CASE statements will generate a relatively large amount of code.

Repetitive Statements

FOR Statements

The FOR statement encloses a sequence of statements that is repeated a fixed number of times. The FOR statement takes the form

```
FORVar:= First TO Last BY Step DO
 statement sequence
END
```

where *Var* is a variable called the *control variable*, *First* is the initial value of
Last is the final value of *Var*, and *Step* is the increment added to *Var* for each ite
tion. *First*, *Last*, and *Var* can be any ordinal type, but they must all be of the sa
type.

The statement sequence is repeatedly executed and the control variable
cremented until the control variable equals or exceeds the value of *Last*. For exa
ple:

```
MODULE ForLoop;
CONST
 Increment = 0.1;
VAR
 I:CARDINAL;
 Result:REAL;
BEGIN
 FOR I:= 0 TO 10 BY 2 DO
  Result:= Result + Increment
 END
END ForLoop.
```

This example repeats the statement sequence six times, with *I* equal to
values 0, 2, 4, 6, 8, and 10. If the optional step value is omitted, then it is assur
to be 1. The step can also be negative; for example:

```
MODULE ForLoopCountDown
VAR
 Time,Add:INTEGER;
BEGIN
 FORTime:= 3 TO -3 BY -1 DO
  Add:= 2 * Time
 END
END ForLoopCountDown.
```

Note that the control variable **can take part in the statement sequence** but car
be altered by it.

WHILE Statements

The WHILE statement repeats a statement sequence until a Boolean expression yields a FALSE result. The WHILE statement takes the form

```
WHILE Boolean expression DO
 statement sequence
END
```

where the statement sequence is repeatedly executed when the Boolean expression is TRUE. For example:

```
MODULE WhileLoop;
 VAR
 Control:CARDINAL;
BEGIN
 Control:= 4;
 WHILE Control>0 DO
  Control:= Control - 1
 END
END WhileLoop.
```

This example repeats the statement sequence until *Control* is equal to 0.

Note that the Boolean expression is evaluated before the statement sequence; therefore, it is possible that the statement sequence will not be executed. In the preceding example, this would be true if *Control* was set to 0 instead of 4. Also note that the values that make up the control expression must be affected by some part of the loop, otherwise the loop will never end.

REPEAT Statements

The REPEAT statement replicates a statement sequence until a Boolean expression yields TRUE. The REPEAT statement differs from the WHILE statement in that the expression is evaluated *after* the statement sequence; therefore, the statement sequence is always executed at least once. The REPEAT statement has the form

```
REPEAT
 statement sequence
UNTIL Boolean expression
```

where the statement sequence is repeatedly executed until the expression is TRUE. For example:

```
MODULE RepeatLoop;
VAR
 N: REAL;
BEGIN
 N := 1.0;
 REPEAT
  N := 2.0 * N;
 UNTILN > 100.0;
END RepeatLoop.
```

This loop repeats until the value of *N* is greater than 100.

LOOP Statements

The **LOOP** statement repeats a statement sequence until terminated by an **EXIT** statement. The **LOOP** statement has the form

```
LOOP
 statement sequence
END
```

This statement is generally used when the termination condition can only be determined in the middle of the loop. Since Modula-2 has no **GOTO** statement, the **EXIT** statement is used to terminate the loop and continue control at the statement after the **END**.

EXIT Statements

The **EXIT** statement specifies termination of a **LOOP** statement. A common example of this occurs in sequential processing, where a data object is obtained and then processed. In this instance, a problem can occur if the program can only test if the object was obtained successfully *after* the statement to obtain it. Thus, there must be a test directly after the *Obtain* statement to see if it was successful. If

it was not, no further processing can be done and control must be passed around the processing statements and go to the statement after the loop. The following example simulates this situation with arrays:

```
MODULE LoopLoop;
TYPE
 DataObject = RECORD
                   Name : ARRAY[0..30] OFCHAR;
                   Age  : CARDINAL
                 END;
VAR
 Record: ARRAY[1..10] OFDataObject;
 WorkRecord : DataObject;
BEGIN
 I := 0;
 Record[10].Name := 'LastRecord';
 LOOP
  I := I + 1;
  WorkRecord := Record[I];
  IFWorkRecord.Name = 'LastRecord' THEN EXIT END;
  (* statements to process the record *)
 END
END LoopLoop.
```

This example presents a more common problem with sequential file processing: The flag in the *Name* field represents the end-of-file condition; thus, the records are continually read and processed until the end-of-file condition is TRUE. At this point the EXIT statement is executed and control continued after the loop's END.

More than one EXIT can be present in a loop. However, if there are too many .XIT statements, it is difficult to understand the meaning of the loop and to determine when and where it will terminate.

Procedural Statements

Procedures are named subroutines and subroutines are pieces of code declared elsewhere and invoked by name. In the following, we will look at how a procedure is called and how it is defined.

Procedure Calls

A *procedure call* is an identifier, optionally followed by a parameter list. The identifier represents a sequence of statements defined elsewhere. When a procedure call is encountered in a statement sequence, the flow of control is transferred to the statements that that identifier represents. A procedure call may have parameters specified by listing variables or constants, which are separated by commas and enclosed in parentheses. These are called *actual parameters*. The following example shows three procedure calls in the body of the module:

```
MODULE ProcedureCall;
 (* Statements that define the procedure Read *)
VAR
 Number,
Count          : CARDINAL;
 First         : INTEGER;
BEGIN
 Read(Number); Read(Count); INC(First);
END ProcedureCall.
```

The first *Read* statement has one actual parameter, *Number*. This parameter can be different each time the procedure is invoked, as with the second procedure call of *Read*, where *Count* is the actual parameter.

Note that there must be a declaration statement defining *Read*. However, this is not so with the third procedure call in the example, (*INC(First)*;). This is because it is a standard procedure, predefined in Modula-2 and callable from anywhere without explicit declarations. (There is a list of standard procedures at the end of this chapter.)

Procedure Declarations

Procedure declarations allow the programmer to define a statement sequence once and use it many times. Procedures allow a series of actions to be abstracted into a single meaningful name. For example, it could take hundreds of lines of code to implement the following procedure calls:

```
ReadChar;
AccessFile;
WriteScreen;
```

However, these three statements abstract the lines of code into names that describe the action the code performs. (This lends itself to the concept of information hiding: The details of each procedure are not relevant to the immediate code; they are hidden in the declarations of each procedure.)

As mentioned at the beginning of this chapter, the declaration section of a module may contain procedure declarations. Each procedure declaration starts with the reserved word **PROCEDURE** and is followed by the procedure's identifier name, an optional parameter list, an optional function result type, and a semicolon. Next is a declaration section that contains constant, type, variable, and possibly additional procedure declarations. The following example shows a skeleton procedure declaration:

```
PROCEDURE Skeleton((* parameter declarations *));
CONST
 (* Constant Declarations *)
TYPE
 (* Type Declarations *)
VAR
 (* Variable Declarations *)

 (* Procedure Declarations *)

BEGIN
 (* Program Statements *)
END Skeleton;
```

Notice the similarities between this procedure skeleton and the module skeleton at the beginning of this chapter. Both have a declaration section and a body, which is convenient when you need to convert a stand- alone program to a subroutine. The following example shows how procedures are declared and how they can be called repeatedly:

```
MODULE ProcedureDeclaration;
VAR
 x: CARDINAL;

PROCEDURE ExampleProcedure;
BEGIN
 x := x + 1;
```

END ExampleProcedure;

BEGIN
  x := 0;
  ExampleProcedure;
  ExampleProcedure;
  ExampleProcedure;
END ProcedureDeclaration.

The module *ProcedureDeclaration* has defined one procedure called *ExampleProcedure*. The body of the module calls it three times, incrementing *x* by 1 each time the procedure is invoked.

Parameters

Procedure declarations may have parameters that are passed to the procedure in a procedure call. The parameters are specified in the procedure statement after the procedure identifier, and enclosed in parentheses. This is called a *parameter declaration*.

The parameter declaration defines the names and types of each parameter, which are its *formal parameters*. For example:

PROCEDURE foo(a,b,c: INTEGER; d: CARDINAL; e: REAL);

Notice that parameters of the same type may be grouped together, as in variable declarations. Each group of parameters is separated by semicolons. In addition to name and type, the declaration determines the kind of parameter that is being passed.

There are two kinds of parameters: *variable parameters* and *value parameters*. Variable parameters link actual and formal identifiers, and are identified by prefixing a group of parameters with the reserved word **VAR**. Value parameters pass a value as an initial condition to a formal identifier. Since **VAR** does not appear in the preceding declaration, all of the parameters are value parameters.

To explain the difference between variable and value parameters, let's look at the procedure call. When a procedure call passes a parameter to a subroutine, the program may or may not want the variable it passes to be changed. If it does not, the formal parameter should be declared as a value parameter. But if the call is

expecting the procedure to do some work on the variable and return the new value, then the formal parameter should be a variable parameter.

When a value parameter is passed, a copy of the value is made. Any reference to that parameter will affect only the copy and not the original; thus, constants as well as variables may be passed as value parameters. For example:

```
MODULE ValParameters;

PROCEDURE Power(VAR A:REAL;I:CARDINAL);
VAR
 Temp: REAL;
BEGIN
 Temp:= 1.0;
 WHILE I > 0 DO
  Temp:= Temp * A;
  I:= I-1
 END;
 A := Temp;
END Power;

VAR
 x : REAL;
 e : CARDINAL;
BEGIN
 x := 10.0; e := 2;
 Power(x,e);     (* Raise 10 to the 2nd power *)
 (* Now x = 100.0 *)
 Power(x,3);     (* Raise 100 to the 3rd power *)
 (* Now x = 1000000.0 *)
END ValParameters.
```

(The reserved word VAR in the procedure heading declares *A* to be a variable parameter; the cardinal *I* is a value parameter.)

In the first call, the value parameter is the variable *e* set to the value 2. Notice that the value of *I* is changed. This will not be reflected in the value of *e*; it will remain the same. In the second call, the constant 3 is passed as a value parameter. A copy of this value is made within the procedure, thus it may be treated as a variable within the procedure.

When a variable parameter is passed, the address of the variable is passed. Thus, any reference to that parameter is to the original copy. For this reason, constants cannot be passed as variable parameters.

The following example shows the use of variable **parameters:**

```
MODULE VarParameters;

PROCEDURE Swap(VAR I,J: INTEGER);
VAR
 Temp: INTEGER;
BEGIN
 Temp:= I;
 I:= J;
 J:= Temp;
END Swap;

VAR
 a,b,c: INTEGER;
BEGIN
 a := 1; b := -1; c := 0;
 Swap(a,b);        (* Now a = -1 and b = 1 *)
 Swap(c,a);        (* Now c = -1 and a = 0 *)
END VarParameters.
```

The integers *I* and *J* are formal parameters. The formal parameters are used inside a procedure in place of the external actual parameters (*a*, *b*, *c*) in the example.

In the first call to *Swap*, *a* and *b* are aliases for *I* and *J*, respectively, which allows the procedure to change the values that *a* and *b* contain. The procedure interchanges the values of *I* and *J*, as well as the values of the external variables *a* and *b*. The second call then interchanges *c* and *a*, whereby *c* ends up with *b*'s original value, *a* gets the value of *c*, and *b* holds the original value of *a*.

Open Array Parameters
Modula-2 has a special feature that allows an array of unspecified size to be passed as a parameter to procedures. These are called *open array parameters*. They are specified in the parameter declaration list in the following form:

ARRAY OF T

where *T* is the type of the item in the array. The lower bound of the array is always 0, and the upper bound is determined by a call to the standard procedure *HIGH*. *HIGH* returns the highest legal index value based on the declaration of the array that was passed, but which has been adjusted so that the lower bound is zero. For example, consider this module:

```
MODULE OpenArrays;
TYPE
Employee = RECORD
               Name  : ARRAY[0..20] OF CHAR;
               Age   : CARDINAL;
               Wage  : REAL;
           END;
VAR
 AllEmployees      : ARRAY[1..20] OF Employee;
 CurrentEmployees  : ARRAY[0..9] OF Employee;

PROCEDURE PrintEmployeeRecords(VAR e: ARRAY OF Employee);
VAR
 I: CARDINAL;
BEGIN
 FOR I := 0 TO HIGH(e) DO
  WITH e[I] DO
   WRITELN(Name,Age:10,Wage:5:2);
  END
 END
END PrintEmployeeRecords;

VAR
 I: CARDINAL;
BEGIN
 FOR I := 1 TO 20 DO
  WITH AllEmployees[I] DO
   Name := 'Hilary';
   Age := 11;
   Wage := 3.35;
  END
 END;
```

```
FOR I := 0 TO 9 DO
  WITH CurrentEmployees[I] DO
    Name := 'Nate';
    Age := 2;
    Wage := 3.35;
  END
END;
PrintEmployeeRecords(AllEmployees);
PrintEmployeeRecords(CurrentEmployees);
END OpenArrays.
```

The procedure *PrintEmployeeRecords* will accept and correctly print the two Employee arrays even though they have different bounds. Note that the function *HIGH* gives the highest legally accessible item in the array.

Although *HIGH* provides the largest index, you may still need another way to determine how full the array is. For instance, it is often useful to define procedures that have open arrays of characters, because you need to pass strings of differing length to the same algorithm. However, the character arrays may not be completely filled when passed to the open array; thus, *HIGH* points to the end of the array rather than the end of the string. Modula-2 appends a null character (0C) if a constant string is passed to a procedure with an open array. Thus, the procedure can determine the actual length by scanning for the null character.

With open arrays of other types that may not be completely filled, you must provide some mechanism for the procedure to determine the last valid item in the array. This may be a flag record or an additional parameter that gives the count.

There is an even more powerful version of an open array that allows you to pass any type to a procedure; for more details, refer to Chapter 8. Turbo Modula-2 allows multidimensional open-array parameters, specified as

M: ARRAY OF ARRAY OF T

The high bound of the first dimension is *HIGH(M)* and the high bound of the second dimension is *HIGH(M[0])*. Thus you may find the high bound of the *n*th dimension by passing *HIGH* the array with *n-1* subscripts.

Function Procedures

As mentioned earlier, a procedure may return a result. This is called a *function procedure*. A call to a function procedure is either an expression or part of an expression. The types of values they return must be defined in the procedure statement, as follows:

```
PROCEDURE Foo( ): INTEGER;
```

This procedure heading defines a function that returns a value of type IN-TEGER. Note that even though there are no parameters, the empty parameter list must be present. This is true for both formal and actual parameter lists. For example, if the preceding function is used to assign a value to an integer variable, then it will look like this:

```
IntVar := Foo( );
```

With the empty parameter list, it is clear whether an identifier occurring in an expression is a variable or a function procedure. This is not a problem with normal parameterless procedures, because they cannot occur in expressions.

As an example, we will change the *Power* procedure we used earlier to a function:

```
MODULE FunctionProcedure;

PROCEDURE Power(A:REAL;I:CARDINAL):REAL;
VAR
 Temp: REAL;
BEGIN
(  np:= 1.0;
 WHILE I > 0 DO
  Temp:= Temp * A;
  I:= I-1;
 END;
 RETURN Temp;
END Power;
```

```
VAR
 x: REAL;
 e: CARDINAL;
BEGIN
 x := 10.0; e := 2;
 x := Power(x,e);   (* Raise 10 to the 2nd power *)
 (* Now x = 100.0 *)
 x := Power(x,3);   (* Raise 100 to the 3rd power *)
 (* Now x = 1000000.0 *)
END FunctionProcedure.
```

The *FunctionProcedure* example has the same outcome as the procedure *Pow*
example, but the methods are different. The parameter declaration for *A* has bee
changed from a variable parameter to a value parameter. Thus, the actu
parameter is not changed by the body of the function, but by the assignment stat
ment in the body of the module. Another difference is that the function *Power* h
an explicit **RETURN** statement to define the result and terminate the procedur

In general, it is a good practice to have only value parameters in a function pr
cedure. When a function's variable parameter or global variable is changed with
the course of the function call, it is called a side effect.

RETURN Statements

The value returned by a function procedure is explicitly named by a **RETUR**
statement. This statement takes the form

```
RETURN value;
```

where *value* is an expression with the same resulting type as the function pr
cedure. Thus, in an earlier example, the statement

```
RETURN Temp;
```

specifies the result of the function. The **RETURN** statement may also be us
in normal procedures with no statement after **RETURN**. In a normal procedu
the **RETURN** statement has the same effect as if the procedure had come to t
final **END** statement. This permits procedures to be exited at any point. As w

the **EXIT** statement previously described, too many **RETURN** statements in a procedure make it difficult to understand.

Examples of **RETURN** statements:

```
RETURN ;          (* normal procedure *)
RETURN 26;        (* function procedure returning an Integer *)
```

Nested Procedures

As shown in the skeleton procedure, the declaration section can also include additional procedure declarations called *nested procedures*, or *local procedures*. (The procedures described earlier are global procedures.) The following is an example of a nested procedure:

```
MODULE NestedProcedures;

PROCEDURE LevelOne;

  PROCEDURE LevelTwo;

    PROCEDURE LevelThree;
    BEGIN
    END LevelThree;

  BEGIN (* LevelTwo procedure body *)
  END LevelTwo;

  PROCEDURE AnotherLevelTwo;
  BEGIN
  END AnotherLevelTwo;

BEGIN (* LevelOne procedure body *)
END LevelOne;

BEGIN (* Module body *)
END NestedProcedures.
```

This example defines four procedures: one is global (procedure *LevelOne*), and the remaining three are nested. Two are considered local to the procedure

LevelOne; they are *LevelTwo* and *AnotherLevelTwo*. The procedure *LevelTwc*
a local procedure called *LevelThree*. At certain points in this program, only
tain procedure identifiers are available to be called. This is because proced
restrict the visibility of identifiers declared within them. This is referred to a:
scope of an identifier.

Scope of Visibility

Here we present the rules for the *scope of visibility* of an identifier for a
cedure. The rules for modules are slightly different and are presented in the
chapter.

   The procedure in which an identifier is declared defines the scope of visil
for that identifier. For procedures, the rules are as follows:

☐ An identifier exists while the declaring procedure is **active.**

☐ An identifier is visible within the declaring procedure after the identifi
   declared and it is also visible within nested procedures unless redeclare
   a nested procedure. The new identifier exists until the redeclaring proce
   is inactive, then the identifier reverts to the original declaration condit

☐ All identifiers that are visible outside of the procedure are also visible ii
   the procedure, unless redeclared in that procedure.

   For example consider the following module:

```
MODULE ProcedureScope;

PROCEDURE A;
VAR
 i,j: CARDINAL;

 PROCEDURE B;
 VAR
  j: REAL;
 BEGIN               (* i and j (the REAL) are visible here *)
  END B;
```

```
PROCEDURE C;
BEGIN              (* i and j (the CARDINAL) are visible here *)
END C;

BEGIN              (* i and j (the CARDINAL) are visible here *)
END A;

BEGIN
END ProcedureScope.
```

An identifier declared in *A* is visible in *A*, *B*, *C*, and all the procedures nested within *B* and *C*. On the other hand, an identifier declared in procedure *B* is visible only in *B* and any of *B*'s nested procedures, but not in *A* or *C*.

Notice that the identifier *j* is declared in both *A* and *B*. The *A* version of *j* (the CARDINAL) is visible throughout *A*, *C*, and the procedures nested in *C*. In procedure *B*, the *A* version is invisible, being replaced by the *B* version of *j* (the REAL). The *B* version is therefore visible in *B* and all the procedures nested in *B*.

FORWARD Statements

The original definition of Modula-2 specified that identifiers declared at the same level have the same scope. Thus a procedure declared textually at the top of a program could call a procedure that was declared later in the text. This was possible because the first Modula-2 compilers were multipass compilers (in most cases four passes) that could resolve an undeclared reference on a subsequent pass.

Turbo Modula-2 and Wirth's latest compilers are one-pass, recursive-decent compilers, which means there must be additional syntax to allow a single pass. This is done with the **FORWARD** statement. Its syntax and usage is similar to that of Pascal's. To use the **FORWARD** statement, simply copy the entire procedure heading, including the parameter list and the optional function result, to the location where it is needed. Then append the **FORWARD** statement and a semicolon. You need not change the procedure at all. The syntax is as follows:

```
PROCEDURE indentifier ( parameter list ) :result type ; FORWARD ;
```

This statement is required for algorithms involving mutual recursion, where one procedure calls another, which then calls the first. For example:

```
MODULE MutualRecursion;

PROCEDURE One; FORWARD ;
PROCEDURE Two;
BEGIN
 WRITELN('Procedure Two'); READLN;
 One;
END Two;

BEGIN
 WRITELN('Procedure Two'); READLN;
 One;
END Two;

PROCEDURE One;
BEGIN
 WRITELN('Procedure One'); READLN;
 Two;
END One;

BEGIN
 One;
END MutualRecursion.
```

Thus, the **FORWARD** statement is a way of extending the scope of a pr cedure's identifier, making it available over a larger program area.

Standard Procedures

Modula-2 includes several predefined procedures that are visible anywhere with a program; in other words, the identifiers do not require importing into a modu. (To see examples of these procedures, turn to the respective entries in Chapter 1: The following is a list of all standard procedures in Modula-2:

```
ABS(X)     Absolute value of variable X
CAP(Ch)    Uppercase of letters Ch
CARD(X)    Conversion of X to CARDINAL
CHR(X)     The character with ordinal number X
DEC(X)     Replace X by its predecessor
```

DEC(X,N)	Replace X by its Nth predecessor
EXCL(S,I)	Exclude element I from set S
FLOAT(X)	Conversion of X to REAL
HALT	Halt program execution
HIGH(A)	Upper index of array A
INC(X)	Replace X by its successor (X := X + 1)
INC(X,N)	Replace X by its Nth successor (X := X + N)
INCL(S,I)	Include element I in set S
FLOAT(X)	Conversion of X to REAL
HALT	Halt program execution
HIGH(A)	Upper index of array A
INC(X)	Replace X by its successor (X := X + 1)
INC(X,N)	Replace X by its Nth successor (X := X + N)
INCL(S,I)	Include element I in set S
INT(X)	Conversion to INTEGER
LONG(X)	Conversion to LONGINT
MAX(T)	Largest element of type T
MIN(T)	Smallest element of type T
ODD(X)	Returns Boolean TRUE if X is odd
ORD(X)	Ordinal number of X in its type
SIZE(X)	Returns the size in bytes of variable X
SIZE(T)	Returns the size in bytes of type T
TRUNC(X)	Truncate real X to CARDINAL
VAL(T,X)	Value with ordinal number X and type T

Figure 7-1: Hierarchy of Modula-2 Program
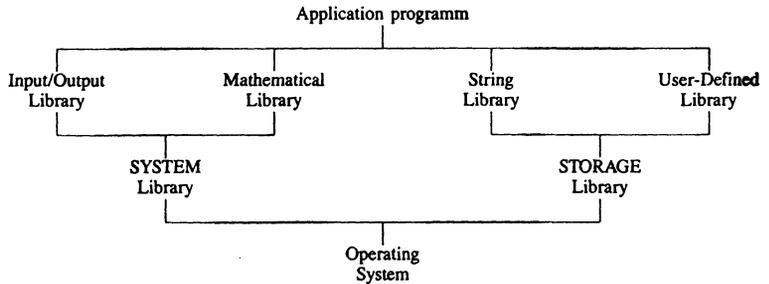
The application program is the main module, requiring the services of the *Ir put/Output*, *Mathematical*, *String*, and *User-defined Library* modules. Thes modules in turn may require the services of the so-called, »low- level« modulе (*SYSTEM* and *STORAGE* Libraries), which in turn require the services of th underlying operating system.

Each library module is a self-contained unit that is compiled separately. Use can build their own library modules and use them as if they were provided by tl Modula-2 system. When a main module uses library procedures, the facilities that library are linked to the main program. In general, this process is the san for both predefined and user-defined modules.

As in any hierarchy, the application program may bypass the intermediɑ modules to use the services of the low-level modules.

When a module requires the services of a library module, it must explicitly stɑ this as the first item in its declaration section. This is done with the IMPOI declaration, which can take the following forms:

FROM <module name> **IMPORT** <identifier list> ;

or

IMPORT <identifier list>;

The first form names a library module after the reserved word FROM, tl names the identifiers it requires from that module after the reserved word I PORT. The identifier list is terminated with a semicolon.

Chapter 7
Modules

In Modula-2, the module is a syntactical construct that serves to encapsulate certain parts of a program, letting the outside world see only identifiers that are explicitly made visible. Up till now, all the sample modules we have seen have been stand-alone programs, or main modules (which are analogous to entire Pascal programs). Typically, though, Modula-2 programs are made up of many modules, where each support module contains a small amount of code to do its part.

In the Turbo Modula-2 system, the smallest modules have less than 100 lines, the most complex ones nearly 1,500. (The upper limit also mirrors the memory limitations of the computer we're working on [Z80].) How large should a module be? There is no general rule. The task is to break the problem into natural pieces and to have little interconnection between the parts. The amount of interconnection is mirrored by the size of import and export lists; the smaller the lists, the better.

In this chapter, we'll examine the types of modules that can be used in Modula-2.

The Main Module

The main module defines a program. When a main module is running, it usually requires the services of library modules and the underlying operating system. Since these may be viewed as disjoint units (only minimal connection), we may look at a Modula-2 program as hierarchically structured, as shown in Figure 7-1.

The second form imports all identifiers made available by the module specified after the reserved word **IMPORT** and followed by a semicolon. However, unlike the first form, use of imported identifiers requires qualification. This means that any identifiers used from this module must be prefixed with the module name, as follows:

```
ModuleName.IdentiferName
```

Typically, this form is used when identifiers imported from different modules have the same name. Thus, the module name specifies which identifier is intended. The following example shows the use of the two forms of the **IMPORT** statement:

```
MODULE ImportLists;
FROM Terminal IMPORT WriteChar, WriteString, WriteLn, ReadChar;
IMPORT Texts;

VAR
 ch: CHAR;
 num: INTEGER;

BEGIN
 WriteString('Enter a character: ');
 ReadChar(ch); WriteLn;
 WriteString('The character you entered was " ');
 WriteChar(ch);
 WriteString(' " . '); WriteLn;

 Texts.WriteLn(Texts.output);
 Texts.WriteString(Texts.output,'Enter an integer and')
 Texts.WriteString(Texts.output,'press return: ');
 Texts.ReadInt(Texts.input,num);
 Texts.WriteString(Texts.output,'The number you entered was ');
 Texts.WriteInt(Texts.output,num,2);
 Texts.WriteLn(Texts.output);
END ImportLists.
```

The facilities of the two library modules *Terminal* and *Texts* are used in the program ImportLists. Individual identifiers are imported from the module *Terminal*: *WriteString*, *WriteChar*, *ReadChar*, and *WriteLn*. All facilities from the module

Texts are made available, though they must be qualified. The identifiers used from *Texts* are *output*, *input*, *WriteLn*, *WriteString*, *ReadInt*, and *WriteInt*. Notice that some procedures are available in both modules, which is why the module *Texts* was imported as a whole and then its identifiers qualified. (For more details on the operations of *Texts* and *Terminal*, refer to Chapter 11, »The Standard Library,« or the specific procedures in Chapter 12.)

Library Modules

Library modules have two advantages over stand-alone modules: *information hiding* and *module decoupling*.

In Modula-2, library modules hide the operation and details from the module user; information needed to use the library facilities is separate from the implementation details.

In the preceding example, we needed to know the types and number of parameters of the *WriteString* procedure in order to use it in the module *Terminal*. The declaration

```
PROCEDURE WriteString(s: ARRAY OF CHAR);
                              (* Write String to Screen *)
```

tells us that the procedure *WriteString* takes an array of characters and prints them to the screen. This information is contained in a *definition module* (explained in the next section). Thus by looking at the definition module, we can determine how to use its facilities.

In addition to information hiding, library modules allow the details of operations to be changed without affecting modules that depend on its definition. This is called module decoupling. This feature of Modula-2 aids in the development of large programs by minimizing recompilation after code changes. Thus, if a library module is made more efficient, users (or »clients«) of that module will only notice the increase in efficiency and will not need to recompile (unless producing a .COM file) or change their source code. A client need only recompile or change source code when the actual definition (and therefore meaning) of a module changes.

Definition and Implementation Modules

Modula-2 formalizes the separation of a module's definition and its implementation with these two modules: the definition module and the implementation module. Every library module consists of one definition module and one implementation module.

The definition module contains all the declarations of the library module that are to be made public, which may include constants, types, variables, and procedures. These declarations do not contain code, but instead contain the interface needed to use the procedures they export, along with comments that describe how ie exported identifiers should be used. Definition modules take the following form, beginning with the reserved words **DEFINITION MODULE**:

```
DEFINITION MODULE  <module name> ;

   < Import Sections >
   < Export Sections >

END <module name> .
```

where Import Sections declare any external identifiers needed to define exported objects and procedures. The Export Sections include the constant, type, variable, and procedure declarations that define identifiers exported by this library module.

Note: Definition modules may not contain an **EXPORT** list (see »Local Modules,« later in this chapter). All identifiers appearing in the definition module are automatically exported.

The following definition module is an example of a library module that provides random-number facilities:

```
DEFINITION MODULE RandomNumbers;
(* This module provides a reproducible set of random numbers. *)

PROCEDURE Randomize(NewSeed: Cardinal);
(* Randomize sets the internal seed to the specified value, thus
   allowing a random sequence to be reproduced when the same value is
   passed to Randomize.
```

```
*)

PROCEDURE Random( ): REAL;
(* The Random function returns a random real between 0 and 1 *)

END RandomNumbers.
```

This module provides information about the use of the random-number facilities its implementation provides. By reading this Modula-2 text, clients may incorporate random numbers into their programs without writing their own random-number generator or understanding how it works. The corresponding implementation module hides all the details from the client.

The implementation module holds the code that performs the actions stated in the definition module. Objects declared in the definition module are visible in the implementation module. Thus, constants, types, and variables should not be redeclared in the implementation module; doing so will cause a duplicate definition error during compilation. Also, the procedure heading declared in the definition module must be identical to the procedure heading in the implementation module or the compiler will flag the implementation's procedure in error as »Two different declarations of same procedure.«

Implementation modules must define all procedures mentioned in the definition module or the compiler will flag unresolved identifiers.

Implementation modules can be considered identical to main modules with **two** exceptions. The first is the form of declaration, as shown in the following:

```
IMPLEMENTATION MODULE <module name> ;
(* Import Declarations *)
(* Data Declarations *)
(* Procedure and Module Declarations *)

BEGIN
 (* Initialization Code *)
end <module name> .
```

Note that the module name **must be the same one used in the** definition module.

The second exception is the optional initialization code, which follows the reserved word **BEGIN**. This code is used primarily to set up variables before any procedure is called or any variable is accessed. Thus, you are guaranteed that this initialization code will be executed before the client modules use any function provided by the library module.

If the initialization code is not present, then the reserved word **BEGIN** is optional. Of course, the reserved word **END**, followed by the module name and a period, must be present.

The following module implements the previously defined module *RandomNumbers*:

```
IMPLEMENTATION MODULE RandomNumbers;
CONST
 multiplier = 100;
 modulus = 257;
VAR
 Seed: CARDINAL;

PROCEDURE Randomize(NewSeed: CARDINAL);
BEGIN
 Seed := NewSeed MOD modulus;
END Randomize;

PROCEDURE RANDOM( ): REAL;
BEGIN
 Seed := (Seed * multiplier) MOD modulus;
 RETURN FLOAT(Seed) / FLOAT(modulus);
END Random;
BEGIN
 Randomize(1986)
END RandomNumbers.
```

To complete the examples presented in this section, we can define a main module that imports the facilities of *RandomNumbers*:

```
MODULE PrintRandomNumbers;
FROM RandomNumbers IMPORT Randomize,Random;
FROM Texts IMPORT output, WriteReal, WriteLn;
```

```
VAR
 Count: CARDINAL;

BEGIN
 FOR Count := 1 TO 10 DO
  WriteReal(output,Random( ),5,3); WriteLn(output);
 END
END PrintRandomNumbers.
```

As mentioned earlier, most Modula-2 programs are made up of several modules. The preceding module depends on two library modules, each made up of two modules (a definition and an implementation module). We have defined the two modules required for random numbers, and thus have the source to both. However, the module *Texts* is a standard library module provided by Turbo Modula-2. This means the source code is not available, though the definition module is listed in the next chapter. So how does the program know the interface and code? This information is stored in compiled form; there is one file for the definition module and one for the implementation module.

Compiled Modules

When a definition module is compiled, the result is stored in a symbol file with the extension .SYM. This file is basically a symbol table containing the names and types of exported identifiers. When the compiler is resolving the import lis of a module, it looks in this file to obtain information about the imported iden tifiers. This information is then inserted into the working symbol table of the module being compiled.

Thus, the compiler can tell if an externally declared identifier is being used cor rectly, at least in a syntactic sense. This concept is referred to as type-checkin; across compilation units. To compile a module that imports identifiers, the com piler must have access to the identifier's corresponding .SYM file. For example if there is an import from a module named *Texts*, then there must be a file named Texts.SYM.

When an implementation module is compiled, its result is stored in a specia code file with the extension .MCD. This file contains the code that is linked i when a program is either loaded or linked.

For convenience, Turbo Modula-2 provides a library manager that groups th

.SYM and .MCD files of various library modules into one file with the extension .LIB. For example, all of the standard library modules provided with Turbo Modula-2 are in the library file called SYSLIB.LIB. (For more information on the library manager, refer to Chapter 10, »System Operations.«).

Opaque Export

Sometimes library modules must ensure that variables of an **exported type are on**ly manipulated by procedures of the library module itself.

Consider a module that exports a file type. One approach is to export the whole structure of this record, making all fields accessible to user programs. If, however, a program updates such descriptor records, chaos may ensue (in the file example, valuable information on disk may be destroyed). In such cases, we would like to hide the fields of the record.

To avoid such problems, Modula-2 offers *opaque export*. This means a name of a type is exported without giving its structure (the actual structure is given in the corresponding implementation module). Thus, user programs may declare variables of that type, but the only operations applied to them will be the provided library procedures. The following example may help clarify this:

```
DEFINITION MODULE Files;
TYPE
 FILE;

PROCEDURE Open(VAR f: FILE; filename: ARRAY OF CHAR);

(* Other declarations *)

END Files.

MODULE OpaqueExport;
FROM Files IMPORT FILE, Open;
```

```
VAR
 f : FILE;
BEGIN
 Open(f,"A:TESTDATA.DAT")
END OpaqueExport.
```

Since the definition module *Files* does not provide information about the type *FILE* the user program cannot apply any operations to it, except for those explicitly provided by *Files* (which in this case is Open).

When compiling the example *OpaqueExport*, the compiler must know the space requirements of the variables of opaque types, which in this case is *FILE*. However, it can only inspect the definition module. Therefore, to be able to find out the space needs of opaque-type variables, a restriction must be imposed on opaque export: Only pointer types can be exported in opaque mode.

Local Modules

The remaining module type in Modula-2 is a *local module*, one that is nested within either a main module or an implementation module. Local modules serve to hide details of some task or object from the surrounding environment. Since they are nested, they cannot be separately compiled. Additionally, they have special scope rules.

Local modules may be declared anywhere a procedure declaration is permitted. Thus, local modules may appear in the declaration section of either a module (main, implementation, or local) or a procedure. In general, modules and their contents are static, meaning they exist throughout the duration of the program that surrounds them. There is one exception: When a module is declared local to a procedure, then the module exists only while the enclosing procedure is active.

A local module declaration is similar to a main module and takes the following form:

```
MODULE <module name> ;
(* Import Declarations *)
(* Export Declarations *)
(* Data Declarations *)
(* Procedure and Module Declarations *)
```

BEGIN
  (* Initialization Code *)
 END <module name> ;

Its export declaration is what makes it different from other modules. These declarations cause the listed identifiers to become visible in the surrounding environment. Both import and export declarations cause the scope of identifiers to be altered in both the external and internal environment of the local module.

Scope and Local Modules

Unlike procedure walls, local module walls are not only opaque from the outside but also from the inside. Identifiers that are to be used within a local module must be imported (via an import declaration), and identifiers declared inside a local module that must be visible outside must be exported (via an export declaration). The rules for importing and exporting are completely symmetrical:

☐ An identifier exists while the declaring module is active. The declaring module is active while its surrounding environment is active.

☐ An identifier is only visible in a local module if declared by the module or imported from the surrounding scope.

☐ An identifier declared in a local module is **only visible within the module**, unless explicity exported.

The form of import declaration for local modules is identical to that for other kinds of modules. The major difference is that any identifier in the surrounding scope may be imported; thus, you may find import lists grouping many different identifiers from many different sources within the same list.

The export declaration that makes identifiers visible outside **the local module** has the following form:

 EXPORT <identifier list> ;

where the identifier list may contain constants, types, variables, and procedures. The identifier list is prefixed with the reserved word **EXPORT** and terminated with a semicolon.

An export declaration introduces an identifier to the next highest scope. For example, an export declaration in a local module would introduce an identifier to the surrounding environment. And an import declaration would introduce an identifier into the local module from the next highest scope.

This modular separation hides many details from the programmer and protects the abstraction of the data structure, which is defined in the local module.

The following stack example demonstrates this:

```
MODULE LocalModules;
FROM Terminal IMPORT WriteString, WriteLn;
TYPE
 StackElement = CARDINAL;

 MODULE AbstractStack;
 IMPORT StackElement, WriteString, WriteLn;
 EXPORT Push, Pop;

 CONST
  StackSize = 100;
 VAR
 SP : [0..StackSize];    (* The stack pointer *)
  Stack : ARRAY [0..StackSize-1] OF StackElement;

 PROCEDURE Push(item : StackElement);
 BEGIN
 IF SP < StackSize THEN
  Stack[SP] := item;
  INC(SP);
 ELSE
  WriteString('ERROR: Stack overflow.'); WriteLn; HALT;
 END
END Push;

 PROCEDURE Pop(VAR item: StackElement);
 BEGIN
  IF SP > 0 THEN
  DEC(SP);
  item := Stack[SP]
```

```
  ELSE
   WriteString('ERROR: Stack underflow'); WriteLn; HALT;
  END
 END Pop;

 BEGIN (* initialization *)
  SP := 0;
 END AbstractStack;

 VAR
  i: CARDINAL;
(  IN (* main program for LocalModules *)
  Push(2);
  Pop(i);
  IF i = 2 THEN WriteString('The stack worked!') END
 END LocalModules.
```

Since the stack pointer variable *SP* is not exported, it is not visible outside the module. Note that all objects declared outside of the module must be imported if they are used. The only exception to this rule are standard identifiers, such as INTEGER, BOOLEAN, *ORD*, and so forth. Notice the use of the standard procedure *HALT* to stop execution when a stack error occurs.

There is a further facility to restrict the number of visible identifiers: *qualified export*. If we had written

```
MODULE AbstractStack;
IMPORT ...
EXPORT QUALIFIED Push, Pop;
```

(' :n the procedures *Push* and *pop* would have to be denoted by Abstract-Stack.Push and AbstractStack.Pop, respectively, in the surrounding scope.

Since it is sometimes difficult to avoid name clashes of different modules' identifiers, qualified export allows you to clarify ambiguous names. Note that identifiers exported from library modules are always exported in a qualified mode, but the qualification is usually overridden with the FROM statement.

Local modules may also overrride the qualified statement by importing with the FROM clause. This allows local modules to share data without cluttering the next

highest scope with unnecessary identifier names. The following example demonstrates **QUALIFIED** exports:

```
MODULE t;

  MODULE one;
  EXPORT QUALIFIED a;

    MODULE two;
    EXPORT QUALIFIED a;
    VAR
     a: CARDINAL;
    BEGIN
     a := 4;
    END two;

    MODULE three;
    FROM two IMPORT a;
    BEGIN
     a := 5;
    END three;

  BEGIN
   WRITELN('one',two.a);
  END one;

BEGIN
 WRITELN(one.a);
END t.
```

Note: In the main body of module one, the identifier *a* must be qualified; however, it is not qualified in the body of module three.

Chapter 8
Low-Level Facilities

Modula-2 is a strongly typed language, an aspect that greatly contributes to its programming safety, yet one that is often too restrictive for system programming. Low-level facilities are the avenue to system programming in Modula-2. They include type-transfer functions, special types of the pseudomodule *SYSTEM*, ab- 'ute addresses, and coroutines and interrupts. It is helpful to restrict their use to small sections of code grouped into one module, so that when the program is moved to another system only the one module need be changed.

Before we begin discussing low-level facilities, though, we should consider some of the decisions that were made in implementing Turbo Modula-2. These decisions have a direct impact on certain low-level facilities, such as type-transfer functions and **SYSTEM** types. The general philosophy at Borland is small and fast; thus we have restrictions like word alignment of variables and benefits such as register variables.

Register variables are used whenever possible. Programmers can hand-optimize procedures by declaring important variables in the first four words of local storage. Only simple variables of unstructured types are chosen as register variables. If there are not four words of storage declared in a procedure, then the procedure's parameters are placed in registers. Consider the following procedure:

```
PROCEDURE RegVars(x,y: CARDINAL);
VAR
  i,j,amount: CARDINAL;
  .GIN
END RegVars;
```

In this procedure, three words of local storage (*i,j,amount*) and one word of parameters (*x*) would be allocated to the registers as follows:

x	sits in BC
amount	sits in DE
j	sits in BC'
i	sits in DE'

A consequence of using register variables is that while a variable sits in a register that variable will have no address. We impose the following restriction: *Simple variables of unstructured type are not accepted by ADR.*

Of course, these variables do not have to be kept in registers during the entire procedure. For example, they may be placed in memory if a procedure is called. However, if a procedure has no procedure calls, then the register variables will not be moved to memory, and the procedure can be very efficient and fast.

Word alignment provides both advantages and disadvantages. Word alignment means placing all variables on word boundaries, and the variables that require less than a word of storage are allocated a full word anyway. Thus, a character that needs only 1 byte of storage is allocated one word.

The reason for allocating variables in a word-aligned fashion is that it makes M-code and the M-code interpreter highly efficient and small. Since the M-code interpreter is part of the runtime system (always present), it makes sense to use M-code for non-time-critical operations. This is because M-code takes up approximately one-third the space of native (machine) code, allowing more code to be fit into Z80 memory space. Turbo Modula-2 allows a single program to be made up of both M-code and native code modules so that very large and efficient programs can be written.

The choice to use an efficient M-code interpreter, however, does impose certain restrictions on accessing memory. To keep the interpreter efficient, it may only access entire words at a time. Thus variables are word aligned, which leads to the following restrictions:

☐ Byte-sized array elements cannot be substituted for **VAR** parameters.

☐ Byte-sized array elements cannot be arguments for *INC* or *DEC*.

Notice these restrictions are only for byte-sized array elements. This is because byte-sized array elements are not word aligned, they are packed. This is apparent when contrasting arrays and records. A record variable that contains two character fields will always occupy two words of storage. An array that is declared to contain two character elements is always packed into 2 bytes (one word).

When considering certain low-level facilities, keep in mind that variables are usually word aligned. Word alignment impacts transfer functions that involve characters and bytes and the **SYSTEM** type *BYTE*.

Type-Transfer Functions

Modula-2 offers facilities to relax strict type-checking, forsaking some portability for the ability to perform system programming.

For this purpose, type identifiers are used as function names. The type- transfer function allows programs to explicitly override the type-checking of the compiler. Type-transfer functions are not meant to perform any computation or manipulation. They merely change the interpretation of the bit pattern contained in their argument.

(.ince the machine representations of types are entirely implementation dependent, so are the correspondences between representations. Modula-2 does not, for example, define which bit pattern corresponds to the integer value 15. Therefore, the use of type-transfer functions makes a program nonportable.

Type-transfer functions can only be applied to simple types. Moreover, type transfers are only allowed between types that occupy the same amount of storage. Since different systems may have types of different sizes, the applicability of type-transfer functions depends upon the implementation.

In Turbo Modula-2 implementation on Z80 computers, simple types have three sizes: REAL and LONGINT occupy 4 bytes, LONGREAL occupies 8 bytes, and the rest (including pointers, set, characters, and subranges) occupy 2 bytes. Because of the nature of the M-code interpreter, all data objects in Turbo Modula-2 are word aligned. This means that a single character declaration allocates one word, not 1 byte. Further, a record structure consisting of two characters occupies 4 bytes. Only arrays are packed so that characters are in an adjacent byte. Thus, an **ARRAY** [0..1] OF CHAR occupies only one word.

('pe-transfer functions are illustrated in the following module:

```
MODULE TypeTransfer;
VAR
ch : CHAR;
 c : CARDINAL;
 i : INTEGER;
 b : BOOLEAN;
 s : BITSET;
```

```
r : REAL;
l : LONGINT;

BEGIN
 ch:= CHAR(c);
 s := BITSET(i)*BITSET(b);   (* bitwise AND operator *)
 r := REAL(l);
END TypeTransfer.
```

This module shows the use of type-transfer functions. The first line in the body allows the CARDINAL *c* to be assigned to the CHAR ch. The second line transfers two different types into type BITSET, performs a set operation, and then assigns the resulting set to the variable *s*. The last line transfers the LONGINT l to a REAL. No physical conversion takes place in any transfer; the bits of the variables remain the same, but are viewed in a different way.

Type Transfer and Type Conversion

There is a difference to note between type-transfer functions and type conversions. While conversions try to preserve meaning, changing the bit patterns if necessary, type-transfer functions leave the bit pattern unchanged but interpret it differently. For example, in the type conversion call

```
LONG(1.0) = 1L
```

the real value is truncated and converted to type LONGINT. But in the type-transfer call

```
LONGINT(1.0) = 1065353216
```

the bit pattern standing for the real value 1.0 is interpreted as a LONGINT number.

In addition, type-transfer functions perform no checking. The statement

```
b := BOOLEAN(3)
```

where *b* is declared as a BOOLEAN, will compile and execute without any error messages. Execution may, of course, lead to somewhat strange results.

In some cases it may appear as if there is no difference between a type- transfer

function and a type-conversion function. Consider the following code:

```
PROCEDURE TransferAndConversion(i: INTEGER);
VAR
 c : CARDINAL;
BEGIN
 c := CARDINAL(i); (* Type transfer *)
 c := CARD(i);     (* Type conversion *)
END TransferAndConversion;
```

As long as the value of *i* is positive, the two statements will have the same meaning and effect. However, if the value of *i* is -1, the first statement will execute fine, but the value that *c* receives may not be what you think (65535). And if *i* is -1, the second statement will cause a runtime error because the conversion is impossible using *CARD*--you cannot convert a negative number to a CARDINAL value.

If you intend to look at the same bits in a different way, then you need the low-level facilities of type-transfer functions.

Low-Level Types and the Pseudomodule System

Besides type-transfer functions, there are the special types *BYTE*, *WORD*, and *ADDRESS* exported by the pseudomodule *SYSTEM* (so-called because it does not really reside in a library module).

Type *WORD* is compatible with all types that occupy one machine word of storage. Since objects of type *WORD* are considered to have no specific interpretation, no operations (except assignment) may be applied to them. Type-transfer functions must be used to indicate the desired interpretation.

There is also the type *BYTE*, which is compatible with 8-bit types such as CHAR. Like *WORD*, the assignment operation is the only one that may be applied to variables of type *BYTE*. Note that whether a variable of type *BYTE* takes only 1 byte or 2 bytes of memory is dependent on the declaration used (see the beginning of this chapter).

The type *ADDRESS* is compatible with all pointer types and has the declaration

TYPE
ADDRESS = **POINTER TO** WORD

In contrast to type *WORD*, arithmetic operators may be applied to operand:
of type *ADDRESS*; such operands will behave like CARDINAL operands. Thi:
feature is especially useful for storage management algorithms (also called pointe
arithmetic). The resulting type from pointer arithmetic is *ADDRESS*.

All knowledge about the pseudomodule *SYSTEM* is built into the compiler
Although there is no actual implementation module that corresponds to the defini
tion module of *SYSTEM*, you may use *SYSTEM* as if it were defined as the defini
tion module presented in Chapter 11, »The Standard Library.«

Untyped Parameters

Modula-2 allows a procedure to declare a parameter that will accept any typ:
passed to it. This is called an *untyped parameter*. Untyped parameters have the
following type declaration form:

PROCEDURE Foo(chunk: **ARRAY OF WORD**);

Notice that the untyped parameter declaration is similar to an open arra
parameter declaration. The difference is that it uses the low-level type *WORD* ex
ported from the pseudomodule *SYSTEM*.

Untyped parameters are functionally different from open array parameters ii
that they accept any type of variable as an actual parameter, not just arrays of on·
type. You may pass either a single character or a large record structure to a pro
cedure with an untyped parameter. For example:

```
MODULE UntypedParameters;
FROM SYSTEM IMPORT WORD;

PROCEDURE HowBigIs(chunk: ARRAY OF WORD);
BEGIN
 WRITELN('Size is ',SIZE(chunk):3,' High is ',HIGH(chunk):3);
END HowBigIs;
```

```
VAR
 c : CARDINAL;
 r : REAL;
 s : ARRAY [0..20]OF CHAR;
 t : RECORD
       a,b,c,d: LONGREAL;
    END ;
BEGIN
WRITE('CARDINAL          : '); HowBigIs(c);
 WRITE('REAL             : '); HowBigIs(r);
 WRITE('String literal   : '); HowBigIs('This is a string');
 s := 'This is a string';
 WRITE('String variable  : '); HowBigIs(s);
 WRITE('Record variable  : '); HowBigIs(t);
END UntypedParameters.
```

The following are the results of this program:

```
CARDINAL        : Size is   2 High is   0
REAL            : Size is   4 High is   1
String literal  : Size is  16 High is   7
String variable : Size is  22 High is  10
Record variable : Size is  32 High is  15
```

The standard function *SIZE* gives you the number of bytes that were passed, and the standard function *HIGH* tells you the index of the last valid word in the open array.

Absolute Addresses

Modula-2 has a facility to explicitly specify the address of a variable and thus ?rride the space allocation scheme used by the Modula-2 system. This is especially useful for memory-mapped I/O devices. In the example that follows, a memory-mapped video screen is declared to reside at address 0C000H:

```
VAR
 screen[0C000H] : ARRAY [0..31],[0..127] OF CHAR;
```

The desired address is specified in brackets after the variable identifier; in every other respect, this is a normal variable declaration. The programmer must ensure

correct and consistent use of such variables. Declaring the previous scree
variable on a computer with a different screen would lead to chaos.

A useful application of this facility is to define an array that stretches acro:
all memory. Using the low-level types *WORD* and *BYTE*, we can define an arr:
from the start to finish of memory, like so:

```
VAR
 Mem [1]:RECORD
        CASE : BOOLEAN OF
         TRUE: b : ARRAY [1..65535] OF BYTE |
         FALSE: w: ARRAY [1..32767] OF WORD |
        END
          END ;
```

Memory locations can be accessed like this:

```
Mem.b[10] := 0C;
Mem.w[CARDINAL(ADR(SomeDataRecordOrArray))+SomeOffset] := 39201;
```

Note that you cannot define an absolute variable at location 0. However, the se
cond byte of a word defined at location 0FFFF resides at location 0.

Coroutines and Interrupts

The pseudomodule SYSTEM provides certain mechanisms to implement *co.
outines* and *interrupt handlers*, in a high-level manner. These facilities ai
machine-specific but are presumed to be in all implementations of Modula-2. Th
actual synchronization and scheduling of processes must be done by suppo
modules. As an example, the standard library module *Processes* offers one poss
ble implementation of coroutines; there are many other ways.

Coroutines

Coroutines can be created by calling the procedure *NEWPROCESS* exported t
SYSTEM. Switching between coroutines must be done explicitly by calling th
procedure *TRANSFER*. Both the source and destination coroutines have to t
identified in the *TRANSFER* statement.

Programs using coroutines can be thought of as consisting of several programs, each with their own program counters. Each program corresponds to a coroutine: One is always active, the others are sleeping or frozen. *TRANSFER* freezes the currently active coroutine and activates another one. Unlike procedures, when a coroutine is reactivated it will always continue processing from the point where it was frozen. In contrast, when a procedure is activated, processing always starts at the beginning.

Note that more than one coroutine may use the same procedure. However, each coroutine must have its own *PROCESS* variable and its own work space. This allows several processes to share the same code.

Interrupts

Usually, an interrupt is an unscheduled jump to some special code triggered by a hardware condition, although, software may also trigger interrupts. The code that receives control *after* an interrupt is called an *interrupt handler*. It services the interrupt and then returns control to the interrupted program.

In Modula-2, interrupts are considered coroutine transfers and interrupt handlers are coroutines. When an interrupt occurs, the currently executing code is suspended, and an unscheduled transfer to the coroutine waiting to service that interrupt takes place. When the interrupt handler is finished, control is returned to the suspended code by using a special transfer statement, *IOTRANSFER*, exported from the pseudomodule *SYSTEM*. This procedure not only acts as a coroutine transfer but also initializes (or re-initializes) the interrupt vector for the next interrupt.

Of course, the coroutine waiting for an interrupt must somehow notify the system which interrupt it wants. For this purpose, interrupt vectors are used. (The Z80 CPU uses this scheme when it is in interrupt mode 2.) Note that it is up to :e programmer to ensure that the processor is in the correct interrupt mode and that the interrupt jump table is in the correct location.

Note that the CP/M operating system is not reentrant, making it difficult to use interrupts. To guarantee proper functioning, be certain the operating system is not active when user interrupts are possible.

Chapter 9
Turbo Modula-2 Extensions

Turbo Modula-2 offers several extensions to Wirth's definition that are not covered by standard Modula-2, such as general-purpose *READ* and *WRITE* statements, string comparison and assignment, multidimensional open arrays, and exception handling. These extensions can be suppressed with the embedded compiler option .* −X- *), or from the Options menu. If this option is turned off, the compiler will flag all extensions with a warning message.

Input and Output Extensions

Although the various input/output modules provide procedures for reading and writing data to external devices, Turbo Modula-2 defines four new statements that allow a simplified approach to input/output: *READ, WRITE, READLN,* and *WRITELN.*

The four statements provide you with a quick way to do output. The following program shows some of the many possible uses for these extensions.

```
MODULE READWRITE;
VAR aNumber: CARDINAL;
    aCharacter: CHAR;

BEGIN
/RITELN('The READ and WRITE statements',' take any number ',
        'of arguments');
 WRITE('Enter a number and a character: ');
 READ(aNumber,aCharacter);
 WRITE('You may even mix types. The number is ',aNumber);
 WRITELN(' The character is ',aCharacter);
 WRITELN('Number may be formatted, as in Pascal ');
 WRITELN('The number 3 in a field of length of 4: " ',3:4, ' " ');
 WRITELN('A real number ',34.556:10:2);
END READWRITE.
```

READ and *WRITE* extensions have essentially the same function as the Pascal equivalents: They eliminate the need to use Modula-2's precise library pr cedures.

During compilation, the *READ* and *WRITE* statements are translated into ca to the appropriate input/output procedures from the library module *Texts*. Usir these statements replaces the need to import the specific procedure from the *Tex* module.

For a more thorough discussion of these statements, refer to the module *Tex* in Chapter 11, »The Standard Library.«

String Extensions

Standard Modula-2 allows the assignment of arrays only if both sides are of the same type, and it completely forbids comparisons of arrays. In Turbo Modula-assignments and comparisons of strings are allowed. A string is any variab whose type is an array with elements of type CHAR. The starting and endi bounds do not matter; strings are assigned and compared as if both strings invol ed start at the same lower bound.

The end of a string is denoted either by the end of the array (if the array is cor pletely filled) or by the null character 0C. Any two strings can be assigned to ea other, even if they are defined as different lengths or start at different bound

Additionally, any two strings can be compared using the relational operators `< >`, `#`, `<`, `>`, `<=`, or `>=`.

Multidimensional Open Arrays

Turbo Modula-2 allows open array parameters to be of any dimension. This exte sion is invaluable for programs that use matrixes. The standard procedure *HIG* will return the highest bound for each dimension. The next higher bound is fou by passing *HIGH* the variable with »[0]« appended (see Chapter 6, »Open Arr Parameters«).

Error-Handling Extensions

Program errors can be divided into the following three types:

Compiler-time errors	Syntax errors discovered by the compiler (for example, a missing semicolon or a misspelling)
I/O errors	Problems occurring during input/output operations (for example, file not on disk or file not open)
Runtime errors	Errors occurring while executing a program (for example, division by zero or integer type passing out of range)

Compiler errors are corrected before the program can be run. However, I/O and runtime errors are only apparent while executing a program.

Should an error occur while running a program, the computer has two options: (1) to write an error message on the screen and halt the program, or (2) to send an error signal to the program.

The first option is fine for simple programs, but could have disastrous results in real-time applications. The second method requires that the running program test some kind of flag after each operation capable of producing an error. This can clutter up the program's logic and reduce efficiency, particularly if the tested condition occurs only in rare circumstances.

As an example, consider a disk-write operation. Testing for a full disk after each disk write is clearly inconvenient. On the other hand, halting the program unconditionally prevents the running program from reacting to this error.

Pascal solves this problem by using a compiler option that determines the program's reaction to I/O errors. However, this is not possible in Modula-2 since disk writes are performed in a library module not within the control of the compiler. It is evident that a system consisting of several largely independent modules needs some way to signal error conditions.

Exceptions serve this purpose well. We do suggest, however, that you use exceptions only if there is no other appropriate way to handle errors. In general, their use should be confined to the signaling of errors across module boundaries. (There are a number of predefined exceptions in Appendix D.)

Syntax and Semantics of Exception Handling

An exception consists of three parts: the *exception declaration*, which defines the

exception identifier and uses the reserved word **EXCEPTION**; the *excepi handler*, which is the programmed response to the error condition; and the *RA.* statement, which calls the exception.

An advantage of exception handling is that handlers are present statically in program text, near the location they are needed. In comparison to other eri handling methods, exception handlers are much clearer. They also follow nested structure of the Modula-2 language itself.

Declaration of Exceptions

An exception declaration contains the reserved word **EXCEPTION**, followed a list of identifiers. Exception declarations have the following form:

EXCEPTION < identifier list > ;

And the following example is taken **from the module** *Files*:

EXCEPTION
EndError, StatusError, UseError, DeviceError, DiskFull;

All of the usual scope rules of Modula-2 apply here. Exception identifiers be exported and imported like normal Modula-2 identifiers.

Raising Exceptions

Exceptions are raised when the program detects an error condition; for exam] when the module *Files* has detected that the disk is full. Raising an exception transfer control to an exception handler that is provided by either the user or system.

A program may raise an exception with the reserved word **RAISE**, followed the exception identifier and optionally by a string. The **RAISE** statement ta the form

RAISE < exception identifier > , < string > ;

This next example is taken from **the** standard module *MathLib*.

```
IF x < 0.0 THEN
 RAISE ArgumentError, 'Negative argument for Sqrt';
END ;
```

When an exception is raised, the system looks in the current procedure for a matching exception handler. If none is found, the calling procedure is examined, then the caller of that procedure, and so on, until a matching exception handler is found. This handler is then executed, and the procedure containing the handler is exited. If no handler is found, the system prints the exception identifier's name and the optional message string.

There is an alternate form of the RAISE statement that is only allowed within exception handlers. It is the reserved word RAISE by itself, as shown in the following:

RAISE ;

This has the effect of passing an exception through one handler and on to the next in the calling chain.

Exception Handlers

Exception handlers are written at the end of procedures and modules to handle exceptions issued by a RAISE statement. The syntax is similar to the familiar CASE statement. Exception handlers have the following form:

```
EXCEPTION
   < vertical bar>
   < exception identifier list>  :  < statement sequence>
   < vertical bar>
   < exception identifier list>  :  < statement sequence>
   . . .
ELSE
   < statement sequence>
END < procedure identifier> ;
```

Exception handlers consist of the reserved word EXCEPTION, followed by any number of exception cases. Each exception case is separated by a vertical bar, and each consists of a list of exception identifiers followed by a colon and a statement

sequence. In addition, an **ELSE** part is executed if the raised exception is not
any of the exception identifier lists.

   In the following example, pretend you are running a program-controlled la:
experiment and want to guarantee that the experiment turns off in all cases.

```
MODULE LaserExperiment;
IMPORT MathLib;

 MODULE GuardedMath;
 FROM MathLib IMPORT Exp;
 EXPORT ExpG, UndefinedExp;
EXCEPTION UndefinedExp;          (* Exception Declaration *)

 PROCEDURE ExpG(X,Y:REAL):REAL;

 BEGIN
  IF (1.E-10<X> AND (X<1.E10) THEN RETURN Y/X
  ELSE RAISE UndefinedExp        (* Signal an error *)
  END
 END ExpG;

 END GuardedMath;

(* Procedure Declarations *)

 BEGIN
  SetUpExperiment;
  PerformExperiment;
                                 (* Normal termination *)
  TurnOffLaser;
  ShutDownLab;
  PrintOutResults;

 EXCEPTION                       (* Exception Handler *)
  Undefined Exp: WRITELN("WARNING: Bad Math");
                 TurnOffLaser;
                 ShutDownLab;
                 DumpProgramVars;
```

ELSE

```
                TurnOffLaser;
                ShutDownLab;
```

END LaserExperiment.

There is one explicit exception raised in a special exponentiation routine. The exception handler in the main body of code has two clauses: one to catch the exception explicitly raised in *ExpG*, the other to catch all other possibilities.

If no exception handler is present for a given exception condition, the **RAISE** statement writes the exception identifier and an optional message to the screen, and halts the program. For example:

RAISE ZeroDivision;

would react to an error by printing the following message:

```
ZeroDivision in module Division
Press "C" for calling chain >
```

If a more comprehensive error message is desired, an additional string can be included in the **RAISE** statement. The string can be a literal or any variable whose type is an array with elements of type **CHAR**. The string will be printed below the exception name, like so:

```
DiskFull in module MYPROG
While processing file OUTDATA
Press "C" for calling chain >
```

A program can respond to an exception by appending a handler to the end of the program unit where an exception may be raised. Such a program unit can be a procedure or the main program. When an exception is raised, normal execution is suspended and the corresponding exception handler is invoked; that is, the handler is executed instead of the rest of the program unit.

Exception Propagation

A scheme requiring an exception handler in the program unit where the exception is raised is really not very useful. After all, you need exceptions to signal your

program that an error has occurred in a called library module. There must be some way to propagate exceptions. If the procedure that raises an exception does not contain a handler for it, the procedure is aborted completely and its caller is searched for a handler.

If the calling procedure also does not contain a handler, the search is continued in the procedure that called the caller, and so on. This continues until a handler for the raised exception is found, or until the main program (or a coroutine) is reached. Note that the order in which procedures are searched for exception handlers duplicates the order in which they are displayed in the calling chain.

If a handler for a raised exception is found in the calling chain, it is invoked and thus replaces the rest of the procedure containing the handler. If all statements in the handler are executed, the procedure returns. Note that a program cannot be restarted at the point where the exception is raised. This restriction makes the exception-handling mechanism relatively safe. The restriction is not severe, since an exception can easily be converted into a flag that indicates an unsuccessful operation.

A handler for an exception can of course raise an exception itself. In this case, the exception is always propagated to the next procedure in the calling sequence thus preventing an infinite loop where every time an exception is handled, another is raised. To propagate the handler exception to higher-level procedures, a handler can raise the handled exception again by using the short form of the **RAISE** statement. If the following statement is found in some exception handler, the handled exception is propagated to the calling procedure (or module):

```
EXCEPTION
  EndError,DeviceError: Close(infile); RAISE
END ReadFile;
```

If a library module should signal an exception, it must first be declared with an exception declaration. An exception declaration can be given anywhere in program where constant, variable, or type declarations would be legal. And like other identifiers, exceptions can be exported and imported.

Many of the library modules export exception identifiers that can be incor porated into programming error handlers. For example, the following program reads data input from the keyboard and writes it to disk. The exception DiskFu

is declared in the module *Files* and raised when the disk is full (*Texts* is implemented with *Files*).

```
MODULE DataWrite;
 FROM Files IMPORT DiskFull;
 FROM Texts IMPORT TEXT,OpenText,CloseText;
 FROM Comline IMPORT PromptFor;
VAR Value: CARDINAL;
 DataFile: TEXT;
 Str: ARRAY [0..30] OF CHAR;
BEGIN
Str:= "Input value (99 to stop): ";
(* Request filename and open file *)
PromptFor("Data file name: ",DataName);
OpenText(DataFile,DataName);
(* Request data from keyboard and write to file *)
 WRITELN(Str);
 READ(Value);
 WHILE Value # 99 DO
  WRITE(DataFile,Value);
  WRITELN(Str);
  READ(Value2);
 END ;
(* Close completed file *)
  CloseText(DataFile);
 (*If disk full, die *)
EXCEPTION
 DiskFull: WRITELN("DISKFULL - YOU JUST LOST ALL YOUR INPUT");
END DataWrite.
```

Chapter 10
System Operations

Turbo Modula-2 is a menu-driven package similar to Turbo Pascal. However, Turbo Modula-2 is a more complex system, containing more menu options and operating differently than Turbo Pascal.

In this chapter, we'll take a look at the basic system operations of Turbo Modula-2. As shown in Chapter 2, the Turbo Modula-2 main menu looks like this:

```
Selected drive: A

Work file:

Edit   Compile   Run    eXecute

Link   Options   Quit   liBrarian

Dir    Filecopy  Kill   reName Type >
```

The first two items on the main menu (Selected drive and Work file) are default values used by the other menu items.

Selected drive

Allows you to log a drive as the default disk drive. This command has the side effect of resetting the drive that is specified, so the BDOS read- only error is avoided. (This command is the same as Turbo Pascal's *L* command.)

Work file

Work file is a file name or partial file name that can be prompted with certain menu commands, such as Compile, Link, Run, Edit, and Find runtime error. It is essentially a default user response in which the user can either specify a whole file name (like A:MYMOD) or a partial string (like B:). These default responses can be overridden; see the section in this chapter, »Avoiding the Menus.«

The Run and eXecute commands are used to execute runnable files from within the shell. The remaining menu items are used for file management. (**Note:** Each item is initiated by pressing the item's capital letter.)

File-Management Utilities

The following five file-management commands provide most of the file facilities you will need when writing Modula-2 programs. Whenever these commands accept a drive specifier, you may also specify a user area. It takes the form

```
DU:filename.ext
```

where *D* stands for the drive letter (*A* through *P*) and *U* stands for a user area (0 through 32). Hard disk users will find this facility invaluable for isolating different projects in different user areas.

Dir

Lets you display a directory. You can mask certain file names by specifying an ambiguous file name at the mask prompt. In addition, the Dir command lists the remaining space on the disk from which the directory is read.

You can use question marks (?) and asterisks (*) as »wild cards« in the mask. A question mark represents an arbitrary letter (including none) and an asterisk represents an arbitrary string. Here are some examples of allowed directory masks:

A:	Lists the names of all files on disk drive A:
B:*.SYM	Lists the names of all files having extension .SYM on drive B:
A:TEXTS.*	Lists the names of all files with the file name *TEXTS*, regardless of extension
T????.*	Lists the names of all files on the logged drive whose file name begins with a *T* and has five letters or less
A10:*.*	Lists the names of all files on drive A: in user area 10

If you enter nothing after the prompt and press ⎡⏎⎤ then the names of all files on the currently logged disk drive are displayed.

The files are listed by number, which you can use to refer to individual files or groups of files in the Filecopy and Kill commands. Groups of files are specified as a range, for example, 4-9 or 1-20. If there are more file names than will fit on the screen, you will receive the following prompt:

```
"C" to continue >
```

⌐ Press ⎡ c ⎤ to continue the directory listings; pressing any other key will stop it.

Filecopy

Allows you to copy a file to another file. The source can be specified either as a complete or ambiguous file name, or as file numbers; the two methods cannot be mixed. The files can be referred to by number only if a Dir command has been performed. As an example, if copying files by numbers:

```
>D

Directory mask: a:
1: PIPES     .DEF  4: PIPES     .SYM  7: PIPETEST  .MOD  10:T2        .MOD
2: PIPES     .MCD  5: PIPETEST  .BAK  8: T         .MOD  11:T3        .MOD
3: PIPES     .MOD  6: PIPETEST  .MCD  9: T1        .MOD  12:TYPETRAN  .MOD
Bytes Remaining on A: 145K


>F
Copy from: 1 2 8-11
Copy to : c:
Copying A00:PIPES    .DEF
Copying A00:PIPES    .MCD
  pying A00:T         .MOD
Copying A00:T1       .MOD
Copying A00:T2       .MOD
Copying A00:T3       .MOD

>
```

Kill

Allows you to delete a file. The files may be referred to by their number if a Dir command has been performed. When Killing by number, you are prompted for

each deletion. If you specify only a mask (ambiguous file name), then you ar
not prompted at all. For instance, look at this example of Killing by numbers

```
>D

Directory mask: B:
1: SAMPLE  .DEF
2: SAMPLE  .MCD
3: SAMPLE  .MOD
4: SAMPLE  .SYM
Bytes Remaining on B: 75K

>K

Kill file: 2 4
Delete B00:SAMPLE.MCD (Y/N)? Y
Delete B00:SAMPLE.SYM (Y/N)? N
```

Note that since CP/M allows numbers as file names, the Kill commar
distinguishes between names and file numbers by insisting that a space be plac
before file names that consist of numbers (this also applies to Filecopy).

reName

Allows you to change the name of a file. You may only change one **name at a tim**
A warning is given if you overwrite another file. For example:

```
>N

Rename from: sample.mod
Rename to  : sample.mud
```

Type

Allows you to display the contents of a file on the screen without leaving Tur
Modula-2 or using its editor. (This works the same as CP/M's TYPE utility.) Pr
[CTRL] [S] to pause and [CTRL] [C] to stop. For example:

```
>T

Type file: sample.mod

MODULE sample;

   .

   .

   .

END sample.
```

Options

Provides you with a collection of infrequently used but occasionally useful options. If you press ☐ O ☐ (for Options), the following submenu will appear:

```
compiler options:

List  (ON )  Native    (OFF)  eXtensions   (ON )
Test  (OFF)  Overflow  (OFF)  Upper=lower  (OFF)


Path to search: SYSLIB

Find run-time error

Save current selection    Quit

>
```

The first six items on the option menu are compiler switches, which affect the way source code is compiled. These switches are global in that they affect the whole file; however, they are overridden if the source being compiled has embedded switches.

Each of the six switches has an embedded counterpart. A brief description of each is given here. (For a complete description of the following compiler switches and how to use embedded compiler switches, refer to the section in this chapter,

»Compiler Options and Switches.« A brief description is also given in Appendix C.)

☐ List toggles compiler listing on and off.

☐ Native toggles native code generation on and off.

☐ eXtensions allows or disallows Turbo Modula-2 extensions.

☐ Test toggles bounds-checking by assignment-compatible assignments assignments (for more information see »Compiler Options and Switches,« later in this chapter).

☐ Overflow toggles INTEGER overflow-checking.

☐ Upper=lower toggles case sensitivity on and off in identifiers.

Path to search

This entry holds the names of the libraries files that will be searched to find separately compiled imported modules. Library files are created with the librarian (see the section, »The Librarian,« later in this chapter). This option contains the entry SYSLIB, which refers to the file SYSLIB.LIB that contains the standard library modules.

Find runtime error

This utility helps you find bugs that occur at runtime. By accepting a module name and a PC number, it can determine where a program has stopped. The module name and PC number are obtained from the calling chain that is displayed when the error occurs. (For details on the calling chain, refer to the section Appendix E, »The Calling Chain.«)

Save current selection

This option selects the current switch settings and library path as the new default setting. The current selections are made permanent by writing them to the M2.COM file; thus, this file must always be available.

Quit

This selection returns you to the main menu.

Avoiding the Menus

A menu-driven shell makes life a lot easier for beginners, but for experienced users it can sometimes be tiresome. Therefore, we have included a way to circumvent the menu scheme. If you answer the menu prompt with the space bar, you are allowed to enter the full selection sequence, optionally followed by additional arguments on the same line. After the selected command is executed, you are again placed in the menu where the blank was entered. The following examples demonstrate this:

```
> C B:MYPROG
```

Compiles the file B:MYPROG.MOD at once. It is equivalent to this dialogue:

```
>W
```

```
Work File: B:MYPROG
```

```
>C
```

```
Compile file: B:MYPROG
```

This next example shows how prompts are avoided:

```
> D B:
```

lists the directory of drive B, as the following dialogue would

```
>D
```

```
Directory mask: B:
```

This method only works on the main menu; you cannot, for example, toggle the native code generation option with the string > ON.

The Turbo Editor

The built-in editor is a screen editor specifically designed for creating progra
text. If you are familiar with Turbo Pascal or WordStar, you'll need little instru
tion in the use of the editor. The Turbo editor includes some extensions, plus yo
can install your own commands on top of the WordStar commands (described
Appendix B, »Installation Procedures«) and the WordStar commands will rema
usable.

Using the Turbo editor is simple: After you have defined a work file and presse
⌑E⌑ (for Edit), the menu will disappear and the editor is activated.
the work file exists on the logged drive, it is loaded and the first page of text
displayed. If it is a new file, the screen is blank, apart from the status line at th
top.

To terminate a line, press the ⌑↵⌑ key. When you have filled the screen wit
text, the top line will scroll off the screen, out of view. You may page back ar
forth in your text with the editing commands described later in this section.

First, let's take a look at the information the status line provides at the top
the screen.

X:FILENAME.TYP. Shows the drive, user area, and name of the file bein
edited.

Line n. Shows the line number that contains the cursor, counting from the sta
of the file.

Col n. Shows the column number that contains the cursor, counting from the le
side of the screen.

Char n. Shows the character number that contains the cursor, counting from th
beginning of the file.

Insert. Indicates that characters entered on the keyboard are inserted at the curs
position. Existing text in front of the cursor is pushed to the right. The inse
mode on/off command, ⌑Ctrl⌑ ⌑V⌑ , switches this message to *Overwrite*, whic
means text entered on the keyboard overwrites characters under the cursor, instea
of being inserted.

Indent. Indicates that auto-indentation is active. It may be switched off with the auto-indent on/off command, [Ctrl] [Q] [I], in which case this space on the status line is blank.

Operating The Editor

This editor is a full-screen editor, which means you can move the cursor anywhere on the screen and begin writing. This is done by using a special group of control characters: pressing the [Ctrl] key while simultaneously pressing any of the keys, [A], [S], [D], [F], [E], [R], [X], or [C]

The characters are arranged on the keyboard in a manner that logically indicates their use. For example, in the following display:

$$E$$
$$S \qquad D$$
$$X$$

pressing [Ctrl] [E] will move the cursor up, [Ctrl] [X] moves it down, [Ctrl] [S] moves it to the left, and [Ctrl] [D] moves it to the right. If your keyboard has repeating key capability, you may hold down the [Ctrl] key and one of these four keys to move the cursor rapidly across the screen.

Editing Commands

The editor accepts and uses many editing commands that move the cursor, page through the text, find and replace text strings, and so on. These commands can be grouped into the following categories:

☐ Cursor movement commands
☐ Extended movement commands
☐ Insert and delete commands
☐ Block commands
☐ Find and replace commands
☐ Miscellaneous commands

Each group contains logically related commands that are described in the following sections. (A summary of the commands is provided in Table 10-1.) The following descriptions consist of a command definition, followed by the default keystrokes used to activate the command. If you would like to redefine the commands, refer to »Installation of Editing Commands« in Appendix B.

Table 10-1 Summary of Turbo Editor Commands

Cursor Movement Commands

Character left	`Ctrl` `S`
Character right	`Ctrl` `D`
Word left	`Ctrl` `A`
Word right	`Ctrl` `F`
Line up	`Ctrl` `E`
Line down	`Ctrl` `X`
Scroll up	`Ctrl` `W`
Scroll down	`Ctrl` `Z`
Page up	`Ctrl` `R`
Page down	`Ctrl` `C`

Extended Movement Commands

Left on line	`Ctrl` `Q` `S`
Right on line	`Ctrl` `Q` `D`
Top of window	`Ctrl` `Q` `E`
Bottom of window	`Ctrl` `Q` `X`
To top of file	`Ctrl` `Q` `R`
To end of file	`Ctrl` `Q` `C`
To beginning of block	`Ctrl` `Q` `B`
To end of block	`Ctrl` `Q` `K`
To last cursor position	`Ctrl` `Q` `P`

Insert and Delete Commands

Insert mode on/off	`Ctrl` `V`
Delete left character	`Del`
Delete character under cursor	`Ctrl` `G`
Delete right word	`Ctrl` `T`
Insert line	`Ctrl` `N`
Delete line	`Ctrl` `Y`
Delete to end of line	`Ctrl` `Q` `Y`
Delete line up to cursor position	`Ctrl` `Q` `H`

Cursor Movement Commands

Block Commands

Mark block begin	Ctrl	K	B
Mark block end	Ctrl	K	K
Hide block	Ctrl	K	H
Copy block	Ctrl	K	C
Move block	Ctrl	K	V
Delete block	Ctrl	K	Y
Read block from disk	Ctrl	K	R
'te block to disk	Ctrl	K	W

Find and Replace Commands

Find	Ctrl	Q	F
Find and replace	Ctrl	Q	A
Repeat last find	Ctrl	L	

Miscellaneous Editing Commands

Delete file on disk	Ctrl	K	J
Save file, exit	Ctrl	K	D
Save, edit	Ctrl	K	
Quit, no save	CTRL	K	Q
Tab	CTRL	I	
Auto-indent on/off	CTRL	Q	I
Control character prefix	CTRL	P	
Abort operation	CTRL	U	

·sor Movement Commands

Character left
CTRL   S

Moves the cursor one character to the left nondestructively (without affecting any characters). When at the start of the line, the cursor will move to the end of the previous line.

Character right
CTRL   D

Moves the cursor **one character to the right nondestructively (without** affecting

any characters). When the last character on the line is reached, the cursor will move to the first character on the next line.

Word left   [CTRL] [A]

Moves the cursor to the beginning of the word to the left.

Word right   [CTRL] [F]

Moves the cursor to the beginning of the word to the right.

Line up   [CTRL] [E]

Moves the cursor to the preceding line. If there is no character in the current col-umn, the cursor is moved to the end of the line.

Line down   [CTRL] [X]

Moves the cursor to the proceeding line. If there is no character in the curren column, the cursor is moved to the end of the line.

Scroll up   [CTRL] [W]

Scrolls the file up one line toward the beginning of the file (the entire screei scrolls down).

Scroll down   [CTRL] [Z]

Scrolls the file down one line toward the end of the file (the entire screen scroll up).

Page up   [CTRL] [R]

Moves the cursor one page up.

Page down   [CTRL] [C]

Moves the cursor one page down.

Extended Movement Commands

The editor provides commands to quickly move to either end of a line, to the beginning and end of the text, and to the previous cursor position. These commands require two control characters to be entered simultaneously: press `CTRL` `Q` and then one of the control characters, `S` , `D` , `E` , `X` `R` , or `C` . Their keyboard arrangement repeats the pattern previously shown.

<div align="center">

E   R
S   D
X   C

</div>

Left on line   `CTRL` `Q` `S`

Moves the cursor to the far left of the screen (column 1).

Right on line   `CTRL` `Q` `D`

Moves the cursor to the far right (the end) of the current line.

Top of window   `CTRL` `Q` `E`

Moves the cursor to the top of the screen.

Bottom of window   `CTRL` `Q` `X`

Moves the cursor to the bottom of the screen.

Top of file   `CTRL` `Q` `R`

Moves to the first character in the file unless the top of the file has been paged out, in which case the cursor is moved to the first character in the buffer. To move to the start of a file that has had the beginning paged out, you must use `CTRL` `K` `S`, which also saves any changes the were made. If you don't want to save your changes, you must quit (`K` `Q`) and restart the editor.

End of file   `CTRL` `Q` `C`

Moves to the last character in the file.

Beginning of block   CTRL   Q   B

Moves the cursor to the block-begin marker set with CTRL   K   B .

End of block   CTRL   Q   K

Moves the cursor to the block-end marker set with CTRL   K   K .   (Bloc commands only work if a block is fully marked.)

Last cursor position   CTRL   Q   P

Moves to the last position of the cursor. This command is particularly useful aft a find or find/replace operation has been executed and you'd like to return to tl last position before its execution.

Insert and Delete Commands

Insert mode on/off   CTRL   V

When you write text, you may choose between two entry modes: *Insert* and *Ov write*. The current mode is indicated in the status line. The CTRL   V   co: mand allows you to switch between these modes.

Insert mode inserts characters at the cursor position; existing text to the right the cursor will move to the right as you write new text.

Overwrite mode may be chosen if you wish to replace old text with **new te** Characters entered will overwrite any characters under the cursor.

Delete left character   [

Deletes the character to the left of the cursor. This can also be used to rem line breaks.

Delete character under cursor   CTRL   [

Deletes the character under the cursor.

Delete right word $\qquad$ [CTRL] [T]

Deletes the word or part of the word to the right of the cursor. It may also be used
to remove line breaks.

Insert line $\qquad$ [CTRL] [N]

Inserts a new line at the cursor position.

Delete line $\qquad$ [CTRL] [Y]

Deletes the line containing the cursor. No provision exists to restore a deleted line,
so use this cautiously.

Delete to end of line $\qquad$ [CTRL] [Q] [Y]

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

Block Commands

All block commands are extended commands (that is, the standard command
definition consists of two characters). You can use them to move, delete, or copy
whole chunks of text, and to perform certain file operations.

A block of text (which you determine) is marked by placing a block-begin
marker at the first character and a block-end marker after the last character of
the desired text.

The marked block may be copied, moved, deleted, or even written to a disk file.
There are also commands to read or write an external disk file into or out of the
editor as a block.

Mark block begin $\qquad$ [CTRL] [K] [B]

Marks the beginning of a block. The marker itself is not visible on the screen,
and the block only becomes visibly marked if the block-end marker is set. (This
only applies if your terminal has highlight capabilities.)

Mark block end   `CTRL`   `K`   `K`

Marks the end of a block. As stated previously, the marker itself is not visible on the screen; it is only visible if the block-begin marker is also set.

Hide block   `CTRL`   `K`   `H`

Causes the visual marking of a block to be switched off and removes block markers. After this command, no block markers are visible and all block commands are invalid.

Copy block   `CTRL`   `K`   `C`

Copies a previously marked block to the cursor position. The original block is left unchanged, and the markers are placed around the new copy of the block.

Move block   `CTRL`   `K`   `V`

Moves a previously marked block from its original position to the cursor position. The markers remain around the block in its new position. If no block is marked, nothing happens.

Delete block   `CTRL`   `K`   `Y`

Deletes the previously marked block. No provision exists to restore a deleted block, so use this cautiously.

Read block from disk   `CTRL`   `K`   `R`

Reads a file into the current text at the cursor position, exactly as if it were a block that was moved or copied. The text read in from the disk is marked as a block

When using this command, you are prompted to give the name of the file to b read (after which you must press `↵`). The file specified may be any legal fil name; no default extension is supplied.

Write block to disk   `CTRL`   `K`   `W`

Writes a previously marked block to a file. The block is left unchanged, and th markers remain in place. When this command is issued, you are prompted for th

name of the file to write to. If the file specified already exists, you are asked if you want to continue before the existing file is overwritten. If no block is marked, you will get an error message.

The file specified may be any legal file name; no default extension is supplied.

Find and Replace Commands

These commands allow you to search for a string of characters or replace one string with another, which is useful when correcting a repetitive error.

Find   [CTRL]   [Q]   [F]

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 desire and terminate by pressing [⏎] .

Search strings may be edited with the Character left, Character right, Word left, and Word right commands. To recall the previous search string, press [CTRL] [L] when prompted for the new string. A search operation may be aborted with the Abort command, [CTRL] [U] .

The search string may contain any characters, even control characters. Enter control characters into the search string by using a [CTRL] [P] prefix. For example, to enter a Control-A, hold down the [CTRL] key and press [P] , then press [A] . Thus you may include a line break in a search string by including Control-M/Control-J (press [CTRL] [P] [M] and [CTRL] [P] [J], respectively.

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

U   Ignores uppercase/lowercase. Uppercase and lowercase alphabetic characters are regarded as equal.

W   Searches for whole words only; skips matching patterns that are embedded in other words.

B   Searches backward.

digits   Finds the *n*th occurrence of a search string.

For example:

BU   Searches backward and ignores uppercase/lowercase. 'Block' will match
          both 'blockhead' and 'BLOCKADE'.

W   Searches for whole words only. The search string 'term' will only **match**
          the word 'term', not the word 'terminal'.

B5   Searches backward and finds the fifth **occurrence.**

     Terminate the list of options (if any) by pressing ⏎ ; the search will then
begin. If the text contains a target matching the search string, the cursor is posi-
tioned at the end of the target, unless you're searching backward, in which case
the cursor is positioned at the start of the target.

     The search operation may be repeated by the Repeat-Last-Find command, CTRL
L .

Find and replace   CTRL   Q   A

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 require and terminate by pressing ⏎
Again, however, control characters entered into the search string must use a CTRL
P prefix. (See the previous example in the »Find« section.)

     As mentioned in the »Find« section, search strings may be edited with the
Character left, Character right, Word left, and Word right commands. (Refer to
the »Find« section for more information.)

     When the search string is specified, you are asked to enter the string that will
replace the search string. You can enter up to 30 characters; control-character en-
try and editing is performed as stated previously. If you press ⏎ , the target
string will not be replaced but will be deleted.

Finally you are **prompted** for **options**. The find-**and-replace** options include the following:

G   Global search and replace; searches and replaces in the entire text, starting at the current cursor position.

N   Replaces every item without asking *Replace (Y/N)* for each **occurrence of** the search string.

U   Ignores uppercase/lowercase; **uppercase and lowercase alphabetic** characters are considered equal.

W   Searches and replaces whole words **only; skips matching patterns that are** embedded in other words.

For example:

GWU   Find and replace **whole words in the remaining text. Ignore upper-**case/lowercase.

Terminate the list of options (if any) by pressing [⏎] ; the search- and-replace operation wili begin. If found (and the N option is not specified), the cursor is positioned at the end of the target and you are asked

```
Replace (Y/N)?
```

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]

Repeats the latest **find-and-**replace operation **exactly as if all informa**tion had been reentered.

Miscellaneous Editing Commands

This section lists the commands that do not fall into any of the earlier categories.

Save and quit   `CTRL`   `K`   `D`

Saves the text file and also gives the original file (if any) the last name .BAK. You
are then returned to the Turbo Modula-2 shell.

Tab   `TAB`   `CTRL`   `I`

Unlike Turbo Pascal, tab positions are fixed to multiples of eight; however,
Modula-2 contains the same Autotab feature present in Turbo Pascal.

Indent on/off   `CTRL`   `Q`   `I`

The indent feature provides automatic indentation. When active, the indentation
of the current line is repeated on each following line. To change this indentation,
use the space bar and `←` keys to select the new column.

When indent is active, the message INDENT is displayed in the status line; when
passive, the message is removed. Autotab is active by default.

Save and edit   `CTRL`   `K`   `S`

Saves the file and remains in the editor. Press `CTRL` `Q` `P` to return to
where you issued the command.

Quit/no save   `CTRL`   `K`   `Q`

Terminates the editor. If changes were made to the text, you are prompted as to
whether or not you want to abandon the edited file. Press `Y` for yes; other-
wise, press any other key.

Delete file   `CTRL`   `K`   `J`

Allows you to delete files from within the editor. This is handy if you run out of
disk space after entering lots of text you want to save.

Abort operation   `CTRL`   `U`

Pressing `CTRL` `U` will let you abort any command whenever it pauses for in-
put.

The Librarian

A typical Modula-2 development system uses a large number of files. There are four files for each library file: two that define it and two that make it usable. It is typical in Modula-2 to write and debug a library routine once and then use it many times. Thus it is the it is the second two files that must be kept around. These are the .SYM and .MCD files, which hold the compiled definition and the compiled code, respectively.

   The liBrarian makes the storage of these files easier by allowing you to keep many .SYM and .MCD files in a CP/M file called a library file. Thus, if you ℓ lop 10 library modules for your general use, you can store the compiled versions in one file instead of 20.

   So, the compiler, the program loader, and the linker all need to know which libraries should be searched for files. This information is given in the search path. As mentioned earlier, the search path can be changed in the Options submenu, which initially consists only of the entry SYSLIB.

Searching Libraries

After a library has been created, the modules included in the library will be invisible to the system until the name of the new library is entered in the »Path to search« selection of the Options submenu.

   Libraries are manipulated via the liBrarian, which is selected by pressing ⬚B⬚ in the main menu. The liBrarian first prompts the user to select which library to work on:

```
>B
select library:
```

   If you enter MYLIB.LIB, then the following menu will appear:

```
Selected library: A:MYLIB.LIB

Dir            Include       Copy

Kill           cOmpress      quit
*
```

You may enter your selection from this menu after the * prompt. The choice
have the following effects:

Include   A CP/M file is included in the specified library. Only th
                    CP/M extensions .MCD or .SYM are allowed.

Dir   Displays all the modules already included in the use
                    library; includes size information.

Copy   Moves files out of the library, either by copying them int
                    another library or to a CP/M file. If the destination nam
                    has a drive code, it is considered to be a CP/M file name
                    without a drive code, the name specifies the destinatio
                    library.

Kill   Terminates a module's existence inside the used library

cOmpress   Copies all files in a library in such a way that there ar
                    no unused »holes.« This command is used quite rarel
                    since it takes a lot of time.

Select   Selects another library.

Quit   Returns the user to the main menu.

The Compiler

The Turbo Modula-2 compiler translates a Modula-2 program (the *source*) int
a sequence of instructions (the M-code) that form the code of a virtual machin
They are optimized for compactness, to make possible execution of large pro
grams in the limited address space of 8-bit computers.

The Z80 processor cannot execute M-code directly; instead, every instructic
is executed by a special program, the M-code Interpreter. In comparison to
directly interpreted language like BASIC, M-code operates much faster. In mo
applications, it approaches machine speed but needs less memory. For those cas
where speed is critical, there is a native code option that tells the compiler to pr

duce Z80 machine code. (For more on this topic, see the section, »Compiler Options and Switches,« later in this chapter.)

The translated program is called an *object program*. The source can only be read by the programmer, while the instructions of an object program can only be executed by the computer. The translation proceeds in a single pass through the source program; in other words, Turbo Modula-2 uses a one-pass compiler for generating M-code. When native code is desired, an additional pass is executed.

The Turbo Modula-2 compiler serves another purpose: If several modules are designed to work together, the programmer must first prepare one or more definition modules. A definition module contains information about parts of the corresponding implementation module that can be used by other modules. The compiler translates the definition module's information into a symbol file. These files do not contain M-code instruction; they merely reference parts of the corresponding implementation module. Whether a program's text constitutes a definition module or not is decided by the reserved word **DEFINITION** in the header line of the module. (Definition modules usually have the extension .DEF in their file name.)

Operating the Compiler

Before a source program is compiled, its program text is written using the editor and then stored on disk. You will have usually provided Turbo Modula-2 with a work-file name for this purpose. The compiler is started from the Turbo Modula-2 shell by pressing ⌈ c ⌋ . Assuming a work-file name of DEMO and a default disk drive of B, Turbo Modula-2 will respond by prompting you with the message: Compile file:B:DEMO. You can confirm this by pressing ⌈ ↵ ⌋ .

Source programs usually have the extension .MOD. This extension can be omitted in the file name, and the compiler will append it automatically.

Note: If a source file has no extension at all, the file name cannot be entered »as is,« since the compiler will then look for a file with the extension .MOD. This problem is solved by appending a single period (.) to the file name. Also note that the Turbo Modula-2 editor assumes the default extension .MOD.

Once the name of the source is entered, the compiler starts execution. It continuously reads the source from disk and, if the List option is on, displays the translated portion of the source on the screen. When compilation is finished, the

object file is written to the disk where the source resides. You can override thi
convention by including a drive code (letter plus a colon) on the command line
somewhere after the name of the source. For example, Compile file:B:DEMO A
translates B:DEMO.MOD *into* A:DEMO.MCD.

The generated file now assumes the same name as the source file. If it is a
object module, it is given the extension .MCD; if it is a symbol file, it assume
the type *SYM*.

The Listing

During translation the compiler can produce a listing of the compiled progran
text. The listing can be switched on and off in the Options submenu. When switch
ed on, it is usually displayed continuously on the screen, constantly updating yo
about how far the compiler has proceeded in the source text. The listing can b
redirected to a disk file or to any of CP/M's logical output devices, such as th
printer, by including a redirection argument n the command line after the Compil
command. Remember that an argument for output redirection consists of the >
symbol, followed by the name of the desired output medium. For example:

```
Compile file:B:POWERS >B:DEMO.LST
```

creates a file B:DEMO.LST and sends the listing to this file.

A program listing could like this:

```
  | (* $U- *)
  | MODULE Powers;
  | FROM Texts IMPORT Done,input;
  | FROM SYSTEM IMPORT ADR,MOVE,WORD;
  |
  | PROCEDURE Power(x: REAL; i: INTEGER): REAL;
  | VAR t: REAL;
  | BEGIN
 0|  t:=1.0'
 9|  WHILE i > 0 DO
12|    t := * x;
21|    i := i - 1;
25|  END ;
27|  RETURN t
```

```
 271 END Power;
---- Size = 31
   |
   | PROCEDURE GetNum(s: ARRAY OF CHAR; VAR n: ARRAY OF WORD);
   | VAR rec: RECORD
   |              CASE : BOOLEAN OF
   |              | TRUE    : r: REAL;
   | | FALSE             : c: INTEGER;
   |              END
   |         END ;
   | BEGIN
  0|   WITH rec DO
 11|    REPEAT
 11|     WRITE(s);
 16|     IF SIZE(n) = SIZE(REAL) THEN
 22|      READLN(r); MOVE(ADR(r),ADR(n),SIZE(r))
 33|     ELSE
 38|      READLN(c); MOVE(ADR(c),ADR(n),SIZE(c))
 47|     END
 50|    UNTIL Done(input)
 50|   END
 57| END GetNum;
---  Size = 59
   |
   | VAR
   |  x: REAL; n: INTEGER;
   | BEGIN
  0|   LOOP
  2|    WRITELN;
  8|    GetNum("X= ",x);
 18|    GetNum("N= ",n);
 28|    IF n < 0 THEN EXIT END ;
8888.
 34|    WRITELN(x :8:4," to the power of ",n :0,"
     = ",Power(x,n):8:4);
 91|   END
 91| END Powers.
---  Size = 94
   |
```

```
End of source reached.
Compiled bytes: 208
M-code file DOO:POWERS.MCD produced.
```

Besides displaying the program text, this listing offers other information. After each procedure and each module, the line »Size =« indicates the length in bytes of the M-code that was generated. The line »Compiled bytes:« indicates the total length of the M-code generated by the compiler.

The left margin of the source program text is marked by a column of vertical bars. The numbers at the left of the bars are only given on lines that actually generate M-code. They indicate the amount (in bytes) of M-code that is generated between the start of the procedure or module and the current line. These numbers form a coordinate system (procedure name and offset) that allows you to find the location of an error should one occur during execution. Notice that this offset is only correct with M-code. If the native code generation switch has been toggled on, these numbers are meaningless.

If a program's execution fails, the calling chain specifies the offset of the point of error from the enclosing procedure or module. This offset corresponds (in the case of M-code) directly to the numbers in the compiler listing, thereby indicating the error's location in the source. You can demonstrate this effect by entering and compiling the previous program example. If you then run the program and present input that is too large, the program will fail.

In the case of native code, the offset from the calling chain of a runtime error corresponds to the set of numbers given during the second native code pass of the compiler. The numbers will look like this:

```
End of source reached.

NAME      START     len
POWER     4         102
GETNUM    110       240
POWERS    354       313

Compiled bytes: 675

Native-code file DOO.POWERS.MCD produced
```

s, a runtime error **with offset = 53 refers to the second line:**

```
POWER     4          102
```

since 53 is within the first 106 bytes of the file. Since the procedure power is 102 bytes in length, you know the error occurred about halfway through the procedure. An easier method to finding runtime errors is to use the Find runtime error utility provided in the compiler Options menu (described earlier).

The primary use for the numbers in the listing is to determine the size of procedure and modules. This is helpful when using overlays. Normally you will want .o keep the listing option off; using the List option reduces compiler speed by about 30 percent.

Error Correction

The Turbo Modula-2 compiler provides for an interactive error-correction scheme. If an error is detected during the compiler run (for example, a syntax error or a type mismatch), the compiler stops and displays the type of error and the point at which it occurred. You are given the option of loading the source immediately into the editor for correction or stopping the compiler. When correction is completed, the compiler takes over again, resuming its work near where you corrected the error (usually at the start of the same line). This is all done automatically, without having to use the Edit and Compile commands everytime an error occurs. The correction cycle is explained further in the following example.

Assume that the ninth line (labeled by the byte offset 12) of the program listing in the preceding section reads $t = t * x$ instead of $t := t * x$. This line is now illegal, since assignment must be expressed by $:=$ instead of $=$. The compiler will translate up to the first illegal line, displaying all processed lines in the listing. When it detects an error, it lists the currently processed line in full before giving a message and stopping. With the List option on, it would look like this:

```
I (* $U- *)
I MODULE Powers;
I
I PROCEDURE Power(x: REAL; i: INTEGER): REAL;
I   VAR t: REAL;
I BEGIN
```

```
 2|  t:=1.0'
 9|  WHILE i > 0 DO
12|    t = t * x;
           ^ Error in Syntax:
" := " expected, but " = " found
E(dit, Q(uit >
```

The caret (^) points to the last symbol read before detection of the error. Y can abort the compiler run by pressing ⌷Q⌷ , or you can correct the error pressing ⌷E⌷ . If you select ⌷E⌷ , the editor is started for correction, yc program is loaded automatically, and the cursor is positioned at the place wh⟨ the error was detected. In the example, the following line would be in the ⟨en of your screen:

```
t = t * x;
```

The cursor (denoted by the ^) is already placed on the faulty " = ". You ⟨ insert the missing " : " at once.

The editing commands available are those in the Turbo Modula-2 editor. course, you can also »walk around« in the source, fixing other pieces of text. make space for a text buffer, the compiler must store some of its internal data disk before editing takes place. The amount needed varies between 0 and 15 depending on the length of your program. To be on the safe side, reserve a w⟨ space of about 15K on your source disk, plus the size of a .BAK file if there i⟨ one yet.

Once correction is completed, you can leave the editor. To save your corr tions, press ⌷CTRL⌷ ⌷K⌷ ⌷D⌷ . As in normal editing, the old version is still p sent, but now has the extension .BAK. The compiler takes over again and tran⟨ tion is resumed near the initial change in your program. (Pressing ⌷CTRL⌷ ⌷⊦ ⌷Q⌷ will save the old version of your source file and the old .BAK file; ⟨ changes you have made will be lost and compilation will be aborted.)

If, after saving your changes and continuing on with compilation, there are ⟨ more errors in your program, the correction cycle will be repeated.

Running Out of Memory

Following are two instances of memory shortage and their respective causes and solutions.

When the Compiler Runs Out of Memory

The compiler uses the free memory space for two purposes: for its own runtime stack and the heap (that is, the symbol table). Note that the size of the compiled code is inconsequential since it is written to disk as soon as the compiler's internal buffer gets full. In most cases, it's the symbol table that gets too big, which means too many identifiers are visible at a certain point in time. In order to circumvent space problems, you must reduce the number of identifiers. Following are several approaches:

☐ First check whether your program imports objects that it doesn't use. Note that unqualified import (for example, **IMPORT** *Texts*) causes all objects exported by the imported module to be put into the symbol table. The space it occupies equals about the length of the module's .SYM file. It is therefore more economic to import only those identifiers that are actually used; for example,

```
FROM Texts IMPORT TEXT, OpenText, CloseText;
```

☐ Second, try to structure your program. After compiling a procedure, the compiler no longer needs to know about the objects local to the procedure (except for the parameters). Therefore, it can throw away all the information concerning local objects and reuse the space for other purposes. After compiling a local module, the compiler keeps only the exported identifiers. Declare objects as global only as necessary and also declare them as local as possible. Use local modules to hide information that is unimportant to the rest of the program. The number of exported identifiers should be as small as possible. This not only reduces symbol table space, it also makes for a better program.

☐ As a last resort, you may eliminate symbolic constants and enumerated types, replacing their occurrences by their values. For readability and maintenance, the names should be placed in comments. For example:

```
VAR
  color : (*(red,green,blue)*) CARDINAL
BEGIN
  FOR color := 0(*red*) TO 2(*blue*) DO
```

```
        . . .
      END
      END
```

Note that changing *red* to *0(*red*)* in the whole program can be achieved most efficiently via the editor's global replacement command ([CTRL] [Q] [A]).

When Your Program Runs Out of Memory
When this happens, your program probably has too much data and code to fit into memory. You might use overlays if your program has too much code. When you have compiled to M-code, data is usually more critical than code. Following are some common pitfalls that can be avoided and some techniques to make better use of memory:

☐ Be sure to close all your files as soon as possible. Closing a file frees the space used for the file buffer and the file descriptor. When you reopen a file without first closing it, you allocate a new buffer and a new file descriptor without first giving back the old one. Thus, the memory occupied by it is never reused. If you do that within a loop, you will quickly run out of memory.

☐ Passing arrays as value parameters is not bad practice. It ensures that the parameter is not changed by the called procedure. However, it forces the compiler to make a copy of the entire array. If the array is large, this also eats up your memory space. So if you are having space troubles, pass large arrays as **VAR** parameters instead. This also applies to large records.

☐ In contrast to other languages (for example, the CP/M version of Turbo Pascal), the data space of a procedure is not allocated statically, but is created only when you enter the procedure. Therefore, it might be advantageous to declare large structures local to a procedure. Here is an example:

```
MODULE example;
VAR a, b : ARRAY [1..10000] OF CHAR;

PROCEDURE p;
BEGIN
  (* statements involving a *)
END p;
```

```
PROCEDURE q;
BEGIN
 (* statements involving b *)
END q;

BEGIN
 p; q;
END EXAMPLE.
```

This module requires 20 Kbytes of data space to run (10 Kbytes for each of the arrays). This can be reduced to 10 Kbytes:

```
MODULE example;

PROCEDURE p;
VAR a : ARRAY [1..10000] OF CHAR;
BEGIN
 (* statements involving a *)
END p;

PROCEDURE q;
VAR b : ARRAY [1..10000] OF CHAR;
BEGIN
 (* statements involving b *)
END q;

BEGIN
 p; q;
END example.
```

In the second example, the arrays *a* and *b* have been declared local to the procedures *p* and *q*. Thus, *a* and *b* never exist at the same time. As soon as *p* is called, 10 Kbytes of memory are used for *a*. When *p* is finished, the space is given back and used for *b* when *q* is called. Thus, you never need more than 10 Kbytes of memory for the data.

☐ Sometimes you don't know exactly how large an array should be, and using a large one to be safe would take up too much memory. In this situation, you might use a pointer to an array and allocate only as much as you need:

```
MODULE example;
FROM STORAGE IMPORT ALLOCATE;
VAR a, b: POINTER TO ARRAY [1..10000] OF CHAR;
...
BEGIN
(* Let us assume the program knows at this point that a *)
(* and b need only be 2000 characters long *)
 ALLOCATE(a,2000*SIZE(CHAR)); ALLOCATE(b,2000*SIZE(CHAR));
END example.
```

This technique is not without danger, however. The compiler does not know that the array is actually shorter than the declaration says. When the array is accessed, it will therefore check index expressions against the declared bounds and not against the actual bounds. Thus, in the preceding example you can access *a* ^ *[8000[* without receiving a runtime error. Instead, you will access memory that does not belong to *a*. You may thus destroy other variables, your program, or the operating system, and make your program crash in very interesting ways--use this technique with caution.

Symbol Files

Quite often a module imports objects from other modules. It then contains one or more **IMPORT** statements following the header line. An **IMPORT** statement could look like this:

```
FROM MathLib IMPORT Sin,Cos, Exp;
```

The compiler must »know« how the imported procedures *Sin*, *Cos*, and *Exp* are defined in *MathLib*; for example, in order to check for type mismatches. This is achieved by compiling the definition part of the imported module into a symbol file (a file with the extension .SYM). This file must be on the disk either by itself or in a library file (a file with the extension .LIB, which contains many .SYMs and their corresponding .MCD files).

The example presented expects the file SYSLIB.LIB to be on your work disk. If the importing module is compiled, the compiler must access the symbol files of all definition modules from which objects are imported. In the example, the compiler would search for a file called MATHLIB.SYM, in order to »learn« about

the definitions of *Sin*, *Cos*, and *Exp*. If not found, then the compiler would check the library path and search for any files found there.

All symbol files needed during a compiler run must be online. Specifications concerning the drive where one particular symbol file resides need not be given. This is because the compiler automatically searches all drives that you indicated when you installed the compiler with INSTM2 and searches all libraries in the search path for a needed symbol file.

The fact that all needed symbol files must be online can cause problems when the source files of one large programming project are distributed over several ˙isks. The recommended scheme in this case is to combine all the .SYM files and their .MCD files into a common library. This is done with the Turbo Modula-2 liBrarian (described in the previous section).

Compiler Options and Switches

The compiler accepts some options that affect the set of programs that are recognized as valid, the generated object code, and the output of a compiler listing. An option can be set or reset in the source program itself. You can either set options globally by using the switches on the Option menu or set options locally by embedding switches in the code.

This is done by including a dollar sign ($), an uppercase letter, and either the symbol + or - anywhere between two comment brackets; for example, (* $T + *). As you would expect, + turns on the option denoted by the uppercase letter; using - turns it off. An option remains set until the opposite option is given.

Options are set to either on or off by default. The default settings are displayed in the Options submenu of the Turbo Modula-2 shell, where they may also be ˙hanged for the current session and, if desired, saved for future sessions by using ˙ne *S* option.

Default settings are save directly in the M2.COM file. Thus, the M2.COM file must be online. Remember the switches on the option menu are global and may be overridden with embedded switches. All available options appear on the menu as follows:

```
L ist (OFF)   N ative     (OFF)   eX tensions   (OFF)
T est (OFF)   O verflow    (OFF)   U pper=lower   (OFF)
```

Each of these switches has the same effect as their embedded counterparts, but these may be overridden. One exception is the Native switch, which is only effective from this menu. The following describes each switch and presents its embedded code:

$L The List Option

☐ $L+ generates a **listing**.

☐ $L- generates none.

If this option is on, the compiler will emit a listing. The listing appears on the screen unless it is redirected, which is done by entering »> listfile« after the »Compile file« message. For example:

```
>C
Compile file: B:MYFILE >MYFILE:LST
```

If $L is turned off, the listing is suppressed; however, the names of procedures and modules will be displayed when encountered.

$O The Overflow Check Option

☐ $O+ checks for INTEGER and CARDINAL overflow.

☐ $O- does not check for overflow.

If this option is on, the generated object program checks for an overflow when certain operations are performed. An overflow occurs when the result of an operation is too large (or too small) to be represented in the computer's memory. For example:

```
1.    VAR a,b: INTEGER; c: CARDINAL
2.    BEGIN
3.        a := 20000;
4.        b:=a + 15000;
5.        c:=10;
6.        WHILE c >= 0DO
7.            c := c - 1
8.            END ;
```

In both the fourth and seventh line an overflow will occur. In the fourth line the sum evaluates to 35000, while 32767 is the largest representable INTEGER. In the sixth line the program loops. The CARDINAL *c* takes values from 10 down to 0, then *c* is decremented to -1. Since -1 is not a value that can be represented by a CARDINAL, an overflow occurs in line seven. Overflows are signaled by the runtime error message shown here:

```
OVERFLOW in (Name of module)
Press "C" for Calling Chain >
```

If $O is turned off, no error message is given on overflow; instead, the result of the operation is computed *modulo* 2 ^ 16. This method is sometimes desirable for computations; for example, in a random-number generator. Furthermore, suppression of overflow checks can speed up a program considerably. The reader should be warned, however, against some unexpected results, particularly in the case of CARDINAL arithmetic. In the previous example, the variable *c* would be decremented from 0 to 65535; hence the condition $c > =0$ would always be fulfilled and the program would loop infinitely!

Overflow checks can be suppressed only for operations involving CARDINAL and INTEGER addition and subtraction and CARDINAL multiplication. Other operations, such as INTEGER multiplication and all operations dealing with REALs and LONGINTs, should always be checked for overflow or have exception handlers provided to trap errors.

$T The Test Option

☐ $T+ checks array indices and subrange variables.

☐ $T- does not check them.

If this option is on, code for several tests is inserted into the generated object program. The tests include checks for subrange variables or array indices lying outside their admissible bounds; functions returning no result; and CASE statements with an ELSE clause, where none of the other alternatives apply. If it is turned off, none of this test code is generated. Like overflows, turning off the test option can speed up a program and make it more compact. As a rule, however, the resulting safety loss does not make up for this advantage until a program is completely debugged.

$U The Upper=Lower Case Option

☐ $U+ does not distinguish between uppercase and lowercase letters.

☐ $U- does distinguish between them.

If this option is on, the Turbo Modula-2 compiler treats an uppercase letter exactly like its lowercase equivalent. If it is off, an uppercase letter and its lowercase equivalent are considered to be different characters. For example:

```
VAR n,N: CARDINAL;
BEGIN
  if n < N THEN ....
```

If this program fragment is compiled with $U turned on, the **VAR** declaration would be illegal because *n* and *N* would denote the same identifier.

Compilation with $U turned off would cause no problems in this line. However, the third line of the program fragment would then be illegal, since *if* would be considered an identifier rather than the reserved word IF. The Modula-2 standard prescribes that lowercase and uppercase letters are different; that is, that $U should be turned off. Programmers conscious of the standard can reset this option permanently in the Options menu by pressing ⬜ s ⬜ .

$X The Extension Option

☐ $X+ allows Turbo Modula-2 extensions.

☐ $does not allow them.

Turbo Modula-2 supports several programming constructs not included in the standard, including the following:

☐ Pascal's *READ* and *WRITE* statements can be used for readable input and output.

☐ Exceptions similar to Ada's are supported.

☐ Two arrays with elements of type CHAR (strings) can be assigned even if they are not of the same type.

☐ Any two arrays of CHAR can be compared.

☐ Open arrays can be multidimensional.

These extensions are supported only if $X is turned on. If this option is turned off, nonstandard constructs are flagged by the compiler as errors.

If a programmer intends to use only standard Modula-2, he can turn off $X permanently in the options menu and save it.

$N The Native Code Generation Option

☐ *N ON* generates native code for the Z80.

☐ *N OFF* generates M-code.

Turbo Modula-2 can also generate native Z80 machine code that will execute faster than the M-code normally generated. In particular, floating-point math operations and I/O will probably not benefit from this option. The generated native code will be about three times as large as the equivalent M-code, and may run more than ten times faster.

Since this option applies only to compilation units, it must be specified from the Options menu; using (* $N+ *) in the text will have no effect.

The Linker

Turbo Modula-2's linker combines several separately compiled modules in a single file. A typical large program system consists of a main module and several other support modules. Before such a program is executed, the main module and all needed support modules are loaded into main memory.

How these modules are loaded depends on the type of output the linker produces. There are two output formats produced by the linker: (1) an .MCD file that contains object code and (2) a .COM file that contains executable code.

The purpose of linking many object files into one object .MCD file is to speed up loading time by eliminating disk searching and the time it takes to open more than one file.

If a program's support modules exist as separate files, they are automatically brought into main memory. Every module involved in the program must be found on disk before it can be read in. The Turbo Modula- 2 system optimizes the search for the necessary files, achieving relatively short loading times. Nevertheless, some gain in speed can be attained if all modules, or even only part of them, are contained in a single file.

The second function of the linker is to produce an executable .COM file by combining the main module, all support modules, and a runtime system.

The linker takes several object files with the extension .MCD and combines them into a single file. Its output consists of either another .MCD file or a .COM file. While .MCD files are run from inside the Turbo Modula-2 environment, .COM files are executed directly form the operating system. For example, if the file DEMO exists with both extensions .COM and .MCD, you would use the eXecute command to run DEMO.COM and the Run command to execute DEMO.MCD. In addition, you could run the DEMO.COM file from your operating system prompt like any other program.

You start the linker by pressing ⌊ L ⌋ at the main menu. You are then prompted for the program's main module and the name of the linker's output, like so:

```
Link main module:
Output file:
```

If any of these arguments do not contain an extension, the default extension .MCD is supplied. If the output file name has an extension, it must be either .COM or .MCD. You must specify the extension .COM if you wish to produce an executable file.

After that, you are asked:

```
Include all needed modules (Y/N)?
```

Normally, you would press ⌊ Y ⌋ (for yes), in which case the linker does the rest of the work for you. It searches for all the modules needed on all the drives that are present in the search path. These modules are combined into one file, which is then written to disk.

If you have several versions of the same object file on different disks and want to pick out one of these to be included in the linked file, you would press ⬚N⬚ (for no) and be prompted like this:

```
Include modules:
```

You can then enter all modules you wish to include on one line. The .MCD extension can be omitted; it will be supplied as a default. If you want to include more modules than would fit on a single line, terminate the line by pressing the space bar and a ⬚+⬚ , The »Include modules:« prompt is then repeated:

```
Include modules: TEXTS FILES B:MATHLIB +
Include modules: B:WINDOWS B:GRAPHICS
```

The linker then displays a list of all linked modules (and produces a linked object file), all modules that are referred to but not linked, and all modules that are missing. An executable file is produced only if all modules are accounted for and unresolved.

Note that unresolved references are perfectly legal if the output file is of type .MCD. The loader will bring all modules that are referred to but not linked automatically into memory during a Run command.

It would be a waste of memory to link one of the modules *Tests*, *Files*, *ComLine*, *Convert*, *Loader*, or *Doubles* into a .MCD file, since they already reside in memory. If you let the linker search library modules automatically, all these modules are left out. When linking a .COM file, the situation changes. When a .COM file is started directly from CP/M, the flexible Turbo Modula-2 loader is not available. Therefore, all modules the main program imports from *must* be included in the linked file.

The included files usually consist of a single object module. It is possible, however, that they are linked and include several modules. In that case, all modules are transferred to the new linked file.

Linking with Overlays

The linker can produce .COM files with overlays. The scheme is more flexible than the one employed in Turbo Pascal, but a bit more difficult. Every overlay consists of a full module; you need not change the module's source code to make it an overlay.

To link a file with overlays, the user must answer the initial question whether to include all needed modules with »No.« The user must then specify all modules that should be linked, but not overlaid. These modules must be entered on a single line or on several lines terminated by $\boxed{+}$ characters. If a module is not preceded by a drive code, its .MCD file is first searched in all libraries of the search path, then on all drives of the system. A name with a drive specification will cause the module to be searched only on the specified drive. After all these modules are processed, the user is asked whether overlays should be used. Typing *Y* will cause the following prompt to appear:

```
Include overlay module:
Map onto linked module:
```

First, the name of the module to be overlaid is entered. Next, the module whose space is to be used when it is loaded is entered. Now both modules will be linked to occupy the same memory segment. The prompts will be repeated until the user answers the first question with a single carriage return.

Note: The second module must already have been included n the linked file, otherwise its space will not be available to the overlay module at runtime. Also, more than one overlay module may be mapped onto the same linked module.

As an example, suppose you want to link the module *MAIN* with three overlay modules (*OVER1*, *OVER2*, and *OVER3*,) and one resident module (*SUPPORT*) into the file MYPROG.COM. The dialogue with the linker would look like this:

```
>L ·

Link main module  : MAIN
Output file       : MYPROG.COM

Include all needed modules (Y/N)? N

Include modules   : SUPPORT OVER1

Linking MAIN
Linking SUPPORT
Linking OVER1

Use overlays (Y/N)? Y
```

```
Include overlay module          : OVER2
Map onto linked module          : OVER1
Linking OVER2

Include overlay module          : OVER3
Map onto linked module          : OVER1
Linking OVER3

Include overlay module          : (RETURN)
```

Figure 10-1. Diagram of Linking Overlay Files

Low

Main			← Always present
Support			← Always present
Over1	Over2	Over3	← Swapped

High

Memory   Initial Load   1st Overlay   2nd Overlay
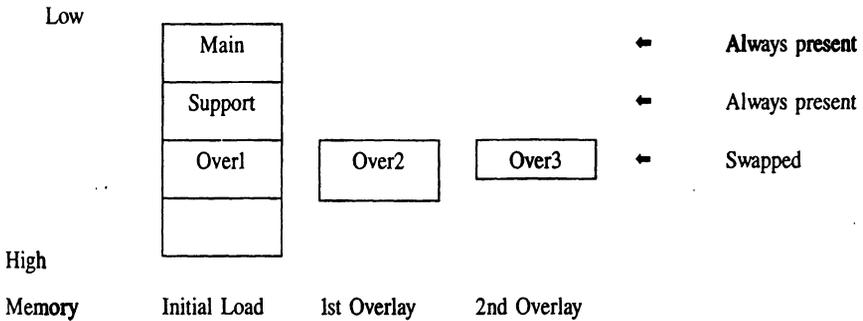
Note that the linker accepts input redirection. Complicated schemes for linking overlay files can be edited once in a file and are then fed to the linker as input. A file to accomplish the task in Figure 10-1 could look like this:

```
SUPPORT         OVER1           (modules to be included)
OVER2           OVER1           (first overlay)
OVER3           OVER1           (second overlay)
```

If the file had the name **LINKPROG** and you started the linker with

> L

```
Link main module  : MAIN <LINKPROG
Output file        : MYPROG.COM
```

you would you would only have to answer two (Y/N) questions; entering all modules to be included or used as overlays would be unnecessary. Of course, input such as this:

```
>L
```

```
Link main module  : MAIN MYPROG.COM <LINKPROG
```

or even

```
L MAIN MYPROG.COM <LINKPROG
```

is also legal.

A few restrictions must be made for overlay modules. First, an overlay module may not contain an initialization part (if so, the linker will issue an error message). Second, two overlay modules residing in the same segment should not import from each other. If the import concerns only constants, types, and variables, this is harmless. However, calling a procedure in another overlay module in the same segment will lead to trouble. In this case the linker issues only a warning message.

A program with overlays will be linked into two files with the extensions .COM and .OVR. The .OVR files contain all overlays needed for the program. They must be online when the program is started.

Every time a procedure in an overlay module is called, the module is brought into memory (provided it is not already there), overwriting whatever was in the segment previously. The same thing happens if a procedure returns to its caller and the caller has moved out of memory in the meantime. The calls to procedures in overlay modules do not need to follow a tree-like structure; they can proceed circularly. By specifying the mapping scheme, you can minimize overlay disk feeds and not bother with address computations, which is done by the linker.

The linker writes all overlay modules to disk as they are processed. Therefore, the length of all overlays is not restricted by memory size (the .OVR file can have a length of up to 256 Kbytes). However, the linker buffers modules that are not

overlays in main memory. Therefore, the size of the .COM file is restricted by the memory available to the linker.

Linking the Linker

If you experience an OUTOFMEMORY error during linking, you can increase the available memory by making the linker smaller. The linker consists of the file LINK.MCD, which is loaded on top of the Turbo Modula-2 system. The Turbo system contains several modules that are unnecessary for the linker's operation and that may be taking up valuable memory space (the editor, for example). These modules can be removed by linking the linker itself. To do so, simply give the f(ving commands:

```
>L

Link main module  : LINK
Output file       : LINK.COM
```

  and answer the question

```
Link all needed modules (Y/N)?
```

with [Y] . The result is a file LINK.COM. If the linker is now started from CP/M with the command A > LINK, the memory available for its operation is increased. If this is still not sufficient, you can use overlays for the linker, producing the files LINK.COM and LINK.OVR. You can do this by editing a file called LINKLINK or something similar that consists of the lines

```
TEXTS   TERMINAL COMLINE LOADER

(  ES   LOADER
 CONVERT LOADER
```

  Next, start the linker with this file as input, and answer »No« to whether to include all modules. The modules *Files* and *Convert*, which are needed by the linker, are both mapped as overlays onto the segment of the module *Loader*. They no longer use main memory for themselves. Therefore, the space needed by these modules is free to link even longer files.

Notice that the standard, but less efficient, »overlay scheme« that uses the procedure *Call* in *Loader* is also available.

Version Control

To prevent a program from running erroneous versions of separately compiled modules, a version number is given to each definition module. There are several ways to generate version numbers. Turbo Modula-2 determines a version number by calculating a checksum of the output produced when compiling a definition module. The number is kept in the symbol (.SYM) files as well as code (.MCD) files that are imported from each library module.

This method of generating numbers is handy since recompiling an unchanged module does not produce a new version number and requires unnecessary recompilation of dependent modules. With this process, you can even add comments to the definition module and then recompile it, which will have no effect on the version number generated.

Version control is the act of checking version numbers for consistency. There are three distinct points where version control is performed: compile time, link time, and runtime.

The compiler checks .SYM files for version numbers. Version conflicts occur when a library module's definition and implementation modules both import from another library module at different times (that is, different definitions, or version numbers, of the imported module are used by the definition and implementation modules). This situation is resolved by recompiling the module that was first compiled of the two importing modules.

If a library definition and implementation module both import from the same library module, then the compiler checks to make sure the definition (version number) has not changed between compiling the definition module and compiling the implementation module.

The linker and loader check .MCD files for version numbers. Version conflicts occur when trying to link or load a module that imports from a library module that has had its definition and implementation modules changed (and recompiled). The importing module must be recompiled to resolve the conflict.

Conflicts can also occur when a module imports from two library modules that in turn import from another library module at different times (that is, the lowest level module being imported from has changed its definition). To resolve this conflict, you must recompile the first compiled of the two library implementation modules.

The linker compares the version numbers of all the import modules. Thus, if more than one module imports a particular library module, all the version numbers for that module must match. In that case, the version numbers are found in the importing .MCD files.

(nce the loader does dynamic linking, it can also find the same kind of version conflicts the linker finds (in case linking was not done). The loader can also detect version conflicts when loading overlays, which occurs when an overlay module imports from its caller and the caller's definition subsequently changes. This conflict is resolved by recompiling the overlay module.

Utilities

The following utility programs will add to your collection of programming tools. The first utility converts preexisting object files to a form usable by the Turbo Modula-2 system. The second tool allows you to evaluate the performance of your programs.

Linking Microsoft Relocatable Files

In some applications, it is desirable to use existing libraries with Modula-2. On Z80-CP/M systems, these libraries exist mostly in Microsoft relocatable format. Object files in this format are generated by such translators as the Microsoft (ro assembler M80, the Microsoft FORTRAN compiler F80, Pascal MT/+, and others.

Turbo Modula-2 offers a utility to link programs in the Microsoft .REL format with Modula-2 programs. Only library procedures can be called from Modula-2, but constants and variables cannot be accessed. Nor can, say, a FORTRAN subroutine call a Modula-2 procedure.

Linkage is accomplished by a special program named REL. It takes as input a compiled definition module (with the extension .SYM) and one or more

relocatable files (with the default extension .REL). Its output is a normal
Modula-2 .MCD file.

In the Modula-2 definition module, all procedures to be called from Modula-2
must be declared with their parameter lists. The names of the procedures must
appear as entry points in the relocatable files. Parameters are normally passed on
the stack. The definition module could look like this:

DEFINITION MODULE Z80Stuff;

FROM SYSTEM **IMPORT** ADDRESS, BYTE;

PROCEDURE MoveL(source,dest : ADDRESS; l : **CARDINAL**);

<other procedure declarations>

END Z80Stuff.

The input file (called, for example, Z80STUFF.MAC) to the M80 macro
assembler consists of the following:

```
        .Z80
        PUBLIC MOVEL        <more identifiers>
MOVEL:POP        HL;        return address
        POP        BC;        length
        POP        DE;        source
   ·    EX         (SP),HL;  exchange return address and destination
        EX         DE,HL;    exchange source and destination
        LD         A,B
        OR         A,C
        RET        Z
        LDIR
        RET            ·
     <more code>
```

You can then translate these files by invoking the Turbo Modula-2 compiler and
M80. Next, REL is executed by entering REL after the run-file prompt; for exam-
ple:

Run **MCD-file: REL Z80STUFF Z80 STUFF**

The first parameter given is the name of the definition module. No extension may be given; the extension .SYM is always used. The second parameter given is the name of the relocatable file. If no extension is given, .REL is assumed.

REL produces an M-code file with the name of the definition module and the extension .MCD. This file may be used normally by a program, for example:

```
MODULE demo;

FROM Z80Stuff IMPORT MoveL;

BEGIN
  s := ...; d := ...; l := ...;
  MoveL(s,d,l);
END demo.
```

The Microsoft FORTRAN convention for passing parameters, however, is different from the scheme explained previously. The difference between these two schemes when linking (for example, FORTRAN routines or libraries) is that parameters are not passed on the stack but in registers. REL offers the command line switch »F/« (/FORTRAN) to accommodate this. The FORTRAN switch may be given anywhere on the command line--its effect is global. The FORTRAN parameter-passing scheme is used for all REL files on the command line.

REL also has a search switch (»S/«) that permits the user to search libraries and link only those modules that are actually needed. Unlike the FORTRAN switch, the search switch has only a local effect. It is given after a file name and causes only that file to be searched.

Here is a FORTRAN example:

```
FUNCTION IDIV(I,J)
IDIV = I/J
RETURN
END
```

And the Modula-2 definition module might look like this:

```
DEFINITION MODULE divide;

PROCEDURE idiv( i, j : INTEGER ) : INTEGER;

END divide;
```

After compiling both programs, link them by entering the following:

```
Run MCD-file: REL DIVIDE IDIV /F FORLIB /S
```

(We call our FORTRAN source file IDIV.FOR.) Once again, you can use function procedure *IDIV* any way that it's allowed in Modula-2.

Warnings: Parameters of structured types are always passed as **VAR** parameters to FORTRAN, even if a value parameter is specified in the definition module (such parameters are not copied).

FORTRAN routines are not reentrant, nor is the scheme used by REL to pass parameters. Since FORTRAN routines cannot currently call Modula-2 procedures, there are no problems with recursive procedures. Programs using interrupts, however, may have trouble.

Profile

Profile is a utility included with Turbo Modula-2 that allows you to estimate the time spent in various procedures of your program.

If the profiler is invoked with the command

```
>R
Run MCD-file:PROFILE
```

the main menu prompt > will reappear on the screen as if nothing happened. The profiler actually loads a second version of the shell. You can then run the program to be profiled by giving another **Run** command. When this program has executed, the profiler outputs a detailed list of all procedures and modules and the percentage of time spent in them.

The profiler can only monitor M-code procedures; native code procedures will not appear in its output. In fact, the profiler simply counts M-code operations.

Since all M-code instructions do not execute in the same amount of time, the measurement is slightly inaccurate; but it should be sufficient to find out which modules and procedures are time-critical.

With this information you can now proceed to compile modules into native code or to write extremely time-critical procedures in assembly.

Since output of the profiler is quite long, output redirection is normally used. For example:

Run MCD-file: **PROFILE** >**MYPROG.PRO.**

(ie following example monitors MYPROG.MCD performance using PRO-FILE and then routes the report to file MYPROG.PRO.

```
>R
Run MCD-file: PROFILE >MYPROG.PRO

>R
Run MCD-file: MYPROG.PRO
```

Chapter 11
The Standard Library

Some constructs needed for programming are not covered by the language Modula-2. Instead, these operations are performed by a set of predefined modules that cover input and output, mathematical functions, string handling, and storage and process management. A good part of the modules deal with input and output, and thus form a hierarchy (see Figure 11-1).

Figure 11-1. Library Module Hierarchy
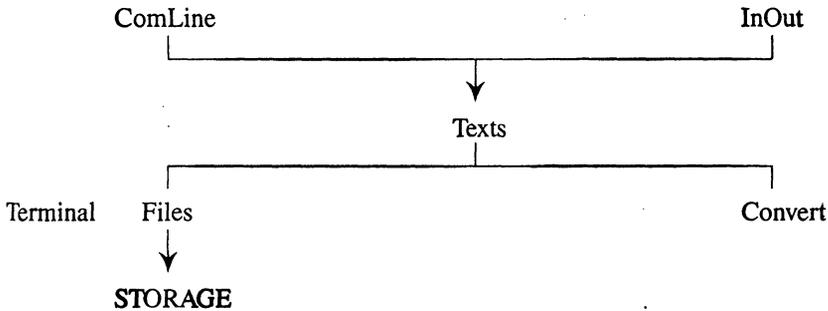
The arrows in Figure 11-1 denote dependencies on other modules. In general, modules at the top of this drawing can be regarded as high-level modules, and those at the bottom as low-level modules. The module *Terminal* is an independent low-level module; that is, it does not depend on other modules and no module depends on it. The module *STORAGE* does not deal with input and output, but manages heap memory for control blocks and buffers.

The modules *MathLib*, *LongMath*, *Strings*, and *Doubles* comprise what is called a utility library. They provide facilities for manipulating objects.

The low-level modules *Loader*, *SYSTEM*, *STORAGE*, and *Processes* are system-dependent. *Loader* and *Processes* are higher level in that the programs that use them can become portable with minor changes. The modules *SYSTEM* and *STORAGE* constitute a special case: They enclose system-dependent primitives

needed for low-level programming. The compiler translates all operations imported from them directly into code. *SYSTEM* and *STORAGE* have neither a definition nor an implementation module (nor compiled counterparts), since technically they are not library modules. Because the compiler knows about them in advance, the use of the facilities from these two modules forces the programmer to explicitly specify (in an import list) the use of system-dependent, low-level constructs. Figure 11-2 displays the interdependencies of these modules.

Figure 11-2. Interdependencies of System-Dependent, Low-Level Modules

```
   Loader                                              Processes
     |_____|
         _____|_____
        |                                              |
      SYSTEM                                         STORAGE
```
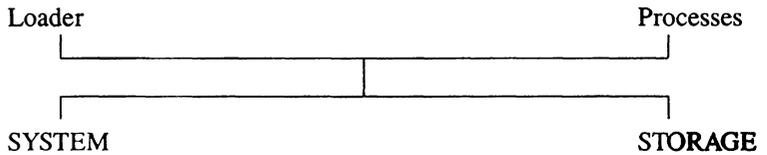
We can see that the Modula-2 library can be divided into three sections: input/output modules, utility modules, and system and low-level modules. In the next section we will briefly discuss the function of each library module within these three module types; a detailed description of each module will follow.

Overview of Input and Output Modules

Most programs involve input and output of data; for example, entering characters at the keyboard or writing program output to a disk file. Although these processes are different, Turbo Modula-2 treats them in a similar manner.

ComLine Module. Gives a program access to command-line parameters. The standard texts *input* and *output* can be redirected from the standard devices with this module.

Files Module. Procedures providing low-level access to disk files, including random access as well as sequential access. This module is used by the higher-level modules *InOut* and *Texts* to interface with the operating system.

InOut Module. Wirth's standard procedures for high-level input and output.

Terminal Module. Procedures to read the keyboard and write to the console with

highlighting and cursor control. Includes a procedure to clear the screen and one to check for characters in the keyboard buffer or console.

Texts **Module**. Procedures to read and write text (character streams) to and from external files and devices. Includes provisions for installable input/output device drivers.

Overview of Utility Modules

Utility modules provide string-handling procedures, **mathematical functions, and** (version from numbers to strings and back again.

Convert **Module**. Contains procedures to convert strings into numbers and numbers into strings; handles CARDINAL, INTEGER, and REAL numbers.

Doubles **Module**. Contains procedures to convert strings into double- precision numbers and double-precision numbers into strings; handles LONGINT and LONGREAL numbers.

LongMath **Module**. Contains several mathematical functions for double- precision numbers, including the common transcendental functions and the natural power functions.

MathLib **Module**. Contains several mathematical functions for single- precision numbers, including the common transcendental functions and the natural power functions; this module also includes a random-number generator.

Strings **Module**. Contains procedures for string handling, including deletion, insertion, and copying of substrings.

(

Overview of System and Low-Level Modules

These modules provide an interface to the operating system, memory, and ports. They allow the programmer to develop system routines for memory management, multiprogramming, and other low-level constructs.

Processes **Module**. Enables the formulation of loosely **coupled processes that are**

implemented as coroutines. You can use this module to simulate concurrent programming.

STORAGE Module. Handles memory managment for allocation and deallocation of dynamic variables; also keeps track of available memory.

SYSTEM Module. Contains system-dependent procedures that work with addresses, ports, operating systems and assembly language calls, and facilities for low-level process control.

Loader Module. Handles loading and execution of compiled object modules, enabling the splitting of large programs into smaller parts; the less commonly used parts are loaded from a disk as overlays. This allows the program to manage its own overlays rather than having it done by the runtime system, which is how the overlays produced by the linker are handled.

Details of the Module Library

Rather than defining specific input/output and system-specific statements in the language, these operations are provided in a hierarchy of modules. The modular structure has the effect (and advantage) of hiding computer- specific details, while also providing a means of including new machine- dependent routines when required. What follows is a full description of the services offered by the predefined modules.

Input and Output

Input and output remain one of the biggest problems in high-level language design. This is not surprising, since these operations have to deal with a large number of peripheral devices whose characteristics cannot be contained in a few simple abstractions. In every input/output system, simplicity and brevity of details are in conflict with efficient and flexible use of peripheral devices.

In Modula-2, this dilemma has been circumvented by not including any input/output primitives in the language at all. Instead, all communication with external devices is provided by a set of modules supplied with the system. And since these primitives are not part of the language (or the compiler), they can be freely supplemented according to the user's needs.

However, such a library of input/output functions has an intrinsic disadvantage: It leads to a large set of different primitives that must be remembered by the user. This is because Modula-2 does not allow procedure names to be overloaded (which is when more than one procedure has the same name, but is differentiated by the types of its parameters). Instead, there must be a separate routine for processing each type, even if each routine is essentially the same. For example, you must have one routine to write a character to the screen, another to write an INTEGER, another to write a REAL, and so on.

In addition, Modula-2 does not allow procedures to have a variable number of parameters. Thus, when you wish to write three strings to the screen, you must cal *riteString* three separate times rather than make one call that passes three parameters.

On the other hand, the advantage of having one call for one parameter for each type means you need not guess at the type of the parameter or search out its definition--it is made explicit directly in the code.

In our experience, it is often more difficult to remember the correct usage of the various input/output operations than to deal with the syntax of Modula-2. This is clearly an undesirable situation, particularly for the novice. In Turbo Modula-2, we opted for the best of both worlds: We have included all input/output modules postulated by the standard and allowed the use of general-purpose read and write statements as well, namely *READ*, *READLN*, *WRITE*, and *WRITELN*.

The form of these read and write statements is similar to that of procedures, but they accept varying numbers of parameters, where each may be a different type. This type of procedure is seen with other standard procedures such as *INC* and *DEC*, which accept any scalar type and an optional count parameter.

(*1D*, *READLN*, *WRITE*, and *WRITELN* are not declared in any standard module; instead, they must be thought of as extensions to the language Modula-2. The compiler translates these statements into corresponding procedure calls to the module *Texts*. This occurs transparently since you can supply your own drivers to the *Texts* module. You lose nothing in terms of efficiency or flexibility, but gain a lot in simplicity of input and output.

The input and output operations form a hierarchy: At its top are the general *READ* and *WRITE* statements, and at its bottom the underlying operating system. Between the extremes there is a sequence of modules incorporating primitives

with varying degrees of abstraction. Before we delve into the explanation of these modules, let's work out some of the characteristics of most input and output operations.

Input and output usually consists of the reading and writing of a sequence of data items. The process is often strictly sequential; for many peripheral devices, this is the only way to do input or output. A line printer, for example, prints one character after the other, and input from the keyboard consists of a sequence of characters as well. Even if some peripheral device allows random access to data (for example, to a disk drive or a video screen where it is possible to write characters at any position), the sequential operation is often the most convenient. A structure that consists of a sequence of data items is called a *stream*. (This is a purely abstract notion and is not part of the Modula-2 language.)

Streams
As you type at the keyboard, picture a stream of data flowing into the computer from the keyboard. The data output to the disk file is another stream disappearing from the program and being stored on the disk.

A stream, as discussed here, is a sequence of data that is either input from or output to an external medium. The external medium may be either a disk file or a logical device (the keyboard, video screen, or printer); both forms are treated in the same way.

In general, the predefined input/output procedures allow you to link either an input or an output stream to a medium, pass data down the stream, and then sever the connection upon completion.

The following lists the main characteristics of a stream:

☐ The number of elements of a stream, called the *length* of a stream, can vary.

☐ Only one element of a stream is visible at any one time. Reading starts at the beginning of a stream. After a read operation, the next element in the stream becomes visible.

☐ A stream can be modified only by appending elements at its end. Appending an element is called *writing*.

☐ A stream has a mode that can be either *read* or *write*. The mode is specified before any reading or writing is carried out and cannot be changed thereafter.

Notice that none of these characteristics specify the type of the elements of a stream. They can be of any type; however, it is useful to make a broad classification. We generally distinguish between legible and illegible input and output. A legible stream consists of elements of type CHAR, which is also called a *text*. Texts provide the means of communication between the user and the computer, where illegible streams store data for further processing by the computer.

There are several input/output modules that differ in both flexibility and type of st   n handled. For example, the module *Terminal* deals only with the screen and keyboard text streams, while the module *Files* can handle common sequential I/O or random access files that are not streams at all.

A special stream type is the *text stream*, which contains only objects of type CHAR; for example, a stream of characters input from the keyboard or output to the screen or a printer. Text streams are a special case because they are formatted with EOT and EOL characters. (An EOT character marks the end of the text stream, while an EOL character terminates a line.)

Logical Devices
We have just discussed where data originates from and how a program may abstract it; where the data ends up is also relevant. In general, operating systems permit you to write to either a disk file or to *logical devices*. A logical device acts as a code word, specifying to the operating system which path leads to which device, whether it is a printer, a terminal, or another device. Which logical devices you have available will depend on your operating system and your hardware. For example, the CP/M operating system recognizes the following code names:

CON: The video screen for output, the keyboard for input

LST: The printer (output only)

RDR: An additional input interface

PUN: An additional output interface

Note: Your configuration determines which devices can be accessed by RDR: a by PUN:.

The Texts Module

Texts provides the means to input and output a legible stream to and fr peripheral devices. The module *Texts* uses the following characters to format t streams:

☐ EOL (=36C, 30 decimal) denotes the end of a line.

☐ EOT (=32C, 26 decimal) denotes the end of the text.

Note that EOT is not defined as a constant in *Texts*.

Peripheral devices usually employ either a single carriage return (15C, decimal) or a carriage-return/line-feed (12C, 10 decimal) pair to denote line er The conversion from these control codes to the EOL character and reverse is d automatically by the module *Texts*.

At the center of this module is the type *TEXT*. It describes any legible, sequ tial input and output. This type is declared as a subrange:

```
TYPE TEXT: [1..16];
```

Variables of type *TEXT* serve as indices to various internal tables of this mod The tables themselves are not exported; they remain hidden to guarantee the tegrity of the system.

Standard Text Streams
Texts exports three standard predefined texts that are declared as

```
input, output, console: TEXT;
```

The frequently used text streams are offered as predefined standard texts alre linked to commonly used mediums. These standard texts do not require the r mal linking and severing operations mentioned earlier. For example:

☐ *Input* is the primary input stream usually assigned to the keyboard, but be redirected to another medium, such as a card reader or other input dev

☐ *Output* is the primary output stream to the video console, but can be redirected to another medium, such as a printer or a file.

☐ *Console* is the output stream to the video console; it cannot be redirected. It is used to print error messages to the screen when the normal output stream is redirected elsewhere.

Texts Module Specification

```
DEFINITION MODULE Texts;
 FROM Files IMPORT FILE;
 TYPE TEXT = [1..16];

 VAR input,output,console: TEXT;

 PROCEDURE ReadChar     (t: TEXT;  VAR ch: CHAR);
 PROCEDURE ReadString   (t: TEXT;  VAR s:  ARRAY OF CHAR);
 PROCEDURE ReadInt      (t: TEXT;  VAR i:  INTEGER);
 PROCEDURE ReadCard     (t: TEXT;  VAR c:  CARDINAL);
 PROCEDURE ReadLong     (t: TEXT;  VAR l:  LONGINT);
 PROCEDURE ReadReal     (t: TEXT;  VAR r:  REAL);
 PROCEDURE ReadLn       (t: TEXT);

 PROCEDURE WriteChar    (t: TEXT;  ch:     CHAR);
 PROCEDURE WriteString  (t: TEXT;  s:      ARRAY OF CHAR);
 PROCEDURE WriteInt     (t: TEXT;  i:      INTEGER; n: CARDINAL);
 PROCEDURE WriteCard    (t: TEXT;  c, n:   CARDINAL);
 PROCEDURE WriteLong    (t: TEXT;  l:      LONGINT; n: CARDINAL);
 PROCEDURE WriteReal    (t: TEXT;  r:      REAL; n: CARDINAL;
                                           digits: INTEGER);

 PROCEDURE WriteLn      (t: TEXT);

 PROCEDURE ReadLine     (t: TEXT; VAR s: ARRAY OF CHAR);
 PROCEDURE ReadAgain    (t: TEXT);

 PROCEDURE Done         (t: TEXT): BOOLEAN;
 PROCEDURE EOLN         (t: TEXT): BOOLEAN;
 PROCEDURE EOT          (t: TEXT): BOOLEAN;
 PROCEDURE Col          (t: TEXT): CARDINAL;
 PROCEDURE SetCol       (t: TEXT;  column: CARDINAL);
 PROCEDURE TextFile     (t: TEXT): FILE;
 PROCEDURE OpenText     (VAR t: TEXT; name: ARRAY OF CHAR):
                                           BOOLEAN;
 PROCEDURE CreateText   (VAR t: TEXT;  name: ARRAY OF CHAR);
 PROCEDURE CloseText    (VAR t: TEXT);
```

```
CONST EOL=36C;

TYPE TextDriver = PROCEDURE(TEXT, VAR CHAR);

PROCEDURE ConnectDriver(VAR t: TEXT; p: TextDriver);

PROCEDURE Init;    (* used only by system *)

VAR haltOnControlC : BOOLEAN;    (* TRUE by default *)

EXCEPTION TextNotOpen, TooManyTexts;

END Texts.
```

The following example shows the use of the *Texts* module. It reads ten from the keyboard and writes them to a disk file.

```
MODULE DataSave;
FROM Texts IMPORT TEXT, input, Done, CreateText, CloseText,
                  ReadCard, WriteCard;
VAR
 DataFile: TEXT;
 Value,Count: CARDINAL;
BEGIN
 (* Create disk file and link text stream *)
 CreateText(DataFile, 'DATA.DTA');
 (* Read 10 values, save on disk *)
 FOR Count:= 1 TO 10 DO
  ReadCard(Input,Value);
  IF Done(Input) THEN
   WriteCard(DataFile,Value,16)
  END
 END; (* Close disk file *)
 CloseText(DataFile)
END DataSave.
```

Note that since *input* is a standard type *TEXT* declared in module *Texts*, it does not require opening or closing.

Stopping the Program During Input and Output
The following variable lets the programmer determine when a program can be interrupted while using the *Texts* module:

```
VAR haltOnControlC: BOOLEAN;
```

The variable *haltOnControlC* is exported by *Texts* and can be specified by the user. If *haltOnControlC* is TRUE (the default), then the program will halt on either of the following conditions:

☐ If you enter a [CTRL] [C] as the first character of an input line;

☐ If you enter a [CTRL] [S] to stop the output to the screen, followed by a [CTRL] [C] . (Note that *WriteLn* is the only output statement that checks for keyboard input; thus output to the screen can only be stopped if it is used.)

If *haltOnControlC* is set to FALSE by the programmer, then the user may only interrupt by program control.

Opening, Creating, and Closing a Text
Before data can be written or read, a connection must be established between a variable of type *TEXT* and some external medium, such as a disk file or a line printer. This is accomplished with the three procedures that follow.

OpenText (VAR t: TEXT; Name: ARRAY OF CHAR): BOOLEAN;

Associates an internal text value assigned to *t* with an existing external file iden-
tif   by the string *Name*. The variable *t* can be used only for input. The parameter *Name* must be the name of a disk file or the code name for one of the operating system's logical input devices, such as CON: or RDR:. In the case of a disk file, a new internal file is created and connected to the existing external file using the procedure *Open* in module *Files*. When an input device is specified, the variable *t* is linked to the external device. In either case, *OpenText* returns TRUE if the text or device is opened successfully; FALSE if the file or device is not found.

CreateText(VAR t: TEXT; Name: ARRAY OF CHAR);

Establishes a new external text specified by *Name* and connects it to the text variable *t*. The text *t* can be used only for writing. Usually, the parameter *Name* identifies a new disk file, which is then created by the procedure *Create* in module *Files*. If the specified name denotes one of the logical output device names, such as CON: or LST:, then the text *t* is connected to that particular device.

After input or output has been completed on a text, the association with an external medium can be severed by use of the procedure *CloseText*. It is always recommended to close a text when processing is complete; otherwise, output files m̜   ̜ot be updated and input files will not have control blocks disposed of pro-
perıy.

CloseText(VAR t: TEXT);

Severs the connection between the given internal text *t* and its associated external text. If the external medium is a disk file, this file is closed using *Close* in module *Files*. When *t* is used for output, the EOT character is appended to the text before it is closed.

A special convention is used for the standard texts *input*, *output*, and *console*. These texts are always open and connected by default to the terminal (imagine them opened with the name CON:). Thus, when reading from the keyboard or writing to the video screen, *input*, *output*, or *console* do not need to be opened or created. When *OpenText* is applied to *input*, the standard input text is redirected to the external medium identified by the parameter Name. Likewise, *CreateText(output,name)* redirects standard output to the medium given by *name*. Closing *input* or *output* will establish the default connection again. Following are some examples.

OpenText(t,"CON:");

Establishes *t* as an input text with data from the keyboard.

OpenText(input,"RDR:");

Opens the standard text *input* with a different device. This time the standard text *input* is linked to the RDR: device instead of the keyboard; that is, *input* is *redirected*.

OpenText(InText,"B:INDATA.DAT");

Opens existing file INDATA.DAT on drive B for input. Subsequent input from text *InText* is taken from this external file.

CreateText(Printer,"LST:");

Connects the internal output text *Printer* with the external printer device.

CreateText(OutText,"OUTDATA.DAT");

Creates a new output file on the currently logged drive. Output statements specifying *OutText* write to this file.

CreateText(output,"PROTOCOL");

Redirects the standard output text. Standard output is no longer displayed on the screen; it is written to a file PROTOCOL that is created by this call.

CloseText(OutText);

Closes the output file B:OUTDATA.DAT, thereby establishing this file in the directory of drive B.

CloseText(input)

Returns the standard *input* **from the RDR: device to the keyboard.**

CloseText(output)

Closes file **PROTOCOL and returns output to the video screen.**

Renaming, Deleting, and Other File Operations
In addition to the previously described functions, you may need to rename, delete, or perform other operations on text files. This can be done via the module *Files* described on page #. All of the services offered in *Files* can be performed on *Texts* by using the function *TextFile*.

TextFile(VAR t: TEXT): FILE;

Returns the value of *FILE* variable associated with the *TEXT t*. If this text has not yet been opened, the value NIL is returned. Here are some examples.

Rename(TextFile(t),'NEWNAME.DAT')

Renames the file associated with the text *t* to 'NEWNAME.DAT,' and also closes it.

Size := FileSize (TextFile(t))

Puts the size of *t* in bytes in the **variable** *Size* **of type LONGINT.**

Flush(TextFile(t))

Flushes the *t* file buffer.

Reading and Writing
Once a text has been opened, data can be read from it. Reading generally involves two operations: (1) the transmission of one or several characters from the external

medium into the computer's memory, and (2) an interpretation of these characters. If, for example, a variable of type INTEGER is read, the decimal representation on the external device must be translated into the binary form used internally by the computer. The following set of procedures are applicable when inputting formatted data.

ReadChar(t: TEXT; VAR ch: CHAR);

Reads a single character from text *t* into **ch.**

ReadString(t: TEXT; VAR s: ARRAY OF CHAR);

Reads a string; that is, a sequence of characters not containing blanks or control characters. Any leading blanks are skipped. The string *s* receives as many legal characters as will fit into it. Input is terminated by a blank, an EOL character, or any other control code (a character with a decimal value less than 32). Note that strings cannot be separated by commas.

   The next procedures can be used for numeric input.

ReadInt(t: TEXT; VAR i: INTEGER);

Reads a string and converts it to an integer. Again, leading blanks are ignored. Valid input is any sequence of digits 0 to 9, possibly preceded by an arithmetic sign (+ or -). The integer must be in the range *MIN(INTEGER)* to *MAX(INTEGER)*.

ReadCard(t: TEXT; VAR c: CARDINAL);

Reads a string and converts it to CARDINAL. Leading blanks are ignored and no sign is allowed. The CARDINAL must be in the range from 0 to *MAX(CARDINAL)*.

ReadReal (t: TEXT; VAR r: REAL);

Reads a string and converts it to a real number. Leading blanks are ignored. The string may contain a decimal point, a mantissa, and an exponent, but these are optional.

ReadLong(t: TEXT; VAR l: LONGINT);

Reads a string and converts it to a number of type **LONGINT. Leading blanks**
are ignored.

Note: To read and write double-precision real numbers, use the module *Doubles*
described later in this chapter.

   Note that when *t* is the standard text *input*, all characters read from the keyboard
are echoed on the screen. Line editing is possible until a ⎡RET⎤ is entered. If
y̧   wish to either input a character from the keyboard without echo or to enter
a carriage return, use the procedure *ReadChar* provided in the module *Terminal*.

Done(t:TEXT):BOOLEAN;

Monitors correct execution of numeric input requests. When applied after a
numerical read operation, *Done* returns a Boolean value that indicates whether
the input is valid.

Thus, when a number is read, *Done* is FALSE if the input is syntactically illegal
(for example, ABC) or too large to be converted (for example, 1000000) when a
cardinal is requested.

   Done is unaffected by **all other operations. Note the following** examples:

```
 Declarations:
VAR ch: CHAR; i: INTEGER; c: CARDINAL; r: REAL;
     s: ARRAY [0..20]OF CHAR;
     t: TEXT;
```

(
```
 ̨ ̗eration:      ReadInt(input,i);
Done is TRUE:    "0", "123", "-10".
Done is FALSE:   "ABC", "1.0", <EOL>, "60000" (too large).
Valid Range:     -32768 to +32767

Operation:       ReadCard(input,c);
Done is TRUE:    "0", "123", "60000",
Done is FALSE:   "ABC", "1.0", "100000" (too large), "-1."
Valid Range:     0 to 65535
```

```
Operation:        ReadReal(t,r);
Done is TRUE:     "0", "-3.", "3.14", "10.00E+10", "10E10".
Done is FALSE:    "E0", <EOL>, "10E40" (too big)
Valid Range:      -6.80564E38 to +6.80564E38

Operation:        ReadLong(t,l);
Done is TRUE:     "0", "-10000000", "78"
Done is FALSE:    "ABC", "1.0", "10000000000" (too big)
Valid Range:      -2,147,483,624 to +2,147,483,623
```

ReadLine(t: TEXT; VAR Line: ARRAY OF CHAR);

Reads a whole line of text into the string variable *Line*. This is similar to *ReadString*, but no leading blanks are skipped and input is terminated only by an EOL or EOT character.

ReadLn(t: TEXT);

Advances the read position past the EOL character of the last line read. All characters on the current line, including the next EOL, are skipped. The next read position starts a new line. This procedure is useful to catch trailing blanks on a line or for skipping over unwanted input.

ReadAgain(t: TEXT);

Causes the last character read from text *t* to be returned upon the next read operation. For example:

```
 Read(t,ch); ReadAgain(t);
 IF (ch>="0") & (ch<="9") THEN ReadCard(t,x)
 ELSE
  ReadString(t,s)
 END ;
```

EOLN(t: TEXT): BOOLEAN;

Returns TRUE if the last character read was an EOL (36C) character.

EOT(t: TEXT): BOOLEAN;

Returns TRUE if the last character read was an EOT (32C) character.

Once a text has been created, data can be written to it. Again, writing involves two operations: (1) the translation of the internal representation of data in the computer's memory to one or more characters, and (2) the transmission of these characters to the external medium. Following are procedures to write data.

WriteChar(t: TEXT; ch: CHAR);

Writes a single character to the text *t*.

WriteString(t: TEXT; s: ARRAY OF CHAR);

All characters in the given string *s* are written, starting from the first element (with index 0) up to the first null character (0C) or the end of *s*.

WriteInt(t: TEXT; i: INTEGER; FieldWidth: CARDINAL);

Write the integer *i* to text *t*. The given number appears right-justified in a field of *FieldWidth* characters. If *FieldWidth* is too small, more space is allocated.

WriteCard(t: TEXT; c,FieldWidth: CARDINAL);

Write the CARDINAL *c* to text *t*. The given number appears right-justified in a field of *FieldWidth* characters. If *FieldWidth* is too small, more space is allocated.

WriteReal(t: TEXT; r: REAL; FieldWidth: CARDINAL; Digits:INTEGER);

Writes real number *r*, using at least *FieldWidth* characters; *r* is right- justified in the *FieldWidth* character field.

The parameter *Digits* controls the number of digits used in the mantissa of *r*. If it is positive, *r* is written in fixed-point format with *Digits* characters after the decimal point. If it is zero, the decimal point is omitted. A negative value of *Digits* indicates scientific notation; that is, the decimal point follows the most significant digit and an exponent is present. In this case, the number of digits in the mantissa is equal to *ABS(Digits)*.

WriteLong(t: TEXT; l: LONGINT; FieldWidth: CARDINAL);

Writes long integer *l*, right-justified in a *FieldWidth* character field. If more than *FieldWidth* columns are needed to print *l*, more space is allocated.

WriteLn(t: TEXT);

Terminates output line. *WriteLn(t)* is equivalent to *WriteChar(t,EOL)*. The EOL character is translated to a carriage-return/line-feed pair.

SetCol(t: TEXT; Col: CARDINAL);

Advances the write position of text *t* to column *Col* in the current output line. If *Col* is not greater than the current column number of *t*, nothing happens. Columns are numbered from 0; that is, the left-most column has number 0.

Col(t: TEXT): CARDINAL;

Returns the current column position of text *t*. After a *WriteLn*, *Col* returns 0; it increases by 1 with every character written.

The following are examples of the preceding procedures:

Operation	Output
`WriteInt(output,1025,5);`	`"  1025"`
`WriteInt(t,1025,7);`	`"    1025"`
`WriteReal(output,12.28,10,);`	`"      12.3"`
`WriteReal(output,12.28,5,0);`	`"   12"`
`WriteReal(output,12.18,12,-5);`	`"  1.21800E+01"`
`WriteLong(output,-1000L*1000L,10);`	`"  -1000000"`
`SetCol(output,Col(output) DIV 10 + 1) * 10);`	Tabs to the next multiple of 10

The following example sums up input line by line. (This is recommended for very portable code; for an easier method, see the *READ* and *WRITE* statements that follow.)

```
MODULE SumUp;
FROM Texts IMPORT input, output, EOLN, EOT, ReadReal, Done,
                    WriteChar, WriteReal, WriteLn, WriteString;

VAR
  item,sum: REAL;
BEGIN
  REPEAT
    WriteString(output,"enter real numbers> ");
    sum:=0.0;
  (  PEAT
      ReadReal(input,item);
      IF Done(input) THEN sum:=sum+item END ;
    UNTIL EOLN(input) OR EOT(input);
    IF NOT EOT(input) THEN
      WriteString(output,"Sum =" );
      WriteReal(output, sum,12,-5 );
      WriteLn(output);
    END
  UNTIL EOT(input)
END SumUp.
```

READ and WRITE Statements

The number of procedures in the preceding section may seem overwhelming; however, the *READ* and *WRITE* statements are much easier to use. The compiler translates them into appropriate calls to procedures in the module *Texts*, which causes an implicit import from that module. *READ*, *READLN*, *WRITE*, and *WRITELN* are predeclared identifiers, not reserved words. Note that *READ* and *WRITE* statements are an extension to Modula-2 and their use may not be portable to (er Modula-2 implementations. Their full definition follows:

☐ Let *t* denote a variable of type *TEXT* and let *v,v1,...,vn* denote variables of a readable type; that is, of one of the types INTEGER, CARDINAL, CHAR, any subrange of them, REAL, LONGINT, or any ARRAY with CHAR elements.

☐ Let *p,p1,...,pn* denote parameters of one of the forms *el*, *el : e2*, or *el : e2 : e3*, where *el* is an expression of a readable type, *e2* is a cardinal expression, and *e3* is an integer expression.

READ(t,vl,...,vn)

Causes *vl,...,vn* to be read from text *t*. The read items must be separated by blanks or line ends. The text variable *t* may be omitted.

READ(vl,...vn) Causes *vl,...,vn* to be read from the standard text *input*.

READLN(t,vl,...,vn)

Works exactly like *READ*, except that the remainder of the last line read is skipped.

WRITE(t,pl,...,pn)

Causes *pl,...,pn* to be written to text *t*. If the text variable is omitted, output goes to the standard text *output*, for example.

WRITE(pl,...,pn)

Characters and strings are written without preceding or subsequent blanks (except for blanks in the string itself).

Integers and cardinals are written right-justified in a field of six characters, which can be overridden with a field specifier.

Reals are displayed in scientific notation, right-justified in a 12-character field. However, this may be overridden with a field specifier.

The field width of numbers can be changed by appending a colon and the desired width to the expression. The representation of real numbers can be changed to fixed-point by appending a second colon and the desired width of the fractional part (also see points 6 and 7 that follow).

WRITELN(t,pl,...,pn)

Works exactly like *WRITE*, except that the line is terminated when all parameters are written. The argument list may be empty; if so, only the EOL character is written.

The translation process from the *READ* and *WRITE* statements to calls of procedures in module *Texts* is as follows:

```
1. READ(v1,...,vn)     is translated to READ(Input,v1,...,vn).
   WRITE(p1,...,pn)    is translated to WRITE(Output,p1,...,pn).
```

The same equivalence holds for READLN and WRITELN.

```
 2.READLN(t,v1,...,vn) is translated to READ(t,v1,...,vn);
                       ReadLn(t).
( "RITELN(t,p1,...,pn)is translated to WRITE(t,p1,...,pn);
                       WriteLn(t).
```

```
3. READ(t,v1,...,vn)   is translated to
                       READ(t,v1);...;READ(t,vn).
   WRITE(t,p1,..,pn)   is translated to
                       WRITE(t,p1);...;WRITE(t,pn).
```

```
4. READ(t,v) is translated to
```

```
   ReadChar(t,v)       if v is of type CHAR.
   ReadString(t,v)     if v is of type ARRAY OF CHAR.
   ReadInt(t,v)        if v is of type INTEGER.
   ReadCard(t,v)       if v is of type CARDINAL.
   ReadReal(t,v)       if v is of type REAL.
   ReadLong(t,v)       if v is of type LONGINT.
   ReadDouble(t,v)     if v is of type LONGREAL.
```

```
5. WRITE(t,p) is translated to
```

```
( .riteChar(t,p)              if p is of type CHAR.
   WriteString(t,p)           if p is of type ARRAY OF CHAR.
   WriteInt(t,p,6)            if p is of type INTEGER.
   WriteCard(t,p,6)           if p is of type CARDINAL.
   WriteReal(t,p,12,-5)       if p is of type REAL.
   WriteLong(t,p,12)          if p is of type LONGINT.
   WriteDoubles(t,p,22,-14)   if p is of type LONGREAL.
```

Note that the standard width for the representation of a number is 6 in the case

of integers and cardinals and 12 in the case of reals; reals are normally written in scientific notation with a 5-digit mantissa.

Also note that you cannot replace the procedure call *ReadLine* with a *READLN* statement. *READLN*(line) would only read the first string of a line, stopping at the first blank encountered, then skipping to the end of the line without reading anything. On the other hand, *ReadLine(Input, Line1)* reads in the full line from the start to the next EOL character.

6. If *x* is of type INTEGER, CARDINAL, or LONGINT and *n* is a cardinal expression, then *WRITE(t,x:n)* is translated to *WriteInt(t,x,n)*, *WriteCard(t,x,n)*, or *WriteLong(t,x,n)*, respectively; that is, the default width specification is overridden.

7. If *x* is of type REAL or LONGREAL, then *WRITE(t,x:n)* is translated to *WriteReal(t,x,n,-5)*. Again, the default width specification is overridden.

    Furthermore, if *i* is an integer, *WRITE(t,x:n:i)* is translated to *WriteReal(t,x,n,i)*.

In this case, the form of notation (scientific or fixed-point) is indicated by the programmer. As in the previous explanation of procedure *WriteReal*, positive values of *m* indicate fixed-point notation with *i* digits behind the decimal point, whereas negative values stand for scientific notation.

The following example presents the same example from the earlier »Reading and Writing« section, except this time we make use of Turbo Modula-2's *READ* and *WRITE* extensions. The module becomes shorter and easier to read, even though it is translated into (nearly) the same code as the previous example. The *READ* and *WRITE* extensions do not in themselves shorten code generated by the compiler, rather they help to write shorter and clearer source programs.

```
MODULE SumUp;
FROM Texts IMPORT input, EOLN, EOT, Done;

VAR
 item,sum: REAL;
BEGIN
 REPEAT
  WRITE("enter real numbers>  ");
```

```
  sum:=0.0;
  REPEAT
   READ(item);
   IF Done(input) THEN sum:=sum+item END ;
  UNTIL EOLN(input) OR EOT(input);
  IF NOT EOT(input) THEN WRITELN("Sum =",sum:12:-5) END
 UNTIL EOT(input)
END SumUp.
```

User-Defined I/O Drivers

[ts provide communication with disk files and the logical devices of the operating system. If you wish to communicate with some device not falling into these categories, you can install your own I/O drivers and procedure for input and output. Such a procedure must match the following procedure type:

```
TYPE
 TextDriver = PROCEDURE(TEXT, VAR CHAR);
```

TextDriver is a template for the many possible I/O drivers. All of these procedures must accept a *TEXT* as their first parameter. Their second argument must be a VAR parameter of type CHAR.

If a text is used for output, the I/O driver sends its second argument; while a driver used for input deposits a character that was read in this parameter. Driver procedures must provide for the correct conversion of the EOL character into end-of-line character sequences used by the external device. The first parameter (of the subrange type *TEXT*) may serve as an index into a table that contains further information about the text on which the I/O takes place. In most situations it can be ignored.

˪ io install a user-defined I/O driver, use the following procedure:

```
ConnectDriver(VAR t: TEXT; driver: TextDriver);
```

After a call to this procedure, I/O on text *t* will be handled via the procedure that was substituted for the *driver* argument. This procedure has the effect of opening text *t*; no further *OpenText* or *CreateText* need be given.

The following is an example of installing a driver for an IEEE interface. We have presented the sketch of a local module, exporting the two texts *InIEEE* and

OutIEEE, communicating with an IEEE interface. It is assumed that line ends are denoted by single carriage-return characters.

```
IMPLEMENTATION MODULE IEEE;
FROM Texts IMPORT TEXT,EOL,ConnectDriver;
EXPORT InIEEE,OutIEEE;
CONST CR=15C;
VAR
 InIEEE,OutIEEE: TEXT;
PROCEDURE Put(t: TEXT; VAR ch: CHAR);    (* Output driver *)
BEGIN
 IF ch = EOL THEN ch:=CR END ;
 (* code to write ch to the IEEE interface *)
END Put;

PROCEDURE Get(t: Text; VAR ch: CHAR);    (* Input driver *)
BEGIN
(* code to read ch from IEEE interface *)
 IF ch = CR THEN ch := EOL END ;
END Get;

BEGIN (* Inititialization: Install Get and Put *)
 ConnectDriver(inIEEE, Get);
 ConnectDriver(OutIEEE ,Put)
END IEEE;
```

The texts *InIEEE* and *OutIEEE* may now be used to input or output characters, strings, or numbers. Statements like the following are now possible:

```
READ(InIEEE,x,y,s);
WRITELN(OutIEEE,"Size of Sample: ",SampleSize);
```

The InOut Module

InOut serves about the same purpose as the module *Texts*: It provides communication to external mediums without any assumptions about the nature of peripheral devices. The module *InOut* is postulated to be present in every Modula-2 system. It is included in the Turbo Modula-2 system for the sake of compatibility, even if abstract input and output is generally accomplished more conveniently by using the *READ* and *WRITE* extensions to Turbo Modula-2 in conjunction with the module *Texts*.

The operations provided by *InOut* can be thought of as a subset of those provided by *Texts*. *InOut* is more restrictive, however. Only operations on the standard text streams *input* and *output* are allowed. These standard streams are hidden by *InOut*, but can be accessed directly from *Texts*. As a consequence, procedures provided by *InOut* do not require a parameter to indicate which text stream is used. Read operations always use *input*, while write operations work on *output*. Correct execution of input procedures is monitored in the global variable *Done*, and the global variable *termCH* always contains the last character read.

In addition to the procedures provided by *Texts*, *InOut* exports two more **output** procedures.

WriteOct (c,n: CARDINAL);

Writes cardinal *c* in octal representation (range 0 to 177777 octal).

WriteHex (c,n: CARDINAL);

Writes cardinal *c* in hexadecimal representation (range 0 to FFFF hex).

Our implementation of *InOut* reads and writes real numbers as well as CARDINALs and INTEGERs. The user should keep in mind that other implementations might choose to separate these tasks. Procedures to read and write *REALs* would most likely be imported from the modules *RealInOut* or *RealIO*.

Note that when using the procedures *OpenInput* or *OpenOutput*, no file name is provided by the program. Instead, the name is requested from the user at the terminal, which limits the usefulness of these procedures.

InOut Module Specification

```
DEFINITION MODULE InOut;
CONST EOL=36C;
VAR Done: BOOLEAN;
    termCH: CHAR;

  PROCEDURE OpenInput (defext: ARRAY OF CHAR);
  (* Requests a file name at the terminal and connects input with that
     file. The file name may indicate one of the logical devices of
     CP/M. If the file name does not contain an extension itself, the
     default extension defext is appended.
  *)

  PROCEDURE OpenOutput (defext: ARRAY OF CHAR);
  (* Same as above but for output *)

  PROCEDURE CloseInput;
  (* Closes input file, returns input to terminal. *)

  PROCEDURE CloseOutput;
  (* Closes output file, returns output to terminal. *)

  PROCEDURE Read (VAR ch: CHAR);
  (* Done := Not past end of input *)

  PROCEDURE ReadString (VAR ss: ARRAY OF CHAR);
  (* Reads string as in Texts. Done := the returned string is not empty
  *)

  PROCEDURE ReadInt (VAR x: INTEGER);
  (* Reads integer as in Texts. Done := integer was read *)

  PROCEDURE ReadCard (VAR x: CARDINAL);
  (* Done := cardinal was read *)

  PROCEDURE Write(ch: CHAR);
  PROCEDURE WriteLn;
  PROCEDURE WriteString(s: ARRAY OF CHAR);
  PROCEDURE WriteInt(x: INTEGER; n: CARDINAL);
```

```
PROCEDURE WriteCard(x,n: CARDINAL);
PROCEDURE WriteHex(x,n: CARDINAL);
PROCEDURE WriteOct(x,n: CARDINAL);

PROCEDURE ReadReal (VAR x: REAL);
PROCEDURE WriteReal(x: REAL; n,digits: CARDINAL);
END InOut.
```

The Files Module

The next lower level of the module hierachy connecting *InOut* and *Texts* with the computer's disk-operating system embodies the concept of a file. There are two ways to look at a file. The first is as an external object: *A file describes a collection of data stored on a magnetic disk*. Often the data is of the same type. The second is as an internal object: *A file represents an internal data structure that describes the external data and how it is accessed*. The two notions should not be confused. The type *FILE* as exported by the module *Files* describes access to data items, not the data itself. In some operating systems this is referred to as a *file control block*.

Files offers the possibility to access data collected in a disk file. The elements of a file may be of any type and can be legible or illegible. No interpretation is performed; elements in a file are represented in computer internal binary form. As an example, writing the integer 279 to a *TEXT* consists of the output of the three numerals 2, 7 and 9, or in binary form:

```
---------------------------------------
| 00110010 | 00110111 | 00111001 |
---------------------------------------
```

When writing the same number as an integer to a file, its internal representation is written without prior conversion into ASCII digits, like so:

```
-------------------------------
| 00000001 | 00011001 |
-------------------------------
```

Physically, a file is a sequence of bytes on disk that can be read or written. The read or write position can be set to any byte of the file, thus allowing data to be accessed randomly.

It is often convenient to think of files at a higher level of abstraction. At this level, files are ordered collections of variables that are usually all of the same type. The type of the file elements is arbitrary--it could be CHAR, a record, or any other type.

Errors During File Handling
Files signals error conditions by raising one of five exceptions. If such an excep-

tion is raised, the executing program is stopped and a message describing the cause of the error is displayed on the screen. Programs may recover from such error conditions if the programmer provides an exception handler for the exception raised (refer to Chapter 9). *Files* defines the following exceptions:

StatusError   Indicates request of an illegal file-processing operation, such as reading from an unopened file or opening a file twice.

Files Module Specification

```
DEFINITION MODULE Files;
 FROM SYSTEM IMPORT BYTE, WORD, ADDRESS;
 TYPE FILE;

 PROCEDURE Open        (VAR f: FILE; name: ARRAY OF CHAR):
                                              BOOLEAN;
 PROCEDURE Create      (VAR f: FILE; name: ARRAY OF CHAR);
 PROCEDURE Close       (VAR f: FILE);
 PROCEDURE Delete      (VAR f: FILE);
 PROCEDURE Rename      (VAR f: FILE; name: ARRAY OF CHAR);

 PROCEDURE GetName     (f: FILE; VAR name: ARRAY OF CHAR);

 PROCEDURE FileSize    (f: FILE): LONGINT;
 PROCEDURE EOF         (f: FILE): BOOLEAN;

 PROCEDURE ReadByte    (f: FILE; VAR ch: BYTE);
 PROCEDURE ReadWord    (f: FILE; VAR w: WORD);
 PROCEDURE ReadRec     (f: FILE; VAR rec: ARRAY OF WORD);
 PROCEDURE ReadBytes   (f: FILE; buf: ADDRESS; nbytes: CARDINAL):
                       CARDINAL;

 PROCEDURE WriteByte   (f: FILE; ch: BYTE);
 PROCEDURE WriteWord   (f: FILE; w: WORD);
 PROCEDURE WriteRec    (f: FILE; VAR rec: ARRAY OF WORD);
 PROCEDURE WriteBytes  (f: FILE; buf: ADDRESS; nbytes: CARDINAL);

 PROCEDURE Flush       (f: FILE);
 (* Flushes the file's internal buffer to disk. Is used to detect
    DiskFulls at once *)

 PROCEDURE NextPos     (f: FILE): LONGINT;
 PROCEDURE SetPos      (f: FILE; pos: LONGINT);

 PROCEDURE NoTrailer   (f: FILE);

 PROCEDURE ResetSys    ( );
```

EXCEPTION EndError, StatusError, UseError, DeviceError, DiskFull;

END Files.

EndError	**Indicates attempt to read past the end of a file.**
UseError	Indicates file creation, renaming, or deletion on a write-protected disk, like the CP/M-message BDOS ERROR ON (drivename): R/O, which is suppressed in this case. Note that changed disks are always write-protected before a system reset. Also note that CP/M will not recognize a physically write-protected disk as R/O when reset; thus, *UseError* is not reliable for physically write-protected disks.
DeviceError	Indicates data is not readable, presumably because of a bad disk sector. This exception is also raised if a *Close* operation is unsuccessful.
DiskFull	Indicates that data cannot be written due to a full disk, or files or extents cannot be created due to a disk- directory overflow.

Operations on Entire Files

Before any processing can be carried out, the internal file must be connected with an external (disk) file. After the connection is made, the file is then said to be open. New files are opened with *Create* and existing files are opened with *Open*:

```
Open   (VAR f: FILE; name: ARRAY OF CHAR): BOOLEAN;
Create (VAR f: FILE; name: ARRAY OF CHAR);
```

These procedures connect an internal file *f* with an external file. *Open* searches the directory for an existing file with the given name, returning FALSE if no such file is found. *Create* always creates a new empty file; if a file with the same name already exists, it is deleted.

The following procedure performs a reset of the disk system. It is useful to prevent *UseErrors* caused by changed disks.

ResetSys;

After a call to *ResetSys*, it is possible to write to drives where disks have been swapped, but all open output files will be lost.

After processing has been completed on a file, the connection may be severed by the *Close* procedure.

Close(VAR f: File);

Disconnects *f* with its associated external file. If output has been sent to *f*, closing is mandatory; otherwise, all data of *f* will be lost. Since closing a file recovers the memory used for control blocks and data buffering, it is advisable to close a file in any case, even if it has only been used for input.

The following procedures are used to delete or rename a file:

```
Delete  (VAR f: FILE);
ename  (VAR f: FILE; name: ARRAY OF CHAR);
```

Both procedures assume that *f* is open. *Delete* deletes the directory entry that corresponds to *f*, while *Rename* renames the entry to its given name. Both procedures have the side effect of closing the internal file *f*.

The external name of file *f* is returned by the following:

```
GetName(f: FILE; VAR name: ARRAY OF CHAR);
```

The name is returned in a standard format: The first character is the drive code, which can assume the values *A* through *P*. The second and third characters are the user area of the file. The fourth character is always a colon. The file name is next, consisting of up to 8 characters, with its end marked by a period. If the file name contains an extension, it is appended. To hold a returned file name, the *name* parameter should be declared with at least 16 characters.

For example, assuming the procedure calls

```
Open(f, "B4:INDATA.DAT" );
Create(out, "OUTDATA")
```

then the results will be *GetName(f,fname)*, which will yield »B04:IN-DATA.DAT,« and *GetName(out,outname)*, which will yield »A00:OUTDATA.«

The following procedure returns the size of an external file:

```
FileSize (f: FILE): LONGINT;
```

This returns the exact number of **bytes** in the file. Since files **can** be larger than

65535, the maximal CARDINAL value, the result is of type LONGINT. (, since its maximum value is in the trillions, it should be adequate for some tir

Unfortunately, CP/M has no means to determine the exact size of a file in b) To correct this restriction, Turbo Modula-2 uses a special convention: The byte of the last CP/M record (128 bytes) in a file indicates the number of def bytes in the record. This last byte has an offset value of 128; that is, if it is no bytes in the record are defined. A value of 255 indicates 127 defined bytes. value below 128 means that the last record is completely filled. Using this byte and the BDOS function that returns the number of 128-byte records in a the file's exact size can be computed.

This scheme imposes almost no restriction on the file formats that can be r In particular, reading ASCII files generated by WordStar or other program possible without encountering problems. If you run into problems transfer files between Turbo Modula-2 programs and other software, you can use this cedure:

```
NoTrailer (f: FILE);
```

Note that the file *f* must be already open, thus allowing *NoTrailer* to cause last byte of the last CP/M record of *f* to be interpreted as a data byte instea a length byte. Consequently, a subsequent call of *FileSize(f)* will always retu multiple of 128. If the length of *f* is changed by write operations, no length will be appended to *f* when it is closed.

File Processing
We can think of a file as a sequence of bytes stored on disk. The number of l in a file is called the *length* of the file. A file's elements are all of the same and each can consist of several bytes. However, only one element can be acce at a time.

The position number of the accessible element is called the *current positii* the file. The current position can point to any element, or it may denote the of the file; that is, it can range from 0 up to the *FileSize(f)*. Initially, if no pro ing has been carried out, the current position will be 0. Reading advances the rent position past the read data. Likewise, writing advances the current pos past the modified data.

Once a file is open (an *Open* or *Create* operation applied to it), reading

writing becomes possible. The following operations are available for file processing and can be carried out only on open files. If one of these operations is applied to a file that is not open, the exception *StatusError* is raised. The file operations that follow read data from the given file *f* and differ in the kind of data that is read:

```
ReadByte  (f: FILE; VAR ch: BYTE);
ReadWord  (f: FILE; VAR w: WORD);
ReadRec   (f: FILE; VAR rec: ARRAY OF WORD);
```

ReadByte reads a single byte and assigns it to the parameter *ch*. *BYTE* is a special type, imported from the pseudomodule *SYSTEM*. It matches every variable occupying 1 byte of memory. This includes variables of the type CHAR, BOOLEAN, subrange types with bounds in the range 0 to 255, and enumeration types with, at most, 256 elements. If the file elements are one of these types, *ReadByte* should be used for reading.

ReadWord reads 2 bytes (a word) and assigns them to the parameter *w*. The type WORD is also imported from the pseudomodule *SYSTEM*. It matches any variable occupying 2 bytes of memory. Examples are variables of type INTEGER, CARDINAL, BITSET, sets, and all pointers. If the file elements are one of these types, *ReadWord* should be used for reading.

ReadRec reads everything not covered by *ReadByte* and *ReadWord*. The type ARRAY OF *WORD* matches variables of any type. This works since the storage of every element is rounded up to a word offset. As many words as are needed to fill out the given parameter are read. Employing this procedure for reading is appropriate for file elements occupying more than one word; in particular, for records, arrays, long integers, and reals.

In all three procedures, the current position is advanced by as many bytes as are read. A subsequent read operation will then access the next element in the file. If the current read position is higher than the end position, the exception EndError is raised.

The following file operations write data to the given file *f* and differ in the kind of data that is written:

```
WriteByte  (f: FILE; ch: BYTE);
WriteWord  (f: FILE; w: WORD);
WriteRec   (f: FILE; VAR rec: ARRAY OF WORD);
```

These procedures give the value specified in the second parameter to the element at the current position in the given file *f*. If the current position is not higher than the end position, the corresponding file element is modified. If the current position points to the end of the file, data is appended and the length of the file is incremented.

While *WriteByte* is appropriate for file elements occupying 1 byte, *WriteWord* is used for word-sized elements and *WriteRec* applies to elements of a larger size (for example, records, arrays, or reals).

If writing is impossible, the exception *DiskFull* is raised. There are also two lower level procedures for reading and writing data.

ReadBytes (f: FILE; buf: ADDRESS; nbytes: CARDINAL): CARDINAL;

Reads as many bytes as are specified in the *nbytes* parameter into consecutive addresses, starting at the address denoted by *buf*. If less than *nbytes* bytes remain in the file *f*, only the number of remaining bytes are read. Hence, a call to this procedure will never cause the exception *EndError* to be raised. The number of bytes read is returned as a function result.

WriteBytes (f: FILE; buf: ADDRESS; nbytes: CARDINAL);

Writes as many bytes as are specified by the given *nbytes*. Data is written from consecutive addresses, starting at the address *buf*.

EOF(f: FILE): BOOLEAN;

Returns TRUE if the end of file *f* is reached; returns FALSE if there are still bytes in *f* to be read.

Note that this procedure provides a limited form of *lookahead*. You know that the end of a file has been reached before undefined data has been read. This works even if the file is empty, a fact that distinguishes a *FILE* from a *TEXT*. With *TEXT* you have to read the EOT character to know that the *TEXT* has been completely read. The appropriate form to process a *FILE* is a **WHILE** loop:

```
  IF Open( f,fname ) THEN
  WHILE NOT EOF(f) DO
   ReadRec(f,data);
   Process(data);
  END;
  Close(f);
END
```

SetPos(f: FILE; pos: LONGINT);

Allows data to be accessed randomly. Sets the current position of the file f to *pos*. This parameter denotes the byte position where subsequent file processing will take place. The next read operation will then access the element at position *pos*. The next write operation will modify the element at this position. The parameter *pos* can point to any element in the file or it can point to the end of the file; its legal range is 0 up to and including *FileSize(f)*. If *pos* is larger than *FileSize(f)*, the exception EndError is raised.

If all file elements are of the same type T, and you know the ordinal number n of the selected element in the sequence (counting again from 0), you can compute its byte position as

```
BytePosition := n * SIZE(T)
```

Thus, a call like *SetPos(f,LONG(n)*LONG(SIZE(T)))* would set the current file position correctly. Note that *SetPos* requires a parameter of type LONGINT. Since type coercion is not performed in Modula-2, the construct *SetPos(f,0)* is illegal; use *SetPos(f,LONG(0))* or *SetPos(f,0L)* instead.

The current file position can be examined by calling

```
NextPos (f: FILE): LONGINT;
```

which will return the current position of the given file f.

To demonstrate how files work we will present two examples. The first example shows a series of operations on a file (represented by boxes) and how these procedures affect the file and the file pointer (the up arrow). The declarations VAR f: FILE; c: CHAR; are used in the procedure calls in Example 1.

Example 1

1. Open(f, " INDATA ");

```
-----------------------------------------------------
| 01001100 | 01100101 | 01101100 | 01101100 |
-----------------------------------------------------

  ^
```

Steps 2 through 4 show how to modify a file element: First read it in, then set the current position back by one and write the modified data.

2. ReadByte(f,c);

```
-----------------------------------------------------
| 01001100 | 01100101 | 01101100 | 01101100 |
-----------------------------------------------------

          ^
```

c now has the value **01001100 binary** (»L«).

3. SetPos(f,NextPos(f)-1L);

```
-----------------------------------------------------
| 01001100 | 01100101 | 01101100 | 01101100 |
-----------------------------------------------------

  ^
```

4. INC(c); WriteByte(f,c);

```
-----------------------------------------------------
| 01001101 | 01100101 | 01101100 | 01101100 |
-----------------------------------------------------

          ^
```

Steps 5 and 6 show how to append data at the end of a random access file: Set the position to the length of the file and write the data to be appended.

5. `SetPos(f,Size(f));`

```
----------------------------------------------------
| 01001101 | 01100101 | 01101100 |01101100 |
----------------------------------------------------
```

<center>∧</center>

6. `writeByte(f,"y");`

```
--------------------------------------------------------------------
| 01001101 | 01100101 | 01101100 | 01101100 | 01111001 |
--------------------------------------------------------------------
```

<center>∧</center>

Example 2 is a practical implementation of a file copy utility that shows the typical usage of the *Files* module. Its operation is similar to the Filecopy command in the Turbo Modula-2 shell. Data is first written into a temporary file with the extension −−−. If the transfer succeeds, the temporary file is renamed to whatever the user indicates. This scheme avoids an accidental loss of data.

Among others, two procedures of the module *FileExtensions* are imported, *StripExt* and *AppendExt*. (This module is not in the standard library. It is used as a programming example in the section on the module *Strings*.) In short, *StripExt* strips the extension from a file name and *AppendExt* appends an extension. The procedure *PromptFor*, imported from the standard module *Comline*, reads text from the command line or, if none is found, prompts the user for input.

Example 2

```
MODULE FileCopy;
FROM SYSTEM IMPORT ADR;
FROM FileExtensions IMPORT StripExt,AppendExt;
(* See the discussion of the Strings Module for a listing of the
   FileExtensions routines.
*)
```

```
FROM ComLine IMPORT PromptFor;
FROM Files IMPORT FILE,Open,Create,Close,Rename,Delete,ReadBytes,
                  WriteBytes,DiskFull;

CONST BufferSize = 20000;
VAR
 inFile,outFile            : FILE ;
 inName,outName,tempName   : ARRAY [0..20] OF CHAR;
 buffer                    : ARRAY [1..BufferSize] OF CHAR;
 fetched                   : CARDINAL;

BEGIN
 PromptFor("Copy file: ", inName);
 IF inName[0] # 0C THEN    (* no empty string entered *)
  PromptFor("Copy to : ", outName);
  IF outName[0] # 0C THEN
   IF Open(inFile, inName) THEN
    tempName:=outName;
    StripExt(tempName);
    AppendExt(tempName,"$$$");
    Create(outFile, tempName);
    REPEAT
     fetched := ReadBytes(inFile,ADR(buffer),BufferSize);
     WriteBytes(outFile, ADR(buffer), fetched);
    UNTIL fetched < BufferSize;
    Rename(outFile, outName);
    Close(inFile);
   ELSE WRITELN(inName, " not found.") END
  END
 END
EXCEPTION
 DiskFull: WRITELN("DISK FULL"); Delete(outFile)
END FileCopy.
```

Files with Elements of Mixed Types
Up to now, we have only encountered files with elements all of the same type.
However, sometimes it is convenient to include elements of varying types in one
file; for example, a file can have a header as its first element, containing various
information about a file's data. The actual data would follow the header in a dif-
ferent format. You can process such a file the same way you would a file with

uniform elements. The procedures *ReadByte*, *ReadWord*, *ReadRec*, *ReadBytes*, *WriteByte*, *WriteWord*, *WriteRec*, and *WriteBytes* are functional. Of course, the procedure chosen must match the type of accessed element. For example, *Read-Byte* would be used to read a character, while *ReadRec* would serve to input a record.

The Terminal Module

The module *Terminal* provides communication over the operator's terminal (the keyboard and the video screen), without the abstraction of a text stream. The following procedures show how to implement basic input/output routines.

ReadChar(VAR ch: CHAR);

Reads a single character from the keyboard. The character is not echoed on the screen, which distinguishes the module *Terminal* from *Texts* or *InOut*. Line ends are denoted by carriage returns (15C); no conversion to the EOL character is performed.

BusyRead(VAR ch: CHAR);

Tests to see if a character has been typed. A typed character is returned in the given *ch*. If no key is pressed, *ch* is set to 0C.

ReadAgain;

Causes the last character read to be returned again **upon the** next call of *ReadChar* or any read statement from *Texts* or *Terminal*.

ReadLine(VAR s: ARRAY OF CHAR);

Reads a line (terminated by a carriage return) **into a string.**

WriteChar(ch: CHAR);

Writes a single character; **no conversion from EOL to carriage-return**/line- feed pairs is performed.

WriteLn;

Terminates a line.

WriteString(s: ARRAY OF CHAR);

Writes a string to the terminal.

Terminal Module Specification

DEFINITION MODULE Terminal;

```
PROCEDURE ReadChar (VAR ch: CHAR);
PROCEDURE BusyRead (VAR ch: CHAR);
PROCEDURE ReadAgain;
PROCEDURE ReadLine (VAR CHAR);    s:              ARRAY           OF

PROCEDURE WriteChar (ch: CHAR);
PROCEDURE WriteLn;
PROCEDURE WriteString (s: ARRAY OF CHAR);

TYPE
  SpecialOps (clearEol, insertDelete, highlightnormal);
  OpSet      = SET OF SpecialOps;

VAR available      : OpSet;
VAR numRows,numCols : CARDINAL;

PROCEDURE ClearScreen;
PROCEDURE GotoXY(x,y: CARDINAL);

PROCEDURE ClearToEOL;
PROCEDURE InsertLine;
PROCEDURE DeleteLine;

PROCEDURE Highlight;
PROCEDURE Normal;

PROCEDURE InitScreen;
PROCEDURE ExitScreen;

END Terminal.
```

Terminals may require a different set of control codes to do certain screen functions, such as cursor positioning or clearing the screen. Thus the following procedures of this module work only if the Turbo Modula-2 system has been properly installed for the terminal running the system. **Note:** It is possible to install stand-alone Turbo Modula-2 programs as well as the system (see Appendix B »Installation Procedures«).

Some of the screen functions available in the module *Terminal* **are described** here.

ClearScreen ( );

Clears the screen and homes the cursor (places it in the upper left corner of the screen).

GotoXY(x,y: CARDINAL);

Places the cursor on column x, row y of the screen. Columns and rows are numbered from 0. The upper left corner is at (0,0). A typical display has 80 columns and 24 lines, as shown in Figure 11-3.

Figure 11-3. Typical Display Using GotoXY.
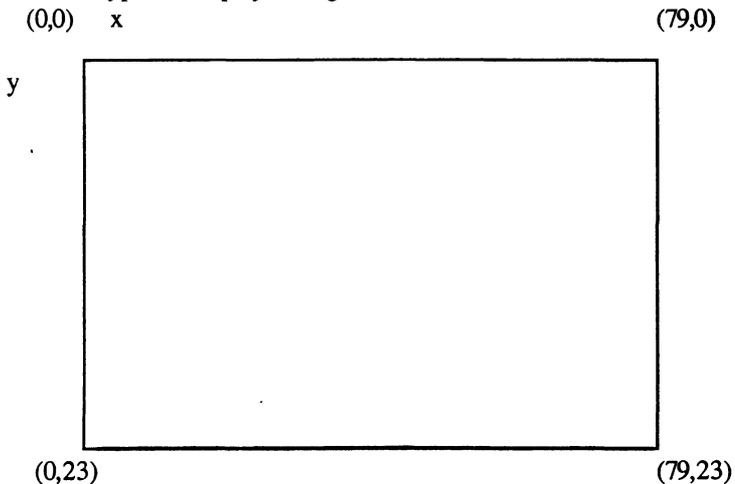
Inside the module *Terminal* are two variables, *numRows* and *numCols*, set by

the terminal initialization to give the screen height and width. They are defined as follows:

```
VAR
  numRows,numCols: CARDINAL;
```

where for common terminals, *numRows* = 24 and *numCol* = 80.

You can highlight a piece of text with the procedures *Highlight* and *Normal*.

Highlight highlights output to the screen by underlining it, using inverse video, or changing its brightness. This effect is dependent on the characteristics of your terminal and how the system is installed. *Normal* returns you to normal display mode.

The remaining procedures, *ClearToEOL*, *DeleteLine*, and *InsertLine*, are used mainly for text editing.

ClearToEOL deletes all characters from the cursor position to the end of the cursor line. *DeleteLine* deletes the cursor line completely and moves all following lines up one to close the gap. *InsertLine* inserts a blank line at the current cursor position.

After a call to any of these three functions, the cursor position is **undefined.** The cursor has to be placed explicitly with a call to *GotoXY.*

Since these functions are not available on all terminals, there is an emumeration type (*SpecialsOps*)and a set variable (*OpSet*) exported by *Terminal* that allow you to determine which functions were installed. They are shown in the following:

```
TYPE
  SpecialOps = (clearEol, insertDelete, highlightNormal);
  OpSet      = SET OF SpecialOps;

VAR
  available  : OpSet;
```

The variable *available* can be used as shown in the following examples. To clear to the end of a line, first determine if the procedure *ClearToEOL* is installed on the terminal.

The ComLine Module

A. This module provides access to the command line and handles the redirection of
2. input and output. The command line consists of any arguments that follow the
command that began the program. *ComLine* will work in both CP/M and Turbo
Modula-2 modes. The command line is simply declared in *ComLine* as a text.

```
VAR
  commandLine: TEXT;
```

Once imported, it can be read like any other text; for instance, *READ(com-
mandLine,fileName)* works perfectly. Of course, this text is quite short, consisting
of only a single line terminated by one EOT (32C) character. The text does not
need to be opened explicitly; *ComLine* does it on initialization.

You may have noticed that all Turbo Modula-2 system programs accept
arguments on the command line. If no argument is given there, the user is pro-
mpted to enter it. This is accomplished by a call to the following procedure:

```
PromptFor (prompt: ARRAY OF CHAR; VAR s: ARRAY OF CHAR);
```

This returns an input string in the result parameter *s*. *PromptFor* first tries to
perform a *ReadString* from the command line. If no further data is found there,
it displays the prompt string and performs a *ReadString* on *s* from the terminal.
This method is user friendly, since it allows for interactive use and batch process-
ing with command line arguments.

Redirection of Input and Output
The command line can contain redirection arguments: The symbol < , followed
y a file name, redirects the standard text *input*; while the symbol > , followed
by a file name, redirects *output*. Since many interactive programs will not work
with I/O redirection, redirection of input or output is not automatic; instead, there
are the procedures *RedirectInput* and *RedirectOutput*.

If an input redirection argument is given on the command line, *RedirectInput*
connects the standard text input with the file or device specified by that argument.
Similarly, *RedirectOutput* connects the standard text output with some device or
file, provided an output redirection argument is included in the command line.

If the command line is read, any redirection arguments **are ignored** to avoid interference with normal command line arguments.

Note: Redirected files are not automatically closed when a program terminates. Therefore, if output goes to a disk file, you must close the text (and the associated file) explicitly with the *CloseText* procedure from *Texts*.
CloseText(output) called before program termination will cause the redirected output file to be correctly closed.

ComLine Module Specification

```
DEFINITION MODULE ComLine;
FROM Texts IMPORT TEXT ;

VAR
  commandLine       : TEXT ;
  inName,outName    : ARRAY [0..19] OF CHAR;
  progName          : ARRAY [0..7] OF CHAR;

  PROCEDURE RedirectInput;
  PROCEDURE RedirectOutput;

  PROCEDURE PromptFor (prompt:ARRAY OF CHAR; VAR s:ARRAY OF CHAR);

END ComLine
```

The names following the symbols <or > on the command line are saved in the two variables *inName* and *outName*. These are declared **ARRAY** [0..19] OF CHAR;. If no redirection arguments are given, both *inName* and *outName* are set to »CON:«.

The last item exported by *ComLine* is the variable *ProgName*. It is declared as **ARRAY** [0..7] OF CHAR;.

This contains the name of the executing program (without drive code or extension). If a program is called from CP/M, this string is empty.

As an example, for *Comline* the following program CO copies the standard input to the standard output text. Using I/O redirection, it can be used for a large number of purposes, displayed in the following:

CO < B:TEXTFILE
                          Displays the contents of **TEXTFILE**
CO < B:TEXTFILE > LST: Prints out TEXTFILE
CO > B:TEXTFILE   Enters characters typed at the keyboard into
                          TEXTFILE until a ^ Z is read
CO (no arguments)   Guess

```
 MODULE CO;
FROM ComLine IMPORT RedirectInput,RedirectOutput;
FROM Texts IMPORT EOT,input,output,CloseText;
VAR
 ch: CHAR;
BEGIN
 RedirectInput; RedirectOutput;
 LOOP
  READ (ch);
  IF EOT(input) THEN EXIT END ;
  WRITE (ch);
 END ;
 CloseText( input); CloseText(output)
END CO.
```

The module *ComLine* has an unexported exception called NoInputFile. This ex-

ception is not usually trapped by user programs. If necessary it **can** be trapped with the **ELSE** clause in an exception handler, like so:

```
MODULE TrapUnexportedException;
FROM ComLine IMPORT RedirectInput, inName;
BEGIN
 WRITELN("inName is ",inName);
 RedirectInput;
EXCEPTION
 ELSE
  WRITELN('Must be non-existent file');
END TrapUnexportedException.
```

The exception is only raised if the name found in *inName* is not found. Thus, if »JUNK« is not a valid file, the following invocation of the program causes the error

```
Run MCD-file: MOO:UNEXPEXC <junk
```

```
inName is JUNK
Must be nonexistent file
```

Utility Modules

This section describes library modules that provide common mathematical and string-handling functions, and conversions to and from strings and numbers.

The MathLib and LongMath Modules

MathLib provides common mathematical functions, while *LongMath* provides the same functions for double-precision variables, replacing every declared REAL with a LONGREAL. Following are the functions provided by *MathLib*.

.qrt(x: REAL): REAL;

Computes the square root of the (positive) argument x. If x is negative, the exception ArgumentError is raised.

Exp(x: REAL): REAL;

Computes the natural exponent of the argument x. To prevent overflow from occurring, x must be smaller than 87.4; otherwise, the exception ArgumentError is raised.

Ln(x: REAL): REAL;

Returns the natural logarithm of the (positive) argument x. The exception ArgumentError is raised on negative x.

Sin(x: REAL): REAL;
Cos(x: REAL): REAL;

Returns sine and cosine of the argument x. The argument must be given in radians (radians = degrees * pi/180).

ArcTan(x: REAL): REAL;

Returns the arc tangent of argument x. The result is given in radians and lies within the interval of -pi/2 to pi/2.

```
DEFINITION MODULE MathLib;

  PROCEDURE Sqrt    (x: REAL): REAL;
  PROCEDURE Exp      x: REAL): REAL;
  PROCEDURE Ln       x: REAL): REAL;
  PROCEDURE Sin      x: REAL): REAL;
  PROCEDURE Cos      x: REAL): REAL;
  PROCEDURE ArcTan   x: REAL): REAL;
  PROCEDURE Entier   x: REAL): INTEGER;

  PROCEDURE Randomize(n: CARDINAL);
  PROCEDURE Random ( ): REAL;

EXCEPTION ArgumentError;
  (* Is raised if an argument is outside its legal range *)

END MathLib.
```

LongMath Module Specification

```
DEFINITION MODULE LongMath;

  PROCEDURE Sqrt    x: LONGREAL): LONGREAL;
  PROCEDURE Exp     x: LONGREAL): LONGREAL;
  PROCEDURE Ln      x: LONGREAL): LONGREAL;
  PROCEDURE Sin     x: LONGREAL): LONGREAL;
  PROCEDURE Cos     x: LONGREAL): LONGREAL;
  PROCEDURE ArcTan  x: LONGREAL): LONGREAL;
  PROCEDURE Entier  x: LONGREAL): LONGINT;

EXCEPTION ArgumentError;
(* Is raised if an argument is outside its legal range *)

END LongMath.
```

Entier(x: REAL): INTEGER;

Returns the integer part of a real number rounded toward negative infinity. For example:

if $x >= 0$, then *entier(x)* = *INT(x)*
if $x <= 0$, then *entier(x)* = *INT(x-1)*

Thus,

Entier(2.7) = 2
Entier(- 2.7) = -3
Entier(- 1.1) = 2
Entier(1.1) = 1

Other mathematical functions can be easily constructed by combining the preceding functions. For example:

Tan(x)	= *Sin(x)* / *Cos(x)*
x raised to the *y*	= *Exp(Ln(x)*y)*

The last tranformation is recommended only when *y* is not a whole number. For integer *y*, the following algorithm is more efficient:

```
MODULE PowerTest;

PROCEDURE power(x: REAL; n: INTEGER): REAL;
VAR
 i: CARDINAL;
 z: REAL;
BEGIN
 i := ABS(n); z := 1.0;
 WHILE i>0 DO
  (* z * x**i = x0**ABS(n) *)
  IF ODD(i) THEN z := z * x END ;
  x := x * x; i := i DIV 2
 END ;
 IF n>=0 THEN RETURN z
 ELSE RETURN 1.0/z
 END
```

```
END power;

BEGIN
 WRITELN(power(23.9,2));
END PowerTest.
```

In addition to the previous functions, *MathLib* (but not *LongMath*) contains a random-number generator. There are two user calls to the random-number generator: *Randomize* and *Random*.

Random( ): REAL;

Returns the next element of a pseudo-random-number sequence. Numbers are distributed evenly in the interval from 0.0 to 1.0.

Randomize(n: CARDINAL);

Reseeds the generator. A different random-number sequence corresponds to every possible argument *n*. Any random sequence may be repeated by providing *Randomize* with the same value of *n*.

Randomize takes one argument, a long (32 bit) integer, and sets the seed for the random-number generator equal to that number. *Random* takes no arguments, and returns a floating-point number x in the range $0 <= x < 1$. The number may be single precision or double precision.

Since there is a default value for the seed, most users need only call *Random*. *Randomize* is provided so that users can replicate simulations, since *Random* is a deterministic function of the seed. (In computer science jargon, it is a pseudo-random-number generator and not a true random-number generator.)

You should not call *Randomize* too frequently, because if *Randomize* is called for each call to *Random* then the numbers will not be much more random than the arguments to *Randomize*.

The least-significant bits in the real number returned by *Random* may not be particularly random.

The random-number generator maintains a 32-bit seed, which it treats as an unsigned integer. Each call to *Random* changes the seed by the formula

```
seed := 134775813 * seed + 1 mod 4294967296
```

and then returns the real number *seed / 4294967296*.

For example, if we call *Randomize* with the argument 1, then the seed is set **to** *1* and the next call to *Random* changes the seed to *134775814* and returns

```
134775814 / 4294967296 + 0.03137994...
```

No matter what the seed is, the random-number generator will pass through 4,294,967,296 numbers before it repeats.

The default value of the seed is 860098850.

The Strings Modules

A string in Modula-2 is an array with elements of type CHAR. The current contents of a string do not have to fill out the array completely; the end of the string is marked by a null character (0C). For convenience, there is the predefined type

```
TYPE
 String = ARRAY [0..80] OF CHAR;
```

The module *Strings* offers a number of functions to manipulate strings, concatenate them, extract or delete substrings, and so on. Since their parameters are open arrays, they will work with any array of characters, not only with the predefined type *String*. Keep in mind, however, that formal, open array parameters map an actual array parameter into an index range starting at zero. Therefore, all index parameters passed to the following procedures are expected to have a range starting at zero. Thus, we recommend setting the lower bound to zero when you define your own string type. The exported string procedures follow.

Length (VAR str: ARRAY OF CHAR): CARDINAL;

Returns the length of the contents of the given array. The returned length does not need to coincide with the size of the *str*. If the array *a* is filled completely, *HIGH(a)+1* is returned; otherwise, *Length* will yield the index of the first null character encountered.

For example, let *Alfa* be declared as **ARRAY** [0..100] OF CHAR. Thus,

```
Alfa := " ";    WRITE(Length(Alfa)) will give 0
Alfa := "ABC"; WRITE(Length(Alfa)) will give 3
```

Pos (substr,str: **ARRAY OF CHAR**): CARDINAL;

Pos scans for the first occurrence of the given string *substr* in string *str*. The procedure returns the index of the substring's first element. If *substr* is not contained in *str*, *HIGH(str)+1* is returned. Hence, this procedure can give back any value between 0 (if *substr* is a prefix of *str*) and *HIGH(str)+1* (if *substr* is not found).

Insert (substr: **ARRAY OF CHAR**; VAR str: **ARRAY OF CHAR**; inx: CARDINAL);

Inserts the given string *substr* into the given *str* at the index position *inx*. As always, indices are assumed to start at zero. If the array *str* is not large enough to hold the augmented string, the exception StringError is raised. Note that no error is raised if the *substr* is inserted at an index greater than the length of the string but still within the size of the string.

Strings Module Specification

```
DEFINITION MODULE Strings;
TYPE
 String = ARRAY [0..80] OF CHAR;

 PROCEDURE Length (VAR str: ARRAY OF CHAR): CARDINAL;
 PROCEDURE Pos    (substr,str: ARRAY OF CHAR): CARDINAL;
 PROCEDURE Insert (substr: ARRAY OF CHAR; VAR str: ARRAY OF CHAR;
                     inx: CARDINAL);
 PROCEDURE Delete (VAR str: ARRAY OF CHAR; inx,len: CARDINAL);
 PROCEDURE Append (substr: ARRAY OF CHAR; VAR str: ARRAY OF CHAR);
 PROCEDURE Copy   (VAR str: ARRAY OF CHAR; inx,len: CARDINAL;
                     VAR result: ARRAY OF CHAR);
 PROCEDURE CAPS   (VAR str: ARRAY OF CHAR);

EXCEPTION StringError;

END Strings.
```

Delete (VAR str: ARRAY OF CHAR; inx,len: CARDINAL);

Deletes *len* characters in the given string *str*, starting at the element with index *inx*.

Append (substr: **ARRAY OF CHAR**; **VAR** str: **ARRAY OF CHAR**);

Appends the given string *substr* to the given *str*. If the array *str* is not large enough to hold the augmented string, the exception *StringError* is raised.

Copy(VAR str: ARRAY OF CHAR; inx,len: CARDINAL; VAR result: ARRAY ( )F CHAR);

Assigns a substring of *str*, starting at position *inx*, consisting of *len* characters to string *result*.

CAPS (VAR str: ARRAY OF CHAR);

Changes all the characters in *str* to uppercase.

The following utility module demonstrates use of the *Strings* module. When dealing with files, it is often necessary to supply default extensions or to use several files with the same name but different extensions. These tasks can be simplified by using a library module like the following one:

```
DEFINITION MODULE FileExtensions;
TYPE
 Extension = ARRAY [0..2] OF CHAR;

 PROCEDURE AppendExt (VAR str: ARRAY OF CHAR; ext: Extension);
 PROCEDURE StripExt VAR str: ARRAY OF CHAR);
 PROCEDURE GetExt    (str: ARRAY OF CHAR; VAR ext: Extension);

END FileExtensions.
```

The procedure *AppendExt* supplies a default extension if the given *str* does not contain an extension itself. *StripExt* removes any extension in *str*, while *GetExt* returns the extension of *str* or the empty string if *str* does not have an extension. Using the services exported by *Strings*, these procedures can be implemented as follows:

```
   IMPLEMENTATION MODULE FileExtensions;
FROM Strings IMPORT Length, Pos, Append, Copy, Delete;


PROCEDURE AppendExt(VAR str: ARRAY OF CHAR; ext: Extension);
VAR len: CARDINAL;
BEGIN
 IF Pos(".", str) = HIGH(str)+1 THEN (* no "." in str *)
  len := Length(str);
  str[len] := "."; (* Strings go from 0 to Length(str)-1 *)
  Append(ext, str)
 END
END AppendExt;


PROCEDURE StripExt(VAR str: ARRAY OF CHAR);
VAR
 period: CARDINAL;
BEGIN
 period := Pos(".", str);
 IF period # HIGH(str)+1 THEN
  Delete(str, period, Length(str)-period)
 END
END StripExt;


PROCEDURE GetExt(str: ARRAY OF CHAR; VAR ext: Extension);
VAR
 period,elen: CARDINAL;
BEGIN
 period := Pos(".",str);
 elen   := HIGH(str)-period;
 IF elen > 3 THEN elen := 3 END ;
 Copy(str, period+1, elen, ext)
 END GetExt;


END FileExtensions.
```

The Convert Module

Convert exports procedures to convert strings into standard Modula-2 numeric types and back again. (*Doubles* provides the same services for double-precision real numbers.)

The first four procedures of the *Convert* module convert strings to INTEGERs, CARDINALs, LONGINTs, and REALs, respectively. The result is placed into the second parameter. The Boolean function result indicates whether a conversion is possible. It is TRUE if the string represents a legal number; otherwise, it is FALSE.

The second four procedures convert numbers to strings. The number is placed into the string starting at the right-most index. The left end of the string is padded with blanks. If the number does not fit into the given string, the exception TooLarge is raised. The *digits* parameter of procedure *RealToStr* has the same meaning as its equivalent in the procedure *WriteReal* (see the module *Texts*): It controls the size of the mantissa and the form of the string result (scientific or fixed point). For example, given the declaration

```
VAR
  s: ARRAY [0..11] OF CHAR;
```

a call of *RealToStr*(2.5, s, 2); gives »2.50« and *RealToStr*(2.5, s,-4) gives »2.5000E+00«.

Convert Module Specification

```
TRCMD3 DEFINITION MODULE Convert;

  PROCEDURE StrToInt  (VAR s: ARRAY OF CHAR; VAR i: INTEGER)  : BOOLEAN;
  PROCEDURE StrToCard (VAR s: ARRAY OF CHAR; VAR c: CARDINAL) : BOOLEAN;
  PROCEDURE StrToLong (VAR s: ARRAY OF CHAR; VAR l: LONGINT)  : BOOLEAN;
  PROCEDURE StrToReal (VAR s: ARRAY OF CHAR; VAR r: REAL)     : BOOLEAN;

  PROCEDURE IntToStr  (i: INTEGER;  VAR s: ARRAY OF CHAR);
  PROCEDURE CardToStr (c: CARDINAL; VAR s: ARRAY OF CHAR);
  PROCEDURE LongToStr (l: LONGINT;  VAR s: ARRAY OF CHAR);
  PROCEDURE RealToStr (r: REAL;     VAR s: ARRAY OF CHAR; digits: INTEGER);
  EXCEPTION TooLarge;
END Convert.
```

The Doubles Module

Doubles exports procedures that provide support for double-precision real numbers *(LONGREALs)*, similar to those provided for single-precision real numbers *(REALs)* in *Texts* and *Convert*.

As shown in the definition module, the variable *legal* monitors input from *Read-Double* in a similar fashion to *Done* and *ReadReal* in *Texts*. The other routines are identical to their single-precision *REAL* counterparts.

Doubles Module Specification

```
DEFINITION MODULE Doubles;
FROM Texts IMPORT TEXT;

VAR legal: BOOLEAN;

  PROCEDURE ReadDouble  (t: TEXT; VAR d: LONGREAL );
  PROCEDURE WriteDouble (t: TEXT; d: LONGREAL; n: CARDINAL;
                         m: INTEGER);

  PROCEDURE StrToDouble (VAR s: ARRAY OF CHAR;
                         VAR d: LONGREAL ): BOOLEAN;
  PROCEDURE DoubleToStr (d: LONGREAL;VAR s: ARRAY OF CHAR;
                         digits: INTEGER);


END Doubles.
```

System-Dependent Modules

The four modules described here provide utilities of a low-level nature. *Processes* and *Loader* are heavily dependent on the underlying system. They are coded in a high-level manner because *SYSTEM* and *STORAGE* provide a standard interface to machine-dependent operations such as memory allocation and process control.

The Processes Module

This module provides the synchronization between loosely coupled or largely in-
dependent processes. A process consists of a piece of program and a work space
its data. It is realized by the standard Modula-2 concept of a coroutine using
the primitives *PROCESS*, *NEWPROCESS*, and *TRANSFER* in the pseudomodule
SYSTEM.

A process takes the form of a parameterless procedure, which must be declared at the global level. Processes are not invoked by a procedure call, however. Instead, flow of control between processes is achieved via variables of type *SIGNAL*. Only two operations are applicable to signals: *SEND* or *Awaited*. When a process waits for a signal, its execution is paused; that is, control transfers to some other process. The waiting process is resumed when the awaited signal is sent by the currently executing process. More than one process can wait for the same signal. In that case, waiting processes are inserted into a queue, with the first process in the queue resuming once it has been sent the required signal. Sending an unanticipated signal has no effect.

The services exported by *Processes* are described next.

StartProcess(P: PROC; n: CARDINAL);

rts a concurrent process with program *P* and a workspace of size *n* (in bytes). A minimum workspace consists of about 50 to 100 words. After a call to this procedure, control is transferred to the newly created process. For more information on this procedure, see the procedure *NEWPROCESS* in the *SYSTEM* module and Wirth's book.

Processes Module Specification

```
DEFINITION MODULE Processes;
TYPE SIGNAL;

  PROCEDURE StartProcessP: PROC; n: CARDINAL);
  PROCEDURE SEND        (VAR s: SIGNAL);
  PROCEDURE WAIT        (VAR s: SIGNAL);
  PROCEDURE Awaited     (s: SIGNAL): BOOLEAN;
  PROCEDURE Init        (VAR s: SIGNAL);

EXCEPTION DeadLock;

END Processes.
```

SEND(VAR s: SIGNAL);

If there is no process waiting for *s*, *SEND* has no effect. If one is waiting, it is resumed. If several processes are waiting for the same signal, they are inserted into a queue. The first process in the queue is resumed when the call to *SEND*(s) occurs.

WAIT(VAR s: SIGNAL);

Causes the currently executing process to be suspended until it receives the signal (ᴾrocesses ready for execution are ordered in a queue. After a call to *WAIT*, the next process in the queue is resumed. If no process is ready for execution (that is, all processes are waiting), the exception *DeadLock* is raised.

Awaited(s: SIGNAL): BOOLEAN;

Returns TRUE if at least one process is waiting for *s*; otherwise, it returns FALSE.

Init(VAR s: SIGNAL);

Initialization of a *SIGNAL*; this is **mandatory** before a *SIGNAL* is **used**.

DeadLock;

This exception is raised if all current processes are waiting for a signal; it indicates an error in your signal-sending logic. A procedure constituting the program of a process cannot be called, nor can it return like a normal procedure. Its instructions are usually enclosed in a **LOOP** statement whitout a corresponding **EXIT**. (⁓ enable transfer of control to some other process, such a procedure must contain ⸜⸜ least one *SEND* or *WAIT* statement.

A state can arise where all processes are waiting for some signal. In this case, no process can execute and subsequently the system will come to a halt. This situation is known as a *deadlock*, or *deadly embrace*. When it is detected by *Processes*, the exception DeadLock is raised.

For example, when polling a number of external devices; assume the computer is connected to a number of external devices that can be serviced independently from one another. The computer will ask each device: Do you require service?

When a service request is found, it sends out the message: Device No. *n* nee
service. If there is a process ready to serve this device, it then takes control
the processor. When it is finished, this process starts the polling again by puttir
itself into a wait state for the next request. If the service is time-intensive, it cou
just as well enable polling repeatedly in the middle of its operation. This is achie
ed by waiting for the signal RESUME, which transfers control to a further lo
through the polling routine. Only when no further service is needed is the sign
RESUME sent, causing one of the processes waiting for it to be resumed.
sketch of a module that services devices by polling follows:

```
MODULE Poll;
FROM Processes IMPORT SIGNAL, SEND, WAIT, Awaited, Init,
StartProcess;
VAR
 Service1,...,ServiceN, Resume : SIGNAL;

PROCEDURE Device1;
BEGIN
 LOOP
  WAIT(Service1);
  (*"Service device #1"*)
  IF (*"Service is complicated"*) THEN
   (* Check if there are other devices waiting for service *)
   WAIT(Resume)
  END ;
  (*"Resume service of device #1"*)
 END
END Device1;

PROCEDURE Device2;
(* Body of Device2 similar to Device1 *)
END Device2;

        .
        .
        .

PROCEDURE DeviceN;
(* Body of DeviceN similar to Device1 *)
END DeviceN;
```

```
BEGIN (* Main program, containing the polling loop *)
 Init(Service1); ... Init(ServiceN);
 Init(DeviceCheck); StartProcess(Device1,100);
 StartProcess(Device2,500);
         .
         .
         .
 StartProcess(DeviceN,200);
 LOOP
  IF (*"ServiceRequest of Device #1"*) & Awaited(Service1) THEN
   SEND(Service1)
  ELSIF (*"ServiceRequest of Device #2"*) & Awaited(Service2) THEN
   SEND(Service2)
     .
     .
     .
  ELSIF (*"ServiceRequest of Device #N"*) & Awaited(ServiceN) THEN
   SEND(ServiceN)
  ELSE
   (* If there are any pending services, resume one of them *)
   SEND(Resume)
  END
 END
END Poll.
```

Note that the polling process encapsulated in the main program does not contain any *WAIT* statements. It is always available for execution, excluding the possibility of a deadlock. Every other process in this module pauses only by executing a *WAIT* statement; therefore, the only process ready for execution after issuing a wait is the polling process--a guarantee that polling occurs regularly.

A more rudimentary polling scheme can be implemented without the notion of processes and signals. In this scheme, the polling loop calls the service procedures directly when a service request is detected. However, in order to express a feature like the resume operation conveniently, processes are essential.

The Pseudomodule SYSTEM

This module contains types and procedures for low-level programming. An implementation part for this does not exist; rather, the compiler translates all exported operations directly into code, which is why *SYSTEM* is called a *pseudomodule*. Everything in it must be imported to become available, thereby making the use of low-level constructs explicit. The following sections describe the definition part of *SYSTEM*.

Low-Level Access to Data
The type *WORD* represents a machine word (2 bytes, 16 bits). Every type occupying a word or less is assignment-compatible with *WORD*. In particular, a formal parameter of type *WORD* accepts any actual parameter occupying one machine word. No operations (except assignment) can be performed on variables of type *WORD*. However, a variable of type *WORD* can be the argument of a type-transfer function.

The type *BYTE* represents a byte. It can be considered a subrange of type *WORD*. A *BYTE* can be assigned any value with a decimal equivalent in the range of 0 to 255.

Array of Word
If a formal parameter is declared as an **ARRAY OF** *WORD*, its corresponding actual parameter can be of any type. The upper bound of the open array is adjusted to match the size of the actual parameter; thus, no restrictions are imposed on a procedure's arguments. The most common examples of these procedures are *ReadRec* and *WriteRec* found in the library module *Files*.

Pseudomodule SYSTEM Specification

```
DEFINITION MODULE SYSTEM
TYPE
 WORD; BYTE; ADDRESS; PROCESS;
VAR
 IORESULT,HLRESULT: CARDINAL
 PROCEDURE ADR(VAR v: AnyType): ADDRESS;
 PROCEDURE TSIZE(AnyType): CARDINAL;

 PROCEDURE TRANSFER(VAR source, dest: PROCESS);
 PROCEDURE IOTRANSFER(VAR source, dest: PROCESS; n: CARDINAL);
 PROCEDURE NEWPROCESS(p: PROC; a: ADDRESS; n: CARDINAL;
                      VAR q: PROCESS);

 PROCEDURE BIOS(n: CARDINAL; w: WORD): CARDINAL;
 PROCEDURE BDOS(n: CARDINAL; w: WORD): CARDINAL;
 PROCEDURE CODE(AnyStringLiteral);

 PROCEDURE MOVE(source,dest: ADDRESS; len: CARDINAL );
 PROCEDURE FILL(adr: ADDRESS; len: CARDINAL; val: BYTE );
 PROCEDURE INP( port: WORD ): CARDINAL;
 PROCEDURE OUT( port: WORD; data: CARDINAL );

 EXCEPTION OVERFLOW,REALOVERFLOW,OUTOFMEMORY,BADOVERLAY;


END SYSTEM.
```

ADDRESS

ADDRESS is a type that represents memory locations. It can be thought of as a **POINTER TO** *WORD*; thus, dereferencing an *ADDRESS* will yield a *WORD*. Addresses can be computed; it is possible to add and subtract CARDINALs from addresses with the result being of type *ADDRESS*. *ADDRESS* is compatible with a CARDINAL and with every pointer type. In particular, if a formal parameter is of type *ADDRESS*, the corresponding actual parameter may be of any pointer type.

ADR

The procedure *ADR* returns the **address of its** argument. **It is declared as**

```
ADR(VAR x: "AnyStructuredType"): ADDRESS;
```

Note that since Turbo Modula-2 implements register variables in local procedures (see Chapter 8, »Low-Level Facilities«), the *ADR* function will not accept simple variables of unstructured types. This is because a scalar variable in a procedure may reside in a register that has no address.

TSIZE

The procedure *TSIZE* accepts the name of a type as its argument. It returns the minimum size of any variable of the argument type. Thus, when variable *x* is declared as VAR *x*: *T*, then *SIZE(x) = TSIZE(T)*.

TSIZE has an alternate syntax similar to the *NEW* procedure: If *T* is a record type whose last field is a variant, a tag value of this variant can be indicated behind the type. In this case, *TSIZE* returns the amount of memory used by a variable created with a call to *NEW* with the same tag- value indication. If the last variant contains another variant as its last field, a tag value for this variant may be given, too. Thus, *TSIZE* is informally declared as

```
TSIZE("AnyType")
```

   or

```
TSIZE("AnyType",TagValue,TagValue,...)
```

Coroutines and Interrupts

Modula-2 includes the concept of a coroutine, which consists of instructions and a workspace separate from the main program. A coroutine can represent a process or detail how statements will manipulate data. (The main program itself is a process, since it consists of statements acting on data.) Coroutines lead to the introduction of loosely coupled processes. At any one time, a single-processor computer can execute only one process. The pseudomodule *SYSTEM* exports the type *PROCESS* to be used to reference a coroutine.

Although the body of a coroutine looks like that of a parameterless procedure, its excecution is more like that of a main program. A coroutine cannot be called with a call statement or exited by a **RETURN** statement--coroutines should never reach the final end. Since coroutines are never exited by a **RETURN** or **END** statement, their procedure body is usually enclosed by a **LOOP** statement without a corresponding **EXIT**.

Control to and from a coroutine is done via the *TRANSFER* statement. When a coroutine transfers control away from itself, its state is preserved until it is reactivated by a transfer from another coroutine. When a coroutine's state is preserved, all variables retain their values and modules remain initialized. When reactivated, the coroutine continues with the statement directly after the last *TRANSFER*.

Note that the program bodies of coroutines and processes (as described in the section, »The Processes Module«) are remarkably similar. In fact, processes are modeled by coroutines. The module *Processes* implements *StartProcess*, *SEND*, and *WAIT* with the primitives that follow.

NEWPROCESS(p: PROC; a: ADDRESS; **n: CARDINAL; VAR q: PROCESS);**

creates a coroutine and initializes the given process *q* with procedure body *p* and a workspace of *n* words starting at address *a*. This workspace must be large enough to contain *p*'s local variables, local stack, and local heap. A subsequent *TRANSFER(r,q)* will transfer control to the first statement of procedure *p*. *PROC* is a standard procedure type representing a parameterless procedure. It is exported from the *SYSTEM* module as

```
TYPE
  PROC = PROCEDURE( );
```

TRANSFER(VAR source, dest: PROCESS);

Transfers control to the coroutine *dest*. The currently active coroutine is assigned to *source* and suspended. A subsequent *TRANSFER* to *source* will resume the suspended coroutine with the next statement after the transfer.

Coroutines are used to express interrupt handling, such that a coroutine waiting for an interrupt will transfer control to some other coroutine (the main program) to do something in the meantime. When the interrupt occurs, control is transferred back to the interrupt-handling coroutine by an unscheduled *TRANSFER* statement.

It is as if a transfer instruction is explicitly inserted into the executing coroutine at the point of interrupt. Since we do not know the state of execution of the interrupted routine, no explicit *TRANSFER* can be used. Thus, a special transfer instruction is provided to allow unscheduled transfers.

IOTRANSFER(VAR source, dest: PROCESS; n: CARDINAL);

Performs a normal transfer from *source* to *dest*. When the interrupt specified by the given *n* occurs, control is transferred back to the next statement after *IOTRANSFER*. Any available interrupt vector can become *n*.

Interrupts must sometimes be disabled to avoid disturbing critical program parts. This is done by placing critical procedures in a module that is given a priority. Such a module is called a *monitor*. The priority (a numeric value) is indicated in square brackets after the module name; for example:

```
MODULE CriticalStuff[1];
```

·   **IMPLEMENTATION MODULE Processes[1];**

When a procedure in a monitor is executed, only interrupts of a higher priority than the monitor are enabled. However, in Turbo Modula-2 there are no interrupts with priorities since the Z80 hardware does not usually recognize them. All interrupts are disabled when a piece of a monitor is executed; that is, any number in brackets after the module name shuts out all (maskable) interrupts.

Warning: CP/M is not designed for interrupts; in particular, it is not reentrant. Therefore, when a CP/M operation is suspended by an interrupt, the operation

(and possibly others as well) cannot be invoked by the interrupting routine. (Since interrupts are only of limited use when dealing with CP/M, those of you interested in a more detailed discussion can refer to Wirth's book.)

Z80-Specific Procedures
The following procedures allow the programmer to take direct advantage of certain Z80 machine instructions. *MOVE* and *FILL* are very fast and their use is recommended anywhere speed is important. *INP* and *OUT* are for accessing the Z80 I/O ports. The native code generator produces highly efficient inline code for *INP* and *OUT*.

MOVE(source,destin: ADDRESS);

Moves blocks of data in memory. Its operation is also appropriate for blocks of overlapping memory: If *source < destin*, then the upper addresses are moved first; and if *source > =destin*, then the lower addresses are moved first.

FILL(adr:ADDRESS; nbytes: CARDINAL; val: BYTE);

Fills the block of memory starting at *adr*, which is *nbytes* large, with the byte value *val*.

INP(port: WORD): CARDINAL;

Receives (inputs) a value from an I/O **port.**

OUT(port: WORD; data: CARDINAL;);

Outputs a value to an I/O port.

interface to CP/M
The next two procedures allow you to directly access the **operations of** CP/M.

BDOS(n,w: WORD);

Results in a call to memory location 5, the CP/M entry location. The function number is specified in the given *n*; *w* denotes an additional parameter. If some *BDOS* operation returns a result in the accumulator *A*, it can be found in the variable *IORESULT*. If an operation returns a 16- bit result in register HL, it can be retrieved from the variable *HLRESULT*.

IORESULT and *HLRESULT* are declared as CARDINALs:

```
VAR HLRESULT, IORESULT: CARDINAL;
```

The action performed by BDOS is shown in the following:

☐ Load register *C* with *n*.
☐ Load register DE with *w*.
☐ CALL 5.
☐ Assign the contents of accumulator *A* to *IORESULT.*
☐ Assign the contents of register HL to *HLRESULT.*

```
BIOS(n,w: WORD): CARDINAL;
```

Results in a call to the BIOS jump vector. The given *n* indicates which entry is called: 0=WARMBOOT, 1=CONSTAT, 2=CONIN, and so on. Its exact operation is as follows:

☐ Load A with the given *n*.
☐ Load BC with the given *w*.
☐ CALL *WARMBOOT+n*3.*
☐ Assign the contents of accumulator *A* to *IORESULT.*
☐ Assign the contents of register HL to *HLRESULT.*

For example, to read a character directly from the console, do the following:

```
BDOS(1,0); ch:=CHR(IORESULT);
```

To do the same using a call to the BIOS vector:

```
BIOS(2,0); ch:=CHR(IORESULT);
```

Assembler Interface
Assembly routines can be included in a Turbo Modula-2 program by using

```
CODE("Any string literal");
```

A call to *CODE* can be inserted behind a procedure heading, replacing the body of the procedure. Its body may not contain a declaration part, **BEGIN** statement, or any other statements. The final **END** must be present. Compilation of a *CODE*

statement causes the .COM file whose name is given as an argument to be inserted into the object program. For example:

```
PROCEDURE MOVE(source,dest: ADDRESS; n: CARDINAL);
 CODE("MOVE")
END MOVE;
```

establishes *MOVE* as an assembly routine. During compilation, the file MOVE.COM is read from disk and inserted into the object program. A call to *MOVE* will execute the assembly routine MOVE.COM. The file extension .COM supplied automatically.

Assembly routines can have an arbitrary number of parameters and can return results; that is, they can be used as function procedures. It is essential to declare function procedures with the correct result type. Assembly routines that return a result when none is declared can crash the system.

Since where an object program will be loaded is unknown at compile time, included assembly routines must be fully relocatable. Only relative jumps may be used, and calls to subroutines or absolute access to data within the routine are not allowed. Furthermore, to determine the exact size of a routine, it must contain its length in the first 2 bytes. These bytes are stripped off by the compiler; they are not executed.

An assembler routine can expect the following environment:

☐ Before an assembly procedure is called, its parameters are pushed onto the stack exactly as they are declared in the procedure heading. Value parameters of unstructured types are loaded immediately. VAR parameters and parameters of an **ARRAY** or **RECORD** type are loaded by address. The return address is at the top of the stack.

☐ Upon entry, register HL points to the beginning of the routine's object code. Thus, data embedded in the program text can be retrieved using HL plus an index.

☐ If an assembly routine returns a result, it is pushed onto the stack. In this case, be sure that all parameters have been removed from the stack and the return address is saved.

☐ The stack itself is 64 bytes deep. If more space is needed, it must be relocated and restored upon return. All registers can be destroyed.

To continue with the *MOVE* example, we will show how the stack is set up after a call to *MOVE*, but before any of *MOVE*'s instructions have executed. *MOVE*'s parameter *source* was pushed first, followed by *dest*, and then the count *n*. The return address is on the top of the stack and the current PC value is in HL.

For example, the parameters for the procedure *MOVE* are situated on the stack as follows:

High

source
dest
n
ret-addr

ret-addr ◄─── SP

Low
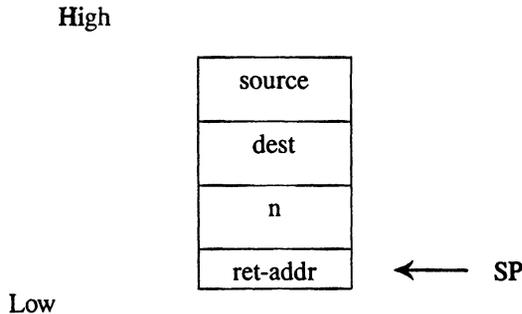
The assembly code for *MOVE* pops all parameters off the stack, replacing the return address. It then checks the count word and if it is greater than zero, it performs the move and returns.

A possible implementation of *MOVE* is shown in the following:

```
        ORG    100H          ;unimportant, must be relocatable

        DEFW   MOVEEND-MOVE  ;indicate size of object code

MOVE    POP    HL            ;save return address
        POP    BC            ;load BC with count n
        POP    DE            ;load DE with destination
        EX     (SP),HL       ;load HL with source and
                             ;put return address back on stack
        LD     A,B           ;if BC=0 then return
        OR     A,C
        RET    Z
```

```
        LDIR            ;Move

        RET

MOVEEND END
```

Modules in Memory Management

This section introduces the modules *STORAGE* and *Loader. Storage* allocates
deallocates dynamic data, while *Loader* loads and executes overlay modul
Since both modules affect the way data and programs are held in memory, l
first take a look at the memory organization of Turbo Modula-2.

The memory available for Turbo Modula-2 is the *transient program area* (T
of CP/M. The TPA starts at address 100 hex and extends to the start of the BD
part of CP/M. How Turbo Modula-2 uses this area is shown in Figure 11-5.

Figure 11-5. Turbo Module-2's Use of TPA

```
Highest
TPA Address
                  ┌─────────────────────────┐
                  │    Runtime Stack         │
                  │   (Grows Downward)       │
                  │                          │   Can use up the amount
                  │                          │   of memory left
                  │        Heap              │
                  │    (Grows Upward)        │
                  ├─────────────────────────┤
                  │   Area for Programs      │
                  │    and Global Data       │   Can vary in size
                  ├─────────────────────────┤
                  │      System Data         │   ca 1K
                  ├─────────────────────────┤
                  │   Support Modules:       │
                  │      ComLine             │
                  │      Files               │   ca 7K
                  │      Texts               │
                  │      Loader              │
                  │      Convert             │
                  │      Doubles             │
                  ├─────────────────────────┤
                  │   Turbo Modula-2 Core    │   ca 1.5K
                  ├─────────────────────────┤
                  │   M-code Interpreter     │   ca 4.5K
                  └─────────────────────────┘
Address 100H
```
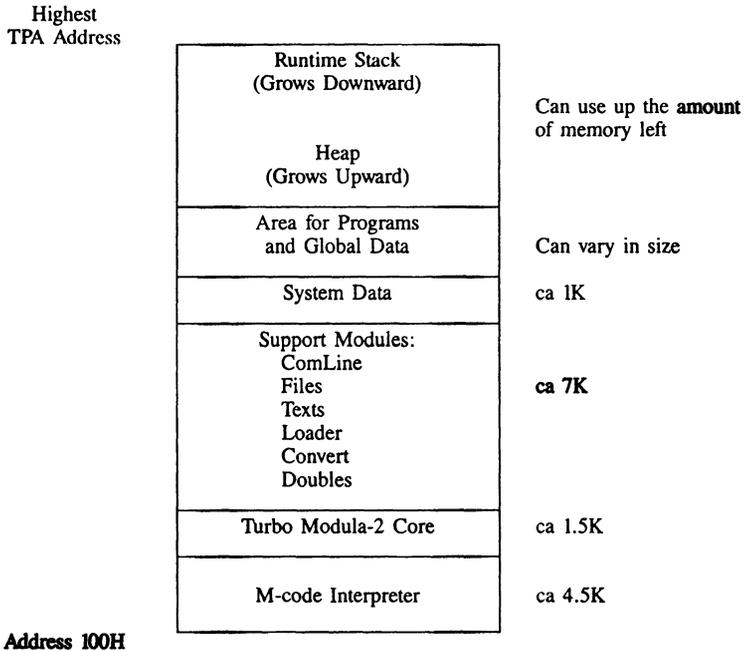
At the bottom of the TPA is the M-code Interpreter. All other parts of the T
Modula-2 system and all application programs written within it and not com
with the native code option exist in an intermediate code called M-code. Th
a variant of the instruction set of Wirth's Modula computer, Lilith. The co

optimized for space conservation. Every M- code instruction is interpreted by this part of the system.

The object code of the Turbo Modula-2 core resides directly above the interpreter. It constitutes a runtime supervisor that has the main tasks of setting up the rest of the system and diagnosing and handling exceptions (errors).

There are a few support modules used by virtually every program: *Texts*, *Files*, *Loader*, *Convert*, and *ComLine*. A program's loading time can be improved if these modules are already present in memory. Thus all of these modules are linked together in the file M2.COM, and are loaded into memory when the system is started.

The resident support modules occupy about 7K of main memory. If an application does not need these modules and memory space is at a premium, the program can be transformed into a .COM file using the linker (see Chapter 10 for more about the linker's operation). The linker will include into the .COM file only those modules that are needed. This can result in a memory savings of up to 6K.

There is space for other programs and their data above the area occupied by M2.COM. Such programs include the system file SHELL.MCD, which interprets the user's commands and the compiler COMPILE.MCD as well as application programs. The object modules together with their global data reside at the bottom of this area.

The remaining space is partitioned dynamically between the runtime stack for local data and the heap. The heap holds all variables that were created dynamically by the *NEW* and *ALLOCATE* procedures. The stack is used for all local variables, procedure parameters, and sometimes to evaluate complicated arithmetic or logical expressions. A program is out of memory when the runtime stack and the heap meet; in that case, the exception OUTOFMEMORY is raised.

The STORAGE Module

This module handles allocation and deallocation of dynamically created variables. All such variables reside in the heap area. Like *SYSTEM*, *STORAGE* is a pseudomodule. It has neither an object file (STORAGE.MCD) nor a symbol file (STORAGE.SYM). Instead, all the following services offered by this module are built into Turbo Modula-2. The following procedures are provided by the module *STORAGE*.

ALLOCATE(VAR a: ADDRESS; size: CARDINAL);

Allocates a space of *size* bytes on the heap and makes the pointer *a* point to the beginning of the allocated area.

Note: Due to the allocation scheme employed by *STORAGE*, a minimum of 6 bytes is always allocated, even if the *size* parameter specifies a smaller value.

DEALLOCATE(VAR a: ADDRESS; size: CARDINAL);

Releases the area referred to by the given pointer *a*. The released area is assumed to have the given *size*. The *size* must be equal to the size indicated when the area was created. The deallocated area is available for future allocation of other data. The pointer *a* is set to nil.

Standard Procedures Dependent on Storage

NEW(a:POINTER TO »AnyType«);

NEW is translated into a call to *ALLOCATE* of the form *ALLOCATE(a TSIZE(»AnyType«))*. *ALLOCATE* must be imported if the standard procedure *NEW* is used.

   DISPOSE(a: POINTER TO »AnyType«); ·

DISPOSE is translated into a call to *DEALLOCATE* of the form *DEALLOCATE a, TSIZE(»AnyType«))*. *DEALLOCATE* must be imported if a program makes use of the standard procedure *DISPOSE*.

Heap Pointer
The top of the heap is marked by the heap pointer, which is inaccessible to a

plication programs. The heap can contain usable space because of the deallocation of variables. If a variable residing in the middle of the heap is disposed, the released area is appended to a *free list*. There is one exception to this rule: If the variable residing at the high end of the heap is disposed, its area is not appended to the free list; instead, the heap pointer is decremented and the heap is compressed. When a variable is allocated, the free list is searched to find an area large enough to hold the required data. If the search is unsuccessful, the area is allocated at the top of the heap; that is, the heap pointer is incremented. Thus, the heap status consists of two elements: (1) the heap pointer, marking its top end, and (2) the free list, containing information about holes available for future allocation.

STORAGE Module Specification

```
DEFINITION MODULE STORAGE;
FROM SYSTEM IMPORT ADDRESS;

  PROCEDURE ALLOCATE    (VAR a: ADDRESS; size: CARDINAL);
  PROCEDURE DEALLOCATE  (VAR a: ADDRESS; size: CARDINAL);

  PROCEDURE MARK        (VAR a: ADDRESS);
  PROCEDURE RELEASE     (VAR a: ADDRESS);

  PROCEDURE FREEMEM     ( ): CARDINAL;

END STORAGE.
```

A user may want to handle the heap in a stack-like fashion; for example, create a number of variables and then throw them away all at once. The following procedures allow for this.

MARK(VAR a: ADDRESS);

Creates a new heap segment. The new segment is directly above the old heap pointer. Its free list is initially empty. Until a call to *RELEASE(a)*, all allocations will allocate space in the new segment. The given address *a* is made to point to the bottom of the segment; that is, it equals the old heap pointer.

RELEASE(VAR a: ADDRESS);

Releases the heap segment created by a call to *MARK(a)*. The heap pointer and free list of the old segment are restored. The heap pointer is reset to the given address *a*. This address must have been set by a previous call to *MARK(a)*. The address *a* has the value nil after a call to *RELEASE(a)*.

In a newly created heap segment, variables can be allocated and disposed of in the normal way. While allocation is always safe, disposing presents some problems. The user must make sure that only variables residing in the currently active heap segment are disposed. Disposing of a variable that resides in a segment buried below the one last created by a call to *MARK* will result in a *BadHeap* exception. Similarly, disposing of a variable in an already-released segment causes the same exception to be raised.

The following instruction sequence is legal:

```
MARK(a);
NEW(p);
NEW(q);
DISPOSE(p);
RELEASE(a)
```

And the following sequences result in the exception *BadHeap*:

```
NEW(p);
MARK(a);
DISPOSE(p);    (Disposing a variable in some buried
                segment is not allowed)

MARK(a);
NEW(p);
RELEASE(a);
DISPOSE(p);    (p is already released)
```

Note: Creating or opening a file allocates space on the heap. When the file or text is closed, this area is disposed again. As a result, the preceding precautions must also be taken when a file or text is closed. For example, the following is illegal:

```
Open(f,"INDATA");
MARK(a);
Close(f)
```

The user can interrogate the space that remains for heap allocation with the next procedure.

FREEMEM( ): CARDINAL;

Returns the number of bytes between the current value of the heap pointer and the current top of the runtime stack. The size of available holes contained in the free list is not taken into account. For this reason, *FREEMEM* gives somewhat pessimistic information about the space available for heap allocation. The exact number of available bytes is returned only if procedures *ALLOCATE* and *DEALLOCATE* are used in a stack-like manner.

Dynamic Variable Errors
While dynamic variables are a very powerful concept, they must be used with some care. The following are common sources of errors and how the module *STORAGE* reacts to them:

☐ *A variable is disposed when there are still other references to it.* In this case, the value(s) of the disposed variable is undefined. Worse, another variable may have been allocated in the same area. Assignment to the disposed variable would result in overwriting other data. Assignment could also ruin the free-list management, which would result in raising the exception BadHeap.

☐ *The heap becomes too big.* If the heap pointer overruns the runtime stack during an allocation request, no space is allocated and the exception OUTOFMEMORY is raised. Since OUTOFMEMORY is exported by *SYSTEM*, it can be handled by an exception handler in the application program. Note that it is possible to generate another OUTOFMEMORY exception in the exception handler by calling a procedure with many or large parameters (for example, *WriteString* or other I/O statements). Thus, exception handlers for OUTOFMEMORY must have enough memory to operate correctly.

☐ *A variable not residing in the active heap segment is disposed.* This error can only occur if *MARK*, *RELEASE*, and *DISPOSE* are not executed in the correct order. This problem will usually cause the exception *BadHeap* to be raised some time in the future.

The Loader Module

The *Loader* module handles the allocation and execution of object modules and provides a standard method to activate *overlays*. These services may be necessary if an application requires more memory than is currently available.

Note: The Turbo Modula-2 system linker provides a more flexible and convenient method for defining overlays, which does not require any code modifications (see the section, "The Linker," in Chapter 10). You should consider using it before using the procedures described here.

If you get an OUTOFMEMORY error during program execution, your program needed more memory than was available. Overlays provide one possible solution to memory problems, they work as follows: A large program usually consists of some number of largely independent pieces, not all of which have to be physically in memory at the same time. The Loader Module provides a method to dynamically load only those modules that are required. Once their execution is terminated, they disappear, freeing the memory space they occupied. Other program modules can be loaded in their place, and executed. Thus the memory re-

quired to run a large program system can be significantly less than the sum of all modules used by the program. *Loader* provides the *Call* procedure to dynamically load a module.

Call (modName: **ARRAY OF** CHAR);

Loads the object module specified by modName into main memory and starts its execution. When execution is complete, the memory space occupied by the object code and data is released again. Modules executed by *Call* involves memory management and disk operations in addition to normal procedure execution time.

As an example, *Call* ("OVER") loads and begins execution of the object file OVER.MCD. The called module can import and access all objects of the calling program. Direct communication in the reverse direction is not possible, because the called module and its data area disappear after execution.

Note: Direct communication between a called module and its caller is not possible. This can be solved by the use of objects (message buffers) imported by the called procedure from the caller. Such objects can be freely modified by the called procedure.

Variables created by an overlay with the *NEW* procedure are preserved after termination of the overlay.

Call searches for the specified module on all logged disk drives (as specified in the .MCD search path during installation). The variable *FirstDrive* can be used to indicate which drive should be searched first for any overlay modules. This can be used to improve the loading time for external overlays.

VAR
    firstDrive: [0..15]

Indicates the first drive to be accessed when searching for the next overlay. The encoding is 0 for drive A, 1 for drive B, and so on. An application may set *firstDrive* to provide the drive where the next execution of *Call* should start its search. This is useful if many overlays are used and the program does not know which drive the overlay files will be on until runtime.

If the loading of a module is unsuccessful, the exception LoadError is raised.

(The reason for an unsuccessful load is displayed in an additional message.) LoadError can occur because of the following:

☐ The object file is not found.

☐ There is not enough space to hold the loaded object modules.

☐ A version conflict is detected. Turbo Modula-2 maintains a system-wide version-control scheme. If objects of a definition module are imported, the importing module remembers the version of this definition module. If the definition module is later changed and recompiled, all client modules have to be recompiled as well. If this is not done, a version conflict results when the client module is loaded. Note that if a definition module is recompiled but not changed, its version is preserved.

The following example shows communication between overlay modules. The main module is declared as an implementation module so that it may export a communication buffer to the overlay modules. Overlays are controlled via this record.

```
DEFINITION MODULE Main;
TYPE
  Commands = ( SayHello, CallOver, none );
VAR
  OverlayCommBuffer: RECORD
                       Command: Commands;
                       StrBuf: ARRAY [0..80] OF CHAR;
                     END ;

END Main.
```

The main program imports the *Call* procedure from the *Loader* module as shown in the following:

```
IMPLEMENTATION MODULE Main;
FROM Loader IMPORT Call;
FROM Terminal IMPORT BusyRead;

PROCEDURE Pause;
  VAR ch: CHAR;
```

```
BEGIN
 REPEAT BusyRead(ch) UNTIL ch#0C;
END Pause;
PROCEDURE DisplayOverString;
BEGIN
 WRITE('The string communication buffer ');
 IF OverlayCommBuffer.StrBuf # '' THEN
  WRITELN('contains "',OverlayCommBuffer.StrBuf,'"')
 ELSE
  WRITELN('is empty.')
 END ;
END DisplayOverString;


BEGIN
 WRITELN('Main program about to call overlay one.');
 OverlayCommBuffer.Command := SayHello;
 OverlayCommBuffer.StrBuf := '';
 Call('OVER1');
 WRITELN('Return from call to overlay 1');
 DisplayOverString;
 WRITELN('Now call overlay one again with a null request.');
 WRITELN('Will it be reloaded, press a key to see . . .');
 Pause;
 OverlayCommBuffer.Command := none;
 OverlayCommBuffer.StrBuf := '';
 Call("OVER1"); WRITELN('Returned from second call to overlay 1',
     'did the drive light go on ?');
 DisplayOverString;
 Pause;
 WRITELN('Call overlay two, which will call overlay one.');
 OverlayCommBuffer.Command := CallOver;
 OverlayCommBuffer.StrBuf := '';
 Call("OVER2");
 WRITELN('Return from call to overlay 2',
     'the drive light did go on, right!');
 DisplayOverString;
 WRITELN('That''s all folks!');
 Pause;
END Main.
```

The overlay modules are defined as follows; **notice that** they **also** import the
Call procedure from the *Loader Module.*

```
MODULE Over1;
FROM Loader IMPORT Call;
FROM Main IMPORT OverlayCommBuffer,Commands;

BEGIN
 CASE OverlayCommBuffer.Command OF
  SayHello: WRITELN('Hello there. This is overlay one running');
            WRITELN('Message left in the string area. Goodbye!');
            OverlayCommBuffer.StrBuf :=
            'Message from overlay 1 passed to caller';
            |
  CallOver: WRITELN('Overlay one about to call overlay two. ');
            WRITELN('Say hello number two . . .');
            OverlayCommBuffer.Command := SayHello;
            OverlayCommBuffer.StrBuf := '';
            Call('OVER2');
            IF OverlayCommBuffer.StrBuf # '' THEN
             WRITELN('Overlay two passed me this string: ',
                 OverlayCommBuffer.StrBuf );
            END ;
            |
  ELSE
   WRITELN('Overlay two : no command passed');
 END ;
END Over1.

MODULE Over2;
FROM Loader IMPORT Call;
FROM Main IMPORT OverlayCommBuffer,Commands;

BEGIN
 CASE OverlayCommBuffer.Command OF
  SayHello: WRITELN('Hello there.');
            WRITELN('This is overlay two running');
            WRITELN('Message left in the string area.');
            WRITELN('Goodbye!');
            OverlayCommBuffer.StrBuf :=
```

```
                      'Message from overlay 2 passed to caller';
                      |
   CallOver: WRITELN('Overlay two about to call overlay one. ');
             WRITELN('Say hello number one . . .');
             OverlayCommBuffer.Command := SayHello;
             OverlayCommBuffer.StrBuf := '';
             Call('OVER1');
             IF OverlayCommBuffer.StrBuf # '' THEN
              WRITELN('Overlay one passed overlay two
                      this string: ',
                  OverlayCommBuffer.StrBuf );
             END ;
             |
   ELSE
    WRITELN('Overlay two : no command passed');
  END ;
END Over2.
```

Loader Module Specification

```
 DEFINITION MODULE Loader;
 VAR
  FirstDrive: [0..15];

  PROCEDURE Call(modName: ARRAY OF CHAR);

  EXCEPTION LoadError;

 END Loader.
(
```

Chapter 12
Turbo Modula-2 Reference Directory

This chapter provides a complete alphabetical directory to Turbo Modula-2's standard identifiers, extensions, library procedures and modules, and reserved words.

Each entry contains a definition, usage and any restrictions, general comments, an example, a declaration, and a cross reference(s) to similar items where appropriate.

The following notation is used throughout this look-up section to describe operators, statements, types, procedures, and/or modules.

( )	Where appropriate, module names are enclosed in parentheses after an entry.
[ ]	Items enclosed in square brackets are optional; the items can be used only once or not at all.
{ }	Items enclosed in curly brackets can be repeated any number of times, including none.
"[" , "]" , "{" , "}"	The use of square or curly brackets within quotation marks means the bracket is part of the syntax and not just notational.
< >	Items enclosed in angle brackets indicate that another syntactic construct is defined elsewhere.
UPPERCASE	Words written entirely in uppercase characters generally have a special meaning. All reserved words and standard identifiers are written in uppercase. All identifiers exported from the pseudomodules *SYSTEM* and *STORAGE* are also uppercase. Other library identifiers that are uppercase are usually special; for example, the

file types *TEXT* and *FILE* exported from the modules *Texts* and *Files* are uppercase.

Each entry in the Reference Directory contains a Class descriptor. This is a quick reference to describe which of the general categories listed below the entry belongs to. In some cases, an entry belongs to more than one general category, and is listed appropriately. The Class descriptor can take on the following values:

E   = Turbo Modula-2 Extension

L   = Library

R   = Reserved Word

S   = Standard Identifier

ABS standard function   S

Description	ABS returns the absolute value of the expression X.
Usage	`Y:= ABS(X)`
	Argument X is of type REAL, INTEGER, LONGINT, or LONGREAL.
	Result Y must be the same type as X.
Comments	The absolute value is the positive value of a number; thus, if the function argument X is -12, the result is 12. A positive value is returned unchanged.
Example	

```
CONST
 Mass = 120.0;
VAR
 XPar,AbsX:INTEGER
 Acceleration,AbsForce:REAL

BEGIN
 AbsX := ABS(XPar);
 AbsForce := ABS(Mass * Acceleration);
```

ADDRESS (SYSTEM)   L

Description	*ADDRESS* is a type that represents memory addresses.
Declaration	`TYPE` `ADDRESS = POINTER TO WORD;`
Usage	`VAR a:ADDRESS`
	ADDRESS must be imported from the pseudomodule *SYSTEM*.

Comments   *ADDRESS* can be considered as **POINTER TO** *WORD*; thus,
                     dereferencing an *ADDRESS* will yield a *WORD*.

                     Addresses can be computed, and it is possible to add and sub-
                     tract cardinals from addresses. The result of such a computa-
                     tion is again of type *ADDRESS*.

                     ADDRESS is compatible with CARDINAL and with every
                     pointer type. In particular, if a formal parameter is of type *AD-
                     DRESS*, the corresponding actual parameter may be of any
                     pointer type.

                     This is a low-level concept; its use will reduce the portability
                     of the code that employs it.

                     Note: Taking the ADR of a simple **variable is not allowed.**

See Also   ADR
                     SYSTEM
                     WORD

Example   **MODULE** WhereAreXandY;
                     FROM SYSTEM **IMPORT** ADDRESS, ADR;
                     FROM InOut **IMPORT** WriteHex;
                     TYPE
                      atype = **RECORD**
                               A,B: REAL
                               END ;
                     VAR
                      adrx, adry : ADDRESS;
                      x, y   : atype;
                     BEGIN
                      adrx := ADR(x);
                      adry :- ADR(y);
                      WRITE('x is at '); WriteHex(adrx); WRITELN;
                      WRITE('y is at '); WriteHex(adry); WRITELN;
                     END WhereAreXandY;

ADR procedure (SYSTEM)   L

Description	*ADR* returns the address **of a structured variable.**
Declaration	**PROCEDURE** ADR(**VAR** v: AnyType): ADDRESS;
Usage	y := ADR(x);

ADR must be imported from the pseudomodule *SYSTEM*.

x may be a variable of any structured type and *y* must be a pointer or a cardinal variable.

Comments   *ADR* yields a value of type *ADDRESS*, which is a 2-byte value indicating the address of the argument variable.

This is a low-level routine; its use will reduce the portability of the code that employs it.

See Also   ADDRESS
SYSTEM

Example

```
MODULE WhereAreXandY;
FROM SYSTEM IMPORT ADDRESS, ADR;
FROM InOut IMPORT WriteHex;
TYPE
 atype = RECORD
            A,B: REAL
          END ;
VAR
 adrx, adry : ADDRESS;
 x, y       : atype;
BEGIN
 adrx := ADR(x);
 adry := ADR(y);
 WRITE('x is at '); WriteHex(adrx); WRITELN;
 WRITE('y is at '); WriteHex(adry); WRITELN;
END WhereAreXandY;
```

ALLOCATE procedure (STORAGE)   L

Description   *ALLOCATE* allocates an area from the heap.

Declaration   **PROCEDURE** ALLOCATE (**VAR** a: ADDRESS;
                                                    size: CARDINAL);

Usage   ALLOCATE(a, size)

                     ALLOCATE must be imported from the pseudomodule
                     STORAGE.

                     a must be compatible with type *ADDRESS* (any pointer).

                     size must be of type CARDINAL.

Comments   A space of *size* bytes is allocated on the heap, and the pointer
                     a is made to point to the beginning of the allocated area. If the
                     memory requested is not available the exception
                     OUTOFMEMORY is raised. You may check available
                     memory with *FREEMEM*.

                     A safer and more comfortable way to allocate dynamic
                     variables is by using the procedure *NEW*, although you must
                     still import *ALLOCATE.*

                     The counterpart routine to release this memory is called
                     DEALLOCATE.

                     Note that the size of an *ADDRESS* is only 2 bytes; thus *AD-
                     DRESS* is compatible with *WORD*.

See Also   DEALLOCATE
                     DISPOSE
                     FREEMEM
                     OUTOFMEMORY
                     NEW
                     STORAGE

Example   Allocate 100 bytes off of the heap:

```
VAR
 work: ADDRESS;
BEGIN
 ALLOCATE(work, 100);
```

AND   R

Description   AND is a **binary logical operator resulting in a Boolean ex-**
                 pression.

Usage   **IF Raining AND** NoUmbrella **THEN** Run **ELSE** Walk **END** ;

                 AND is a reserved word.

                 Raining and *NoUmbrella* **are Boolean expressions.**

Comments   AND is used to concatenate two logical expressions, where
                 both subexpressions must be true for the entire expression to
                 be true.

                 Note that the second subexpression in an AND expression is
                 only evaluated if the first subexpression is true. This is called
                 short-circuit evaluation.

See Also   OR
                 NOT

Example   WHILE statement using an AND to test bounds **condition**
                 prior to indexing an array (short-circuit evaluation):

```
VAR
 s: ARRAY [1..maxitems] OF Info;
BEGIN
 WHILE i#0 AND s[i] > amount DO
  (* process s[i] *)
 END ;
END ;
```

APPEND procedure (STRINGS)   L

Description	*Append* appends a substring to a string.

Declaration   **PROCEDURE** Append(substr: **ARRAY OF** CHAR;
                                 VAR str: **ARRAY OF** CHAR);

Usage   Append(substr, str)

               Append **must be imported** from the library module *Strings*.

               str and *substr* can be arrays of character with any bounds.

Comments   A string's »end« is marked by a zero byte; thus *Append* will
               tack on *substr* at the first zero byte in *str*.

               If *str* has not been declared to be long enough to hold the two
               together, the exception StringError is raised.

See Also   Delete
               Insert
               StringError
               Strings

Example   Write out 'Hello world !':

```
MODULE Main;
FROM Strings IMPORT Append;
VAR
 str, substr: ARRAY [0..80] OF CHAR;
BEGIN
 str     := 'Hello';
 substr  := ' world !';
 Append(substr, str);
 WRITELN(str);
END main.
```

Arctan function (MATHLIB, LONGMATH)   L

Description
: *Arctan* returns the arc tangent of *X*, with the result expressed in radians.

Declaration
: **PROCEDURE Arctan (x: REAL): REAL;**

and

PROCEDURE Arctan (x: LONGREAL): LONGREAL;

Usage
: `Y:= Arctan(X)`

Arctan must be imported from the library module *MathLib* for single-precision reals and from *LongMath* for double- precision reals.

Argument *X* and the target *Y* of the result must be the same type, either both REAL or both LONGREAL.

Comments
: The result is within the range -Pi/2 to Pi/2 radians. If you want the result expressed in degrees instead of radians, use the following expression:

`Degrees:= Arctan(X) * 180.0/3.141592`

The inverse of the *Arctan* function is tangent, which you can calculate with the *Sin* and *Cos* functions:

`Tan:= Sin(X)/Cos(X).`

See Also
: Mathlib
LongMath
Sin
Cos

Example
: ```
MODULE ArcTanTest;
FROM MathLib IMPORT Arctan;
CONST
 RadsToDegs = 180.0/3.141592;
```

```
VAR
 Opposite, Adjacent, Angle: REAL;

BEGIN
 Opposite := 1.0;
 Adjacent := 1.0;
 Angle := Arctan(Opposite/Adjacent) * RadsToDegs;
 WRITELN('Angle=',Angle,' and it better be
 45 degrees !');
END ArcTanTest.
```

ArgumentError exception (MATHLIB)   L/E

Description	ArgumentError is raised by the module *MathLib* if an invalid parameter is passed to one of the procedures exported by *MathLib*.
Declaration	**EXCEPTION** `ArgumentError;`
Usage	**EXCEPTION** `ArgumentError: WRITELN('Argument error');` ArgumentError must be imported from the module *MathLib*.
Comments	This exception is raised if the parameter to a procedure is out of the valid range for that function.
See Also	MathLib LongMath

Example   **MODULE** MathError;
                         FROM MathLib **IMPORT** Exp, ArgumentError;
                         VAR x: REAL;
                         BEGIN

```
MODULE MathError;
FROM MathLib IMPORT Exp, ArgumentError;
VAR x: REAL;
BEGIN
 x := Exp(99.0);
EXCEPTION
 ArgumentError:
  WRITELN('argument to Exp must be less than 87.4');
END MathError.
```

ARRAY standard type   R

Description   An array consists of a fixed number of components all of the same type, the *base type*. Each component can be individually accessed by indices whose values are of the *index type*. Arrays can also be treated as a whole when being assigned or passed as parameters.

Usage   a = **ARRAY** IndexType **OF** ComponentType

The index type can be BOOLEAN, CHAR, subrange, or enumeration types. The *ComponentType* can be of any type.

Multidimensional arrays are expressed with additional index types separated with commas.

Comments   You can use an array to handle several related items of identical
                            type; for example, an array of soups of the day could be

```
Day = (Sun,Mon,Tue,Wed,Thu,Fri,Sat);
Soup = (Minestrone, CreamOfAsparagus,
        VegetableBeef, SplitPea, ChickenNoodle,
        Chowder, Bouillabaisse);

SoupOfTheDay = ARRAY Day OF Soup;
```

Each component in an array has an index with a value defined
by the index type. Thus, in the preceding example, a compo-
nent is specified with an index of type *Day*; for example,
SoupOfTheDay[Mon], *SoupOfTheDay[Sat]*, and so on.

A component can take values defined by the component type.
In the example, an element of *SoupOfTheDay* can take any
value from *Minestrone* to *Bouillabaisse*.

A multidimensional array is simply an **array with an array as**
a component type. For example:

```
a2=ARRAY IndexType1 OF ARRAY IndexType2 OF CARDINAL
```

This can also be rewritten as

```
a2 = ARRAY IndexType1, IndexType2 OF CARDINAL
```

The components are accessed as either

```
MultiDim[Index1][Index2]
```

or

```
MultiDim[Index1,Index2]
```

No operators other than simple assignment are directly ap-
plicable to arrays, but any operator applicable to the compo-
nent type can operate on a component. Thus, if *Numbers* is an
array of REAL components, then

```
Numbers[X]:= Numbers[Y] * Numbers[Z]
```

Simple assignment is possible for identically structured arrays. For example, if *First* and *Second* are arrays of the same type, the following assignment makes each component of *First* equal to the corresponding component in *Second*:

```
First:= Second
```

Example

```
MODULE Arrays;
TYPE
 Alphabet = ['A'..'Z'];
 Days = (Sun,Mon,Tue,Wed,Thu,Fri,Sat);
 Shift = RECORD
            Day : [0..8];
            Swing: [0..8];
            Night: [0..8]
         END ;
VAR
 HoursSun  : ARRAY Days OF REAL;
 Character : ARRAY [1..10] OF Alphabet;
 CharType  : ARRAY CHAR OF (Letter,Digit);
 WorkRecord: ARRAY Day,[1..100] OF Shift;
BEGIN
 HoursSun[Tue] := 12.6;
 HoursSun[Mon] := HoursSun[Sat] + HoursSun[Sun];
 Character[3]:= "R";
 Character[1]:= Character[3];
 WorkRecord[Sun,1] := 0;
END Arrays.
```

available variable (TERMINAL)   L

Description	*available* holds the currently installed terminal operation, which may not be available on all terminals.

Declaration

```
TYPE
 SpecialOps = (clearEol, insertDelete, highlight-
 normal);
 OpSet : SET OF SpecialOps;
VAR
 available : OpSet;
```

Usage

IF clearEol IN available THEN ClearToEOL END ;

available is a predefined variable declared as type *OpSet*, which must be imported from the module *Terminal*.

Comments

This variable is useful for writing terminal-independent programs. The value of *available* is set by the installation program INSTM2.

See Also

SpecialOps
OpSet
Terminal

Example

```
MODULE TermStuff;
FROM Terminal IMPORT available, SpecialOps,
                    ClearToEOL,
                    InsertLine, numRows, numCols;
VAR
 I, CurRow: CARDINAL;
 ScreenLine: ARRAY [0..23] OF ARRAY [0..79] OF
  CHAR;
BEGIN
 IF insertDelete IN available THEN InsertLine
 ELSE
  FOR I := 1 TO numCols DO WRITE(' ') END ; WRITELN;
  FOR I := CurRow TO numRows DO
   WRITELN(ScreenLine[I])
  END ;
 END ;
END TermStuff.
```

Awaited procedure (PROCESSES)   L

Description   *Awaited* tests to see if **any processes are waiting for a certain**
              signal.

Declaration   **PROCEDURE** Awaited(s: SIGNAL): BOOLEAN;

Usage   flag := Awaited(siggy)

               Awaited must be imported from the library module *Pro-*
               cesses.flag must be of type BOOLEAN and *siggy* must be of
               type *SIGNAL* imported from *Processes*.

Comments   *Awaited* only makes sense when used with the related routines
               WAIT and *SEND*.

               Awaited will not start up the waiting processes, it only lets the
               program know if processes are waiting for a certain signal.
               Presumably, if *Awaited* returns TRUE, then the signal will be
               sent with the *SEND* command.

See Also	Init
	Processes
	SEND
	StartProcess
	WAIT

Example	**IF** Awaited(Printer) **THEN** SEND(PrinterAvailable) **END**

BADOVERLAY exception (SYSTEM)   L/E

Description	BADOVERLAY is raised by the **module** *SYSTEM* when **an** overlay file cannot be read.

Declaration	**EXCEPTION**
	BADOVERLAY;

Usage	**EXCEPTION**
	BADOVERLAY:
	WRITELN('Bad **overlay**');
	END
	BADOVERLAY must be imported from the module *SYSTEM*.

Comments	This exception occurs only when the linker is used to produce overlays.

See Also	SYSTEM
	OUTOFMEMORY
	OVERFLOW
	REALOVERFLOW

Example
```
BadOverlays;
FROM Overlayed IMPORT Proc1;
FROM SYSTEM IMPORT BADOVERLAY;
BEGIN
 Proc1;
EXCEPTION
 BADOVERLAY:
  WRITELN('Could not load overlayed module.');
END BadOverlays.
```

BDOS procedure (SYSTEM)   L

Description BDOS invokes a BDOS (CP/M operating system) function.

Declaration PROCEDURE BDOS(n, w: WORD);

Usage BDOS(n,w)

BDOS must be imported from the pseudomodule *SYSTEM*.

n and w must be compatible with type *WORD* (which can be an integer, character, enumeration, and so on).

Comments The BDOS function n will be invoked; w is a parameter to the operating system, inserted in register DE.

If a function returns a result in the accumulator A, it can be extracted out of the variable *IORESULT*.

If a result is returned in the register HL, it can be extracted from *HLRESULT*.

They are declared in *SYSTEM* as

VAR

HLRESULT,IORESULT:CARDINAL.

See Also	BIOS
	SYSTEM
	IORESULT
	HLRESULT
	Also see the BDOS section of your CP/M manual.

Example	Read a character from the console:
	`BDOS(1,0); ch:=CHR(IORESULT);`

BEGIN   R

Description	BEGIN is a key word that divides the declaration part of a module or procedure from the statement part of the same.

Usage	**BEGIN**

Comments	A BEGIN is used to mark the beginning of the code part o a procedure or module.

See Also	END

Example	**MODULE** Begin;
	BEGIN
	IF TRUE **THEN**
	WRITE('Only ');
	WRiTELN('one BEGIN')
	END ;
	END Begin.

BIOS procedure (SYSTEM)   L

Description	BIOS results in a call to the specified BIOS jump vector.

Declaration	**PROCEDURE** BIOS(n, w: WORD);

Usage   BIOS(n,w)

BIOS must be imported from the pseudomodule **SYSTEM**.

n and *w* must be compatible with type *WORD* (which can be an integer, character, enumeration, and so on).

Comments   The given *n* indicates which vector is called: 0=WARMBOOT, 1=CONSTAT, 2=CONIN, and so on.

If a function returns a result in the accumulator *A*, it can be extracted out of the variable *IORESULT.*

If a result is returned in the register HL, it **can be extracted** from *HLRESULT.*

HLRESULT and *IORESULT* are declared in *SYSTEM* **as** cardinals.

See Also   BDOS
SYSTEM
HLRESULT
IORESULT

Example   Read a character directly from the console:

BIOS(2,0); ch := CHR(IORESULT);

BITSET standard type   S

Description   BITSET is a variable type that can assume as a value a set of
                     up to 16 integers with values between 0 and 15.

Usage   As a constant set:

                     [BITSET] "{" [element {,element}] "}"

                     element may be a constant, subrange, or variable of type CAR-
                     DINAL with a value between 0 and 15. The standard identifier
                     BITSET is optional in constant sets.

                     The standard identifier is used mainly in variable declarations,
                     as in the following:

                     VAR
                      mask: BITSET;

Comments   BITSET is a standard predefined set type. All the operators
                     you can apply to other set types can be used on BITSET-type
                     variables (see »Operators« in Chapter 3).

                     You denote BITSET constants in the same way as other set
                     types:

                     {0..9,12,14} { }   BITSET{6,8,11}.

                     A **BITSET** value **requires** 2 bytes for storage **in memory.**

Example

```
MODULE BitSets;
CONST
 Mask = {5..7,12};
VAR
 Bits: BITSET;

BEGIN
 Bits:= {1,3,6..13};
 Bits:= Mask * Bits;    (* AND the bits *)
(* Bits is now {6,7,12} *)

 Excl(Bits,7);          (* Exclude bit 7 *)
(* Bits is now {6,12} *)
END BitSets.
```

BOOLEAN standard type   S

Description

BOOLEAN is a variable type that can assume the logical truth values FALSE and TRUE.

Comments

You can use BOOLEAN variables and expressions when you want to calculate a statement that will have the answer TRUE or FALSE; for example, the statement »Today is Tuesday and it is raining« gives a BOOLEAN result.

There are two groups of operators you can apply to obtain BOOLEAN expressions: relational operators and logical operators.

Relational operators compare different variables, constants, and so on, and give a BOOLEAN result. For example, the expression:

```
(Altitude > 2000.0)
```

compares the REAL variable *Altitude* with the value 2000.0 to get a TRUE or FALSE result. Thus, relational operators can compare any items of similar type.

On the other hand, logical operators are only applicable to BOOLEAN variables and expressions. For example:

SwitchOn **AND** (Altitude > 2000.0)

has a logical AND operator and is TRUE only if both parts of the expression are TRUE.

A BOOLEAN value requires 1 byte for storage in arrays **and** 2 bytes as stand-alone variables or fields in a record.

Example
```
MODULE Booleans;
VAR
 PowerAvailable, SwitchOn, PowerOn: BOOLEAN;
 Temperature:REAL;
BEGIN
 PowerAvailable:= SwitchOn AND PowerOn;
 IF PowerAvailable AND (Temperature > 90.0) THEN
  WRITELN('Turn on air conditioner ');
 END
END Booleans.
```

BusyRead procedure (Terminal)   L

Description   *BusyRead* tests to see if a character has been typed at the keyboard.

Declaration   **PROCEDURE** BusyRead(**VAR** ch: CHAR);

Usage   BusyRead(ch)

                BusyRead must be imported from the system library *Terminal*.

                ch must be of type CHAR.

Comments   *BusyRead*, together with *ReadAgain*, provides much the same service as Turbo Pascal's *KeyPressed*; however, *BusyRead* and *ReadAgain* are more flexible.

If a character is typed at the keyboard, *BusyRead* will return it in *ch*. If nothing is typed, *ch* is returned as 0C (a zero byte). For example:

```
PROCEDURE KeyPressed( ): BOOLEAN;
VAR ch: CHAR;
BEGIN
 BusyRead(ch);
 IF ch#0C THEN
  ReadAgain; RETURN TRUE
 ELSE
  RETURN FALSE;
 END
END KeyPressed;
```

The procedure emulates the *KeyPressed* function. To use the character again in another portion of the code we can use *ReadAgain*.

Notice that using the function *KeyPressed* requires two steps: (1) determine if there is a character and (2) if there is, read the character. This can be a problem in Modula-2 because of its interrupt handlers.

Suppose a program tests (with *KeyPressed*) whether a character is at the keyboard, and before it can read the character it is interrupted. If the interrupting routine also reads the keyboard, then when control returns to the interrupted program the character will not be available. This problem is solved if the procedure that reads a character simply returns a flag when no characters are available.

See Also   ReadAgain
             Terminal

Example   Display the ordinal value of a key that is pressed:

```
MODULE BusyReads;
FROM Terminal IMPORT BusyRead;
VAR ch: CHAR;
BEGIN
 REPEAT
  WRITELN('Press a key ('Q' to quit): ');
  REPEAT
   BusyRead(ch);
  UNTIL ch#0C;
  WRITELN('The ordinal value is ',ORD(ch));
 UNTIL CAP(ch)='Q'
END BusyReads.
```


BYTE type (SYSTEM)   L

Description   *BYTE* is a low-level type used to represent the scalar types that
                   take up 1 byte of memory.

Usage   **VAR**
                    b :BYTE;

                   BYTE must be imported from the pseudomodule *SYSTEM*.

Comments   *BYTE* is declared as a subrange of *WORD* and is therefore
                   assignment-compatible with all scalar types.

                   BYTE is compatible with BOOLEAN, CHAR, and enumera-
                   tion types that have 256 elements or less.

                   BYTE is compatible with *WORD*.

                   Only assignment operations are allowed on variables of type
                   BYTE. A stand-alone variable of type *BYTE* takes 2 bytes of
                   storage (also fields in a record). A *BYTE* in an array is packed
                   into 1 byte of storage.

Example   **PROCEDURE** p1(b: BYTE);

Call procedure (Loader)   L

Description   *Call* allocates and executes an overlay in the form of a module.

Declaration   **PROCEDURE** Call(modName: **ARRAY OF** CHAR);

Usage   Call(modname)

                 Call must be imported from the library module *Loader.*

                 modname must be the character array containing the name of
                 the overlay file.

Comments   If *modname* does not have an extension, .MCD will be ap-
                 pended.

                 The object module specified by *modname* will be loaded into
                 memory and begin executing. In this sense, it is not unlike a
                 procedure call, but it is much slower due to disk access. Also,
                 when execution is completed, the memory needed for object
                 code and data is released again.

                 Communication between the calling program and the overlaid
                 module is only possible through variables declared in the call-
                 ing program and imported by the called overlay.

See Also   firstDrive
                 Loader
                 Also see linker described in Chapter 10.

Example   Load and execute the module *OVER.MCD*:

                 Call("OVER")

CAP standard function   S

Description   *CAP* returns the capital of an alphabetic character.

Usage   Ch := CAP(Ch)

               Argument *Ch* (and result) is of type **CHAR.**

Comments   You can use this function if you want to change lowercase le
               ters to uppercase letters. This is useful for testing keyboard in
               put.

               Note that if the argument has no uppercase equivalent, th
               argument is returned unchanged. In other words, spaces an
               punctuation characters will remain unaltered.

Example
```
MODULE CAPs;
VAR ch: CHAR;
BEGIN
 LOOP
  BusyRead(ch);
  IF CAP(ch) = 'Q' THEN EXIT END
 END
END CAPs.
```

CAPS procedure (Strings)   S

Description   *CAPS* makes an entire string variable uppercase.

Declaration   **PROCEDURE** CAPS(**VAR str: ARRAY OF** CHAR);

Usage   CAPS(str)

               CAPS must be imported from the library module *Strings*.

               str must be of type ARRAY OF CHAR.

| Comments | *CAPS* saves you from typing the following code anywhere you don't want to distinguish between uppercase and lowercase: |

```
FOR i := 0 TO Length( str )-1 DO
  str[i] := CAP( str[i] );
END ;
```

| See Also | CAP
Length |

| Example | Capitalize the string »Hello«: |

```
str := "Hello";
CAPS(str);
(* str now has the value "HELLO" *)
```

CARD standard function   S
_____

| Description | *CARD* converts a numeric argument to type CARDINAL. |

| Usage | `Y:= CARD(X)` |

Argument *X* can be of INTEGER, CARDINAL, REAL, LONGINT, or LONGREAL.

The argument must be in the range 0 to 65535.

Result *Y* is of type CARDINAL.

| Comments | When you use this function with a REAL argument, the value is truncated (the fractional part removed); thus, value 12.35 will become 12. |

CARD is not applicable to enumeration or BOOLEAN- and CHAR- type arguments. Use *ORD* with these types.

If the argument cannot be converted (that is, to negative), the runtime error BoundsError is raised.

Example   **MODULE** CARDConversion;
                 VAR
                  r: REAL;
                  c: CARDINAL;
                 BEGIN
                  r := 3.14;
                  c := CARD(r);
                  (* c now has the value 3 *)
                 END CARDConversion.

CARDINAL standard type   S

Description   CARDINAL is a standard type with variables that can assume
                 the values between 0 and 65535. In addition, CARDINAL may
                 be used as a type-transfer function.

Usage   **VAR** c: CARDINAL;

                 or

                 c := CARDINAL(-1)

Comments   You can use CARDINAL variables whenever you know tha
                 possible values are limited to positive whole numbers. For ex-
                 ample, if you are counting the number of people in a room
                 you know the count can only be a positive whole number. I
                 your program suddenly produces a negative count, an erro
                 message will be written.

                 The *CARD* standard function will convert other simple dat;
                 types into CARDINAL. The use of CARDINAL as a typ
                 transfer function should be considered a low-level facility tha
                 may not be portable.

                 A CARDINAL value requires 2 bytes for storage.

Example
```
MODULE CARDINALs;
CONST
 SearchValue = 3.65;
 N = 20;
VAR
 DataArray: ARRAY [0..N] OF REAL;
 I, Count : CARDINAL;
BEGIN
 FOR I:= 0 TO N DO
  IF (DataArray[I] = SearchValue) THEN
   Count:= Count + 1
  END
 END
END CARDINALs.
```

CardToStr procedure (Convert)   L

Description   *CardToStr* converts a **CARDINAL to a** string.

Declaration   **PROCEDURE** CardToStr(c: CARDINAL;
                                        VAR s: ARRAY OF CHAR);

Usage   CardToStr(card, string)

CardToStr must be imported from the library module *Convert*.

card must be of type CARDINAL.

string can be any array of characters.

Comments   The number is placed into the string starting at the right- most index. The left end of the string is padded out with blanks. If the number does not fit into the given string, the exception TooLarge is raised.

See Also   Convert
            IntToStr
            RealToStr
            StrToCard

Example
```
MODULE xcts;
FROM Convert IMPORT CardToStr;
VAR
 card: CARDINAL;
 string: ARRAY [0..4] OF CHAR;
BEGIN
 card := 12;
 CardToStr(card, string);
 WRITE('"', string '"');   (* "  12" is output *)
END xcts.
```

CASE statement   R

Description   CASE selects a different statement sequence depending on the value resulting from an expression.

Usage
```
CASE Selector OF
 Case {| Case}
[ELSE Statement sequence]
END
```

Selector is a variable or expression of type CARDINAL, IN-TEGER, BOOLEAN, CHAR, enumeration, or subrange.

```
Case = [Case label: Statement sequence]
```

```
Statement sequence = Statement {; Statement}
```

Each case label can be either a single constant value or a range of constant values of the same type as the selector, separated by commas.

The ELSE part is executed if it is present and there are no matching values in the CASE label lists.

Comments   The selector, whether it is an expression or a variable, gives a value that causes the selection and execution of the statement sequence containing that value in its label. A value can appear in only one label.

Example

```
CASE X + Y OF
| 0..5 : Z := 0; Y:= R * X
| 6..8 : Z := X + Y - 5
| 9..10: Z := 8; Y:= R * X
END ;
```

```
CASE Day OF
 Sun:              Message:= 'Closed all day';
                   OpenHours:= 0|
 Mon..Wed,Fri:     Message:= 'Normal hours';
                   OpenHours:= 8|
 Thu,Sat:          Message:= 'Half day';
                   OpenHours:= 4
END
```

You can define an empty case if necessary. For example, suppose in the last example you have no statement sequence for the value *Sun*:

```
CASE Day OF
 Sun:              |
 Mon..Wed,Fri:     Message:= 'Normal hours';

                   OpenHours:=                              8|
 Thu,Sat:          Message:= 'Half day';
                   OpenHours:= 4
END
```

Notice that the case of *Sun* explicitly states that no action is performed. When you have code to execute for every value except those in the **CASE** lists, then you can use an ELSE:

```
CASE Day OF
 | Sat : WRITELN('Go for a picnic');
 | Sun : WRITELN('Go shopping');
ELSE
 WRITELN('Go to work')
END ;
```

CHAR standard type   S

Description	CHAR is a standard type with variables that can assume a character as a value. In addition, CHAR may be used as a type-transfer function.

Usage

VAR
 ch: CHAR;
AND
 ch := CHAR(27);

Comments

You can use CHAR-type **variables and constants** wherever **you** want to handle text.

A CHAR variable can take any ASCII character as a value (which are ordered by their ASCII value); for example, the ordinal of character *A* is equal to the ASCII value 65, while the character *B* has an ordinal of 66.

Constants of type CHAR are denoted by a character enclosed within single or double quotation marks, or are octal numbers within the range 0 to 377 followed by a *C*; for example, »9«, 'D', 0C.

You cannot apply arithmetic operators directly on CHAR-type variables, but you can convert the value to CARDINAL and vice versa. To convert a character to a CARDINAL type, you use the standard function *ORD*. To convert a CARDINAL to a CHAR, use the standard function CHR.

Turbo Modula-2 permits the comparison and assignment of strings of characters, and the module *Strings* provides several procedures for string handling.

A CHAR variable occupies 2 bytes in memory unless it is declared as an element of an array, in which case it occupies 1 byte in memory.

See Also   BYTE
                CHR
                ORD
                VAL

Example   **MODULE** Characters;
                 ch: CHAR;
                BEGIN
                 LOOP
                  READ(ch);
                  IF CAP(ch) = 'Q' **THEN** EXIT **END**
                  IF ch < ' ' **THEN** WRITE('.')
                  ELSIF ch > CHR(127) **THEN** WRITE('!')
                  ELSIF (ch >= 'a') **AND** (ch <= 'f') **THEN**
                   WRITE(CAP(ch))
                  ELSE WRITE(ch)
                  END
                 END
                END Characters.

CHR standard function   S
_____

Description   *CHR* returns the character with ordinal *X*.

Usage   Ch:= CHR(X)

                Argument *X* is of type INTEGER or CARDINAL.

                Result *Ch* is of type *CHAR*.

                The argument must be in the range 0 to 255.

Comments   You can use the *CHR* standard function to convert from an
                    ASCII value in the range 0 to 255 to the corresponding
                    character.

                    Note: The characters *A* to *Z* have ASCII values 65 to 90;
                    characters *a* to *z* have ASCII values 97 to 122; numerals 0 to
                    9 have ASCII values 48 to 57.

                    Converting a character to ASCII is performed by the standard
                    function *ORD*.

See Also   CHAR
                    ORD

Example   **PROCEDURE** NumToChar(N: CARDINAL): CHAR;
                    BEGIN
                     IF N < 10 **THEN RETURN** CHR(N + 48)
                     ELSE RETURN CHR(48)
                     END
                    END NumToChar;

ClearEol enumerated value (terminal)   L

Description   *clearEol* is the first value of the enumerated type *Terminal.SpecialOps*.

Declaration   SpecialOps = (clearEol, **insertDelete, highlightNormal**);

Usage   **IF clearEol IN available THEN ClearToEOL END ;**

                    SpecialOps must be imported from the library module *Terminal*.

Comments	If this value is a member of the set *Terminal.available*, then the program may use the procedure *ClearToEOL* to clear to the end of the line.

To use this identifier, include *SpecialOps* in your import list. The identifiers of each of *SpecialOps'*values become visible automatically.

See Also	ClearToEOL available insertDelete highlightNormal Terminal

Example	Clear the right half of the screen:

```
MODULE CheckAvailableOperations;
FROM Terminal IMPORT
                     available,SpecialOps,ClearEOL,
                     GotoXY;
VAR row: CARDINAL
BEGIN
 FOR row := 0 TO 24 DO
  GotoXY(40,row);
  IF ClearEol IN available THEN
  ELSEarToEOL;
   ManualClearEOL;
  END ;
 END ;
 GotoXY(0,0);
END CheckAvailableOperations.
```

ClearScreen procedure (Terminal)   L
_____

Description	*ClearScreen* clears the screen and homes the cursor.

Declaration	**PROCEDURE** ClearScreen;

Usage ClearScreen

ClearScreen must be imported from the library module *Terminal*.

Comments This function will work only if the Turbo Modula-2 **system has** been installed on your terminal.

See Also ClearToEOL
GotoXY
Hightlight
Terminal

Example Clear the **screen:**

```
MODULE Clear;
FROM Terminal IMPORT ClearScreen;
BEGIN
 ClearScreen;
END Clear.
```

ClearToEOL procedure (Terminal)   L

Description *ClearToEOL* clears the screen from the current cursor position until the end of the line.

Declaration **PROCEDURE** ClearToEOL;

Usage ClearToEOL

ClearToEOL must be imported from the library module *Terminal*.

Comments The cursor position is undefined after **a call to this routine;** reposition it with a call to *GotoXY.*

See **Also**   ClearScreen
                      GotoXY
                      Highlight
                      Terminal

Example   Clear the right half of the screen:

```
MODULE ClearHalfScreen;
FROM Terminal IMPORT ClearEOL, GotoXY;
VAR row: CARDINAL
BEGIN
 FOR row := 0 TO 24 DO
  GotoXY(40,row);
  ClearToEOL;
 END ;
 GotoXY(0,0);
END ClearHalfScreen.
```

Close procedure (Files)   L

Description   *Close* disconnects a *FILE* variable from an external disk file
                      and updates the disk directory.

Declaration   **PROCEDURE** Close **(VAR** f: FILE);

Usage   Close(f);

                      Close must be imported from the module *Files*.

                      f must be of type *FILE* imported from *Files*.

Comments   If output has been sent to *f*, closing is mandatory; otherwise,
                      the file size in the directory and the trailer byte will not be up-
                      dated and *f*'s data could be lost.

                      Since closing a file frees up the memory used for data buffer-
                      ing, we advise closing a file in any case, even if it has been
                      used only for input.

See Also	Files Open NoTrailer
Example	Perform minimal file handling (open and close a *FILE*):

```
MODULE OpenAndCloseFile;
FROM Files IMPORT FILE, Open, Close;
VAR oldFile : FILE;
BEGIN
 IF Open(oldFile,'FILE.DAT') THEN
  Close(oldFile)
 END
END OpenAndCloseFile.
```

CloseInput procedure (InOut)   L

Description	*CloseInput* closes an input **file and returns input to the ter-** minal.
Declaration	**PROCEDURE** CloseInput;
Usage	CloseInput *CloseInput* must be imported from the library module *InOut*.
Comments	These routines are provided to maintain compatibility with other Modula-2 implementations (the *Texts* module is easier to work with). *CloseInput* is equivalent to the call *CloseText(input)*, using the *CloseText* procedure from *Texts*.
See Also	OpenInput CloseOutput InOut

Example   **MODULE** Input;

```
FROM InOut IMPORT
 OpenInput, CloseInput, ReadString,
 WriteString, WriteLn;
VAR
 s: ARRAY [0..255] OF CHAR;
BEGIN
(* Prompt user for filename *)
(* If no extention given, use MOD *)
 OpenInput("MOD");
 ReadString(s);        (* Read string from file *)
 WriteString(s);       (* Display it *)
 WriteLn;
 CloseInput;           (* Sever the link to the file *)
 ReadString(s);        (* Input comes from console *)
 WriteString(s);       (* Display it *)
 WriteLn;
END Input.
```

CloseOutput procedure (InOut)   L

Description   *CloseOutput* closes **an output file and returns output to the ter-**
minal.

Declaration   **PROCEDURE** CloseOutput;

Usage   CloseOutput

              CloseOutput must be imported from the library module *InOut.*

Comments   These routines are provided to maintain compatibility with
other Modula-2 implementations (the *Texts* module is easier to
work with).

              CloseOutput results in the call *CloseText(output)*, using the
CloseText procedure in *Texts.*

See Also   OpenOutput
                     OpenInput
                     Inout

Example

```
MODULE Output;
FROM InOut IMPORT
 OpenOutput, CloseOutput, ReadString,
 WriteString, WriteLn;
VAR
 s: ARRAY [0..255] OF CHAR;
BEGIN
 WriteString('Enter output file and data:');
 WriteLn;
 OpenOutput("MOD");
 ReadString(s);     (* Read data from console *)
 (* Place it in the file *)
 WriteString(s); WriteLn;
 CloseOutput;       (* Sever the link to the file *)
 WriteString('The file has been closed.');
 WriteLn;
END Output.
```

CloseText procedure (Texts)   L

Description   *CloseText* disconnects a *TEXT* file variable from an external disk file.

Declaration   **PROCEDURE** CloseText(**VAR** t: TEXT);

Usage   CloseText(t)

              CloseText must be imported from the module *Texts*.

              t must be of type *TEXT* imported from *Texts*.

Comments   If output has been sent to *t*, closing is mandatory; otherwise,
                the file size in the directory and the trailer byte will not be up-
                dated and *t*'s data could be lost.

                Since closing a file frees up the memory used for data buffer-
                ing, we advise closing a file in any case, even if it has been
                used only for input.

                If the external medium is a disk file, the file is closed internal-
                ly using the procedure *Close* from the module *Files*.

                Closing *input* or *output* re-establishes the default connections
                (to CON:).

See Also   OpenText
                TEXT
                Texts
                NoTrailer

Example   Perform minimal text-file handling (open and close a *TEXT*):

```
MODULE OpenAndCloseText;
FROM Texts IMPORT TEXT, OpenText, CloseText;
VAR oldFile : TEXT;
BEGIN
 IF OpenText(oldFile,'T.TXT') THEN
  CloseText( oldFile )
 END
END OpenAndCLoseText.
```

CODE procedure (SYSTEM)   L

Description   *CODE* allows the user to include an assembler routine.

Declaration   **PROCEDURE** CODE(s: **ARRAY OF** CHAR);

Usage   CODE("Any string literal");

                    CODE must be imported from the **pseudomodule** *SYSTEM*.

                    The string must be a literal file name.

Comments   A call to *CODE* must be inserted after a procedure heading;
                    thus replacing the body of the procedure. No declaration part,
                    BEGIN, or any other statements may be present in its body;
                    however, the final **END** must be included.

                    Compilation of a *CODE* statement causes the ».COM«-File
                    whose name is given as argument to be inserted into the object
                    program. The length of the file must be in the first 2 bytes of
                    the file.

                    A *CODE* procedure may use all Z80 registers **without restor-**
                    ing them.

See Also   SYSTEM

Example   Include MOVE.COM written in assembler:

                    PROCEDURE MOVE(source: ADDRESS; n: CARDINAL);
                    CODE("MOVE")   (* MOVE.COM is inserted *)
                    END MOVE;

Col function   (Texts)

---

  scription   *Col* returns the current column position of a text file.

Declaration   **PROCEDURE** Col(t: TEXT): CARDINAL;

Usage   currentCol := Col(t)

                 Col must be imported from the library module *Texts*.

                 currentCol must be of type CARDINAL.

                 t must be of type *TEXT* imported from *Texts*.

Comments   After a *WriteLn* or a *CreateText*, the column position is zero;
                 it increases by one with every written character.

See Also   SetCol
                 Texts

Example   Start a new line if the current column position of *TEXT t* is
                 greater than 79.

                 IF Col(t) > 79 THEN WriteLn(t) END

ComLine module   L

Description   *ComLine* enables command line processing (arguments
                 and redirection).

Declaration   DEFINITION MODULE ComLine;
                 FROM Texts IMPORT Text;

                 VAR
                  commandLine: Text;
                  inName, outName: ARRAY [0..19] OF CHAR;
                  progName: ARRAY [0..7] OF CHAR;

                 PROCEDURE RedirectInput;
                 (* Input redirection as ordered on command line. *)

                 PROCEDURE RedirectOutput;
                 (* Output redirection as ordered on command
                 line.*)

```
PROCEDURE PromptFor(Prompt: ARRAY OF CHAR;
                    VAR  S: ARRAY OF CHAR);
(* Prompts for command line. *)
END ComLine.
```

The *TEXT* variable *commandLine* contains any arguments written after the command that started the program. It can be read like a normal *TEXT.*

inName contains the input medium, which is normally CON:.

outName contains the output medium, which is normally CON:.

progName contains the program name.

Comments

The command line consists of any characters or arguments that follow the command that started the program. The contents of the command line are accessed through the *TEXT commandLine.*

The command line can contain *redirection arguments*:

> FileName   Redirects the standard text *Output* to *FileName.* The new medium *FileName* is stored in *outName.*

< FileName   Redirects the standard text *Input* to *FileName.* The new medium *FileName* is stored in *inName.*

The redirection symbols and file names do not appear in the command line.

Note that if there is no redirection argument, *inName* and *outName* contain the name of the standard medium CON:.

See Also

Texts
Also refer to each identifier in the definition module.

Example   If you start a program by typing

```
MyProg Name
```

the *TEXT commandLine* will **contain the word Name.** You would access this like so:

```
READ(commandLine,InputFilename);
```

If you write a program to compare two disk files, you will want to specify the file to be compared on the command line.

By using *ComLine* procedures you can pass the file names directly to the program. Your program will include the following lines:

```
MODULE Compare;
FROM Files IMPORT FILE, Open,Close;
FROM ComLine IMPORT
 commandLine, RedirectInput, RedirectOutput;
VAR
 Filename1, Filename2: ARRAY [0..14] OF CHAR;
 f1,f2: FILE;
BEGIN
 RedirectInput; RedirectOutput;
 READ(commandLine,Filename2);
 IF Open(f2,Filename2) THEN
  READ(commandLine,Filename1);
  IF Open(f1,Filename1) THEN
   READLN(options);
   (* compare the files *)
   (* Write differences to standard output *)
  END
 END ;
 Close(f1);
 Close(f2);
END Compare.
```

For example, you can **start a program** and define **a comm**and line by typing

```
Compare A:testfile B:testfile <B:Compare.Ops
        >A:Compared.dif
```

This line invokes the compare program with two files as arguments. Any input requested from the console is redirected from the keyboard to the file COMPARE.OPS. The output of the program is redirected from the screen to the file COM-PARED.DIF.

---

commandLine variable (ComLine)   L

Description   *commandLine* is Predefined text that contains arguments passed on the command line when a program is invoked.

Declaration   **VAR**
               commandLine: TEXT;

               TEXT is imported from the module *Texts*.

Usage   READ(commandLine,filename);

              filename is a string variable

Comments   Treat *commandLine* like any *TEXT* variable. It contains anything typed on the command line after the program name, up to 80 characters long.

See Also   PromptFor
              ComLine

Example   **MODULE** OffTheCommandLine;
                 FROM ComLine **IMPORT** commandLine;
                 VAR
                  x,y: REAL;
                 BEGIN
                  READ(commandLine,x,y);
                  WRITELN(x:5:2,'*',y:5:2,' = ',x*y:5:2);
                 END OffTheCommandLine.

ConnectDriver procedure (Texts)   L

Description   *ConnectDriver* allows the user **to install her own** input **and ot**
                 put drivers.

Declaration   **TYPE**
                  TextDriver = **PROCEDURE**E(TEXT, **VAR** CHAR);
                 PROCEDURE ConnectDriver(**VAR** t: TEXT; p: **TextDriver**

Usage   ConnectDriver(t, driver)

                 t must be of type *TEXT* imported from the library **mod**
                 Texts.

                 Driver must be compatible **with the** type *TextDriver.*

Comments   Allows users to install their own read and write charac
                 routines, which are then used by all I/O statements in the st
                 dard module *Texts*, including the extensions *READ, WRI*
                 READLN, and *WRITELN*.

See Also   Texts

Example   Connect the user-defined driver to the text *t*:

```
MODULE AlwaysUpCase;
FROM Texts IMPORT TEXT, output, ConnectDriver;

PROCEDURE WriteUpcase(t: TEXT; VAR ch: CHAR);
BEGIN
 Write(output,CAP(ch));
END WriteUpcase;

VAR t: TEXT;
BEGIN
 ConnectDriver(t, WriteUpcase);
 WRITELN(t,'This will be displayed all uppercase');
END AlwaysUpCase.
```

---

console (Texts)   L

Description   An auxiliary output *TEXT* attached to the console.

Declaration   `VAR console: TEXT;`

Usage   `WRITELN(console, 'output to the console');`

                    console must be imported from the library module *Texts* and can be used anywhere an output *TEXT* can be used.

Comments   *console* is always open and, by default, is connected to the terminal. It is as if the call *CreateText(console, »CON:«)* has taken place before the execution of the main module.

                    console is useful if the *output* text stream has been redirected but you still want certain messages (prompts, errors, and so on) to go to the terminal. The *console* cannot be redirected.

See Also   output
                 input
                 Texts

Example   Write out an error message by passing redirection:

```
MODULE consoleOutput;
FROM Texts IMPORT console;
VAR recnum: CARDINAL;
BEGIN
 (* processing loop *)
 WRITE(console, 'Error at record number: ', recnum);
END consoleOutput.
```

CONST declaration   R

Description   The reserved word **CONST** precedes the declaration of con-
                   stants.

Usage   ```
CONST
 Pi   = 3.14159265;
 TwoPi = 2.0*Pi;
```

Comments   Modula-2 allows the use of constants in the definition of fur
                   ther constants, like the *Pi* in *TwoPi*.

Example   ```
MODULE Constants;
CONST
 nThings = 100;
 LastThing = nThings-1;
TYPE
 ThingRange = [0..LastThing];
VAR
 Things : ARRAY ThingRange OF CHAR;
 which   : [0..lastthing];
 thingy : CARDINAL;
BEGIN
 FOR thingy := 0 TO lastthing DO
  Process(things[which])
 END
END Constants.
```

Convert module      L

| | |
|---|---|
| Description | *Convert* is a standard module that you **can use when converting** between numeric types and strings. |
| Declaration | **DEFINITION MODULE** Convert; |

```
PROCEDURE StrToInt    (VAR s: ARRAY OF CHAR;
                          VAR i:INTEGER) :BOOLEAN;
PROCEDURE StrToCard   (VAR s: ARRAY OF CHAR;
                          VAR c:CARDINAL) :BOOLEAN;
PROCEDURE StrToLong   (VAR s: ARRAY OF CHAR;
                          VAR l:LONGINT) :BOOLEAN;
PROCEDURE StrToReal   (VAR s: ARRAY OF CHAR;
                          VAR r:REAL) :BOOLEAN;

PROCEDURE  IntToStr (i:INTEGER
                    VAR s: ARRAY OF CHAR);
PROCEDURE CardToStr (c:CARDINAL
                    VAR s: ARRAY OF CHAR);
PROCEDURE LongToStr (l:LONGINT
                    VAR s: ARRAY OF CHAR);
PROCEDURE RealToStr (r:REAL
                    VAR s: ARRAY OF CHAR;
                        digits:INTEGER);

EXCEPTION  TooLarge;

END Convert.
```

Usage	See individual identifiers
Comments	The first four procedures convert **strings to numeric types; the** next four reverse the conversions.
	The exception TooLarge is raised in the last four procedures if the number is too large to fit in the target string. (TooLarge is only raised by the second four procedures.)
See Also	The module *Doubles* provides the same functions for double-precision variables (LONGREAL).

Copy procedure (Strings)   L

Description	*Copy* copies a substring starting at a given position with given length of a string.

Declaration   **PROCEDURE** Copy(**VAR** str: **ARRAY OF** CHAR;

```
            inx,len: CARDINAL;
        VAR result: ARRAY OF CHAR);
```

Usage   Copy(string, from, length, resultstring)

Copy must be imported from the library module *Strings*.

string and *result* must be character array variables.

from and *length* must be of type CARDINAL.

Comments   A string is terminated by a zero byte. If a zero byte is countered before *length* characters have been processed, copy will be prematurely interrupted.

See Also   Concat
Delete
Insert
Strings

Example   Copy 'holes' into *holestring*:

```
Copy( 'Donut holes', 6, 5, holestring )
```

Cos function (MathLib, LongMath)   L

Description	*Cos* returns the cosine of *X*, where *X* is expressed in radians.

Declarations

```
PROCEDURE Cos(x: REAL): REAL;
PROCEDURE Cos(x: LONGREAL): LONGREAL;
```

Usage

```
Y:= Cos(X)
```

Argument *X* and result *Y* must be of the same type--either both REAL or both LONGREAL.

Comments

If you want the argument expressed in degrees instead of radians, use the following expression:

```
Result:= Cos(X * 3.141592/180.0)
```

If you need to calculate the inverse of cosine, you can use the *ArcTan* function in the following expression:

```
ArcCos:= Arctan(X/Sqrt(1.0 - X*X) ) - 3.141592/2.0
```

Example

```
MODULE Cosine;
FROM MathLib IMPORT Cos;
CONST
 DegsToRads = 3.141592/180.0;
VAR
 Degs,Radians,CosAngle:REAL
BEGIN
 Radians:= Degs * DegsToRads;
 CosAngle:= Cos(Radians);
END Cosine.
```

Create procedure (Files)   L

Description	*Create* creates a file on disk with a given **name.**
Declaration	**PROCEDURE** Create(**VAR** f: FILE; name: **ARRAY OF** CHAR);
Usage	Create(f, filename)

Create must be imported from the library module *Files.*

f must be declared as *FILE.*

FILE must be imported from the library module *Files.*

filename must be declared as **ARRAY OF CHAR.**

Comments   *filename* must be a legal CP/M filename, and there must be room on the directory and the disk for a new file.

f can only be used for output after being opened with *Create*; use *Open* for reading a file.

If a file is already present, it will **be overwritten, so use caution.**

After writing to this file, close it to update the file size in the directory and the file trailer byte. Failure to do this will probably result in the loss of all written information.

See Also   Close
                  Files
                  Open
                  NoTrailer

Example   Create an empty file on disk:

```
MODULE CreateAnEmptyFile;
FROM Files IMPORT FILE, Create, Close;
VAR
 newFile :FILE;
BEGIN
 Create(newFile,'FILE.DAT');
 Close(newFile);
END CreateAnEmptyFile.
```

CreateText procedure (Texts)   L

Description   *CreateText* associates a *TEXT* with an external medium and opens it for writing.

Declaration   **PROCEDURE** CreateText**(VAR**
                 t: TEXT; name: **ARRAY OF** CHAR);

Usage   CreateText(t, fname)

            CreateText must be imported from library module *Texts*.

            t must be of type *TEXT* imported from *Texts*.

            fname must be of type **ARRAY OF CHAR**.

Comments   *fname* must be a legal CP/M file name or device, such as »MYFILE.DAT« or »CON:«.

            t can only be used for output after being opened with *CreateText*; use *OpenText* for reading a *TEXT* file.

            If a file is already present it will be overwritten, so use caution.

            After writing to this file, close it to update the file size in the directory and the file trailer byte. Failure to do this will probably result in the loss of all written information.

See Also   CloseText
                  OpenText
                  Texts
                  NoTrailer

Example   Create an empty text file on disk:

```
MODULE CreateAnEmptyText;
FROM Texts IMPORT TEXT, CreateText, CloseText;
VAR
 newFile: TEXT;
BEGIN
 CreateText(newFile,'EMPTY.TXT');
 CloseText(newFile);
END CreateAnEmptyText.
```

DeadLock exception (Processes)   L/E

Description   DeadLock is raised in the module *Processes* when all pi
                  cesses are waiting for a signal.

Declaration   **EXCEPTION**
                  DeadLock;

Usage   **EXCEPTION**
                  DeadLock:
                   WRITELN('Deadly embrace: all processes
                  terminated.');
                  END

                  DeadLock must be imported from the module *Processes*.

Comments   When a dead lock occurs, your signal handling logic is fal
                  You can detect this problem by using this exception.

See Also   Processes
                  SYSTEM

Example   **MODULE** Signals;

```
           FROM Processes IMPORT Init, SIGNAL, WAIT, DeadLock;
           VAR s: SIGNAL;
           BEGIN
            Init(s); WAIT(s);
           EXCEPTION
            DeadLock: HALT;
           END Signals.
```

DEALLOCATE procedure (STORAGE)   L

Description   *DEALLOCATE* deallocates a block of memory on the heap that had been previously associated with a pointer variable.

Declaration   **PROCEDURE** DEALLOCATE**(VAR**
a: ADDRESS; size: CARDINAL);

Usage   DEALLOCATE(a, size)

    DEALLOCATE must be imported from the library module *STORAGE*.

    a must be compatible with type *ADDRESS* (any pointer).

    size must be of type CARDINAL.

Comment   You can only deallocate portions of memory that you have previously allocated. The variables *a* and *size* must be the same location and amount as when they were allocated.

    The pointer *a* will be set to *NIL* and the deallocated area will be available for future allocation.

See **Also**	ALLOCATE
	NEW
	DISPOSE
	STORAGE
	MARK
	RELEASE
	FREEMEM

Example   Allocate and deallocate 100 bytes:

```
MODULE HeapMem;
FROM STORAGE IMPORT ALLOCATE, DEALLOCATE;
VAR
 a: POINTER TO ARRAY [0..99] CHAR;
BEGIN
 ALLOCATE(a,100);
 a^ := 'This string is on the heap';
 DEALLOCATE(a, 100);
 (* Now the string is gone. *)
END HeapMem.
```

DEC standard procedure   S
_____

Description   *DEC* returns a **predecessor of argument** *X*.

Usage   DEC(X)

                 or

                 DEC(X,N)

                 Argument *X* may be of type INTEGER, CARDINAL, **CHAR**,
                 BOOLEAN, or a user-defined scalar type.

                 The second form returns the *N*th predecessor of *X*.

Comments   If you subtract 1 from a CARDINAL value, the result is the value prior to the original value. With *DEC*, you can do exactly the same thing with any ordinal type.

When decrementing a value, you may reach its *MIN* value; for example, the *MIN(CARDINAL)* is 0. You may choose whether or not this situation will generate an error by using the overflow compiler switch. This can be done globally or locally. Thus with overflow checking turned off, as in

```
VAR
 i: CARDINAL;
BEGIN
 i := MIN(CARDINAL);
 (*$O-*)
 DEC(i);
 (*$O+*)
 WRITELN(i);
END
```

the *i* would wraparound to *MAX(CARDINAL)* and no error will be generated. If overflow checking has been turned on, then the exception OVERFLOW will be raised. This can be trapped as follows:

```
MODULE dec;
FROM SYSTEM IMPORT OVERFLOW;
VAR
i: CARDINAL;
BEGIN
 (*$O+*)
 i := MIN(CARDINAL);
 DEC(i);
EXCEPTION
 OVERFLOW: WRITELN('Trapped overflow error');
END dec.
```

See Also   INC

Example   For **example, suppose you** have **an enumeration** variable:

                   VAR Energy: (Kinetic,Potential,Heat);

                   You can assign any of the three values to the variable *Energy.*
                   Now suppose you want to decrement the value of *Energy* to a
                   lower value; for example, to the value preceding the current
                   value:

                   DEC(Energy)

                   You have performed a subtraction operation on the enumera-
                   tion variable.

                   The procedure **can** operate on any ordinal type.

                   MODULE Decrement;
                   VAR
                    i: INTEGER;
                    state: BOOLEAN;
                   BEGIN
                    i:= 0;
                    DEC(i,2);
                    (* now i= -2*)
                    state:= TRUE;
                    DEC(state);
                    (* now state is **FALSE***)
                   END Decrement.

DEFINITION declaration   R

Description   **DEFINITION** specifies to the compiler that a compilation
                   unit is the definition part of a library module.

Usage   **DEFINITION MODULE** myMod

Comments   The definition module is the »visible« part of a library module that can be imported into other modules. In the definition module *myMod* you will find all the declarations necessary to use the facilities of *myMod*.

Every definition module will have a corresponding implementation module that contains the hidden parts of that module. It will contain all executable code, as well as those definitions relevant only to the internal operation of the module.

All identifiers mentioned in a definition module are exported. The identifiers are exported in qualified mode; although, in most cases, they will be dequalified using the **FROM** clause in the module that imports them.

There is no code in a definition module. The compiler simply produces a symbol table file that can later be used to do type-checking.

Definition modules usually reside in files with the extension .DEF. Compilation results in the creation of a file with the extension .SYM.

The definition module cannot have the compilation unit in the same file; thus definition modules and implementation modules must be in separate files. The resulting .SYM and .MCD files can be combined, however, into a .LIB file using the Turbo Modula-2 librarian.

See Also   IMPLEMENTATION
MODULE

Example   A simple user-definition module in file 'SIMPLE.DEF':

DEFINITION MODULE SimpleMath;

PROCEDURE ADD(y,z :INTEGER): INTEGER;

END SimpleMath.

The corresponding implementation module in file 'SIM-PLE.MOD':

IMPLEMENTATION MODULE SimpleMath;

PROCEDURE ADD(y,z :INTEGER) :INTEGER;
BEGIN
 RETURN (y+z);
END Add;

END SimpleMath.

DELETE procedure (Files)   L

Description   *Delete* erases a file from disk.

Declaration   **PROCEDURE** Delete(**VAR** f: FILE);

Usage   Delete(f)

                   Delete must be imported from the library module *Files*.

                   f must be an open file of type *FILE* imported from *Files*.

Comments   The file must have already been successfully opened.

                   The file is closed when it is deleted, thus its buffers are freed.

See Also	Close Files Open Rename
Example	Delete the file 'SCRATCH.SCR' from disk:

```
MODULE deleteFile;
FROM Files IMPORT FILE, Open, Delete;
VAR f: FILE;
BEGIN
 IF Open(f,'SCRATCH.DAT') THEN Delete(f) END
END deleteFile.
```

DELETE procedure (Strings)   L

Description	*Delete* deletes a substring from a string.
Declaration	**PROCEDURE** Delete(**VAR** str: **ARRAY OF** CHAR; inx,len: CARDINAL);
Usage	Delete(str, from, length)
	Delete must be imported from the library module *Strings*.
	from and *length* must be of type CARDINAL.
	str must be a character array variable.
Comments	The counting starts from 0, so the user's strings should be declared starting from 0.
See Also	Copy Insert Pos Strings
Example	*Delete(»Lover«, 0, 1)* will yield the string *»over«*.

DeleteLine procedure (Terminal)   L

Description	*DeleteLine* deletes the cursor line completely and to fill the gap moves up all following lines by one.
Declaration	**PROCEDURE** DeleteLine;
Usage	DeleteLine;
	DeleteLine must be imported from the library module *Terminal*.
Comments	The current cursor line will be deleted. (This can be set by the procedure *GotoXY* in *Terminal*.) Since the cursor position on a screen is undefined after certain operations, the only way to determine the current cursor position is by keeping track of it yourself.
	This function will work only if the Turbo Modula-2 system has been installed on your terminal.
	A program can determine if the option has been installed by examining the set variable *Available* exported by *Terminal*. If *insertDelete* IN *Available* is TRUE, then these functions are available as installed on the current terminal.
	See *GotoXY* for a map of the screen coordinates.
See Also	GotoXY
	InsertLine
	Terminal
Example	Delete the **twelfth line from the** top:

```
GotoXY(0,11);
DeleteLine;
```

DeviceError exception (Files)   L/E

Description	DeviceError is raised by the *Files* module when a write operation on a file is impossible because the disk has a bad sector.

Declaration   **EXCEPTION**
  DeviceError;

Usage   **EXCEPTION**
  DeviceError: WRITELN('File error');
END

DeviceError must be imported from the module *Files*.

Comments   Use the DeviceError exception to prevent your program from crashing when the program encounters a bad disk.

See Also   EndError
               StatusError
               UseError
               DiskFull

Example
```
MODULE WriteFile;
FROM Files IMPORT
 FILE, Create, WriteWord, Close, EndError,
 StatusError, UseError, DeviceError;
VAR f: FILE; c: CARDINAL;
BEGIN
 Create(f,'testfile.dat');
 WriteWord(f,c);
 Close(f)
EXCEPTION
 EndError:  WRITELN('End  of  file  reached.');
 Close(f)
| StatusError: WRITELN('Error file not opened');
| UseError : WRITELN('Disk not logged in.');
| DeviceError: WRITELN('Error writing disk:
  bad sector');
END WriteFile.
```

DiskFull exception (Files)   L/E

Description	DiskFull is raised by the *Files* module when a write or close operation on a file is impossible because the disk or directory is full.
Declaration	**EXCEPTION** DiskFull;
Usage	**EXCEPTION** DiskFull: WRITELN('File error'); **END** DiskFull must be imported **from the module** *Files*.
Comments	Use the DiskFull exception to prevent your program from crashing when the program encounters a full disk.
See Also	EndError StatusError UseError DeviceError

Example   **MODULE** WriteFile;
           FROM Files **IMPORT**
           FILE, Create, WriteWord, Close, EndError,
           StatusError, UseError, DeviceError, DiskFull;
           VAR f: FILE; c: CARDINAL;
           BEGIN
           Create(f,'testfile.dat');
           WriteWord(f,c);
           Close(f)
           EXCEPTION
              EndError: WRITELN('End of file reached.');
              Close(f)
           | StatusError: WRITELN('Error file not opened');
           | UseError : WRITELN('Disk not logged in.');
           | DeviceError: WRITELN('Error writing disk: bad
              sector');
           | DiskFull: WRITELN('This disk is FULL, buddy!');
           END WriteFile.

DISPOSE standard procedure (STORAGE)   S

Description   *DISPOSE* deallocates a dynamic variable from the heap.

Usage   DISPOSE(p)

        or

        DISPOSE(p, tag1, tag2, ... tagn)

To use this procedure, *DEALLOCATE* must be imported from the library module *STORAGE*, or it must be defined in some other way.

p must be a pointer to a variable.

tag1,... tagn are relevant if the type of the variable is a tagged variant record.

Comments   *DISPOSE* is essentially an abbreviated *DEALLOCATE* command. If, for example, *p* is of type **POINTER TO** LONGREAL, then *DISPOSE(p)* will be equivalent to *DEALLOCATE(p, TSIZE(LONGREAL))*.

p must point to a variable that has been previously allocated using either *ALLOCATE* or *NEW.*

Care must be taken when using *DISPOSE* and **DEALLOCATE** together with *MARK* and *RELEASE.*

When variant records are allocated using a tag field, **you** should use the same tag when disposing of that record.

See Also   DEALLOCATE
MARK
NEW
RELEASE
STORAGE

Example   Allocate a variable and use it:

```
MODULE Dispose;
VAR
 p1, p2, p3: POINTER TO REAL;
BEGIN
 NEW(p1);
 NEW(p2);
 NEW(p3);
(* use them *)
 DISPOSE(p2);
 DISPOSE(p3);
 DISPOSE(p1)
END Dispose.
```

DIV standard operator   R

Description	DIV is an integer division operator; it is a binary operator that works on operands of the same type. The operands may be both CARDINAL, both INTEGER, or both LONGINT.

Usage

```
i := i DIV j;
```

where *i* and *j* are of the same type.

The result of the expression is the same type as the operands.

Comments   Integer division is useful when you do not care about the fractional parts of a quotient.

See Also   MOD

Example
```
MODULE IntegerDivision;
VAR
 i,j : INTEGER;
 c,d : CARDINAL;
 l,m : LONGINT;
BEGIN
 i := 1; c := 1;
 i := 30 DIV i;
 c := d DIV c;
 l := m DIV 344212L;
END IntegerDivision.
```


Done procedure (InOut)   L

Description   *Done* checks for successful completion of input.

Declaration   **VAR** Done: BOOLEAN;

Usage   **IF** Done **THEN** DoSomething **END** ;

Done must be imported from the library module *InOut*.

Comments   *Done* allows a program to check for valid input before pro-
                    ceeding. Each of the input procedures in *InOut* sets *Done* as
                    follows:

                    Read   Done := Not past end of input
                    ReadString   Done := The returned string is not empty
                    ReadInt   Done := Integer was read
                    ReadCard   Done := Cardinal was read

See Also   Texts

Example   Get a number from the user:

                    MODULE ReadInteger;
                    FROM InOut **IMPORT** Done, ReadInt;
                    BEGIN
                     REPEAT
                      WRITE('Gimme a number> ');
                      ReadInt(i)
                     UNTIL Done;
                    END ReadInteger.

Done procedure (Texts)   L

Description   *Done* checks for successful completion of numeric input.

Declaration   **PROCEDURE** Done(t: TEXT): BOOLEAN;

Usage   okay := Done(t)

                    Done must be imported from the library module *Texts*.

                    t must be of type *TEXT* imported from the library module
                    Texts.

Comments   *Done* allows the program to check for valid numeric input
                  before proceeding.

                  Done is a function returning TRUE if the numeric input is both
                  syntactically correct and the input number is in the valid range
                  for the type being read; otherwise FALSE is returned.

See Also   Texts

Example   Get a number from the user:

```
MODULE DoneInput;
FROM Texts IMPORT Done;
BEGIN
 REPEAT
  WRITE('Gimme a number> ');
  READ(num);
 UNTIL Done(input);
END DoneInput.
```

DOUBLE standard function   S

Description   *DOUBLE* converts the argument to type LONGREAL.

Usage   Y:= DOUBLE(X)

                  Argument *X* is of type CARDINAL, INTEGER, LONGINT,
                  or REAL.

                  The result *Y* is of type **LONGREAL**.

Comments   Do not confuse this function with the LONGREAL type-
                  transfer function, which transfers data without converting it.

See Also   FLOAT
                  LONGREAL
                  TRUNC

Example   **MODULE** Doubles;
                 VAR
                  Converted: LONGREAL;
                  r: REAL;
                 BEGIN
                  r := 234.234;
                  Converted := DOUBLE(r);
                 END Doubles.

Doubles   L

Description   *Doubles* is a library module that provides string handling and
                 I/O support for double-precision variables.

Declaration   **DEFINITION MODULE** Double
                 FROM Texts IMPORT TEXT;

                 VAR
                  legal :BOOLEAN

                  PROCEDURE ReadDouble (t: TEXT; **VAR** d: LONGREAL);
                  PROCEDURE WriteDouble (t: TEXT; d: LONGREAL;
                    width: CARDINAL; digits: INTEGER);

                  PROCEDURE StrToDouble(**VAR** s: **ARRAY OF CHAR;**
                    d: LONGREAL);
                  PROCEDURE DoubleToStr (s: **ARRAY OF** CHAR;
                    VAR d: LONGREAL);

                 END Doubles.

Comments   This module provides conversions for LONGREALs, which
                 are similar to the ones that *Convert* and *Texts* provide for the
                 standard types in Modula-2.

DoubleToStr procedure (Doubles)   L

Description	*DoubleToStr* converts a double-precision variable to a string variable.

Declaration   **PROCEDURE** DoubleToStr
(d: LONGREAL; **VAR** s: **ARRAY OF** CHAR;
    digits: INTEGER);

Usage   DoubleToStr(d, s, digits)

DoubleToStr must be imported from the library module *Doubles*.

s must be a character array variable.

d must be of type LONGREAL.

See Also   Doubles
               StrToDouble

Example   Convert a double-precision number into a string:

d := 2.99D10;
DoubleToStr(d, str, 15);

END   R

Description   END marks the end of the scope of a control structure, procedure, or module.

Usage   **END**

               or

               END pname

Comments	In the case of a control structure (IF, WITH, WHILE, or CASE), the END stands alone. In the case of a module or procedure, the END statement is followed by the name of the module of procedure it terminates. A semicolon is never needed before an END.

Example

```
MODULE endexample;
BEGIN  (* Main *)
  IF appropriate THEN DO
   WITH SomeRecord DO
        (* code *)
   END
   END
END endexample.
```

EndError exception (Files)   L/E

Description	EndError is raised by the *Files* module when an attempt is made to read past the end of the file.

Declaration

```
EXCEPTION
  EndError;
```

Usage

```
EXCEPTION
  EndError: WRITELN('END OF FILE');
END
```

EndError must be imported from the module **Files.**

Comments	Use the EndError exception to prevent your program from crashing from user input.

See Also

StatusError
UseError
DeviceError
DiskFull

Example
```
MODULE ReadFile;
FROM Files IMPORT FILE, Open, ReadWord,
Close, EndError;
VAR f: FILE; c: CARDINAL;
BEGIN
 IF Open(f,'testfile.dat') THEN
  LOOP
   ReadWord(f,c);
  END
 END ;
EXCEPTION
 EndError: WRITELN('End of file reached.'); Close(f)
END ReadFile.
```

Entier function (MathLib, LongMath) · L

Description
: *Entier* returns the integer part of a real number rounded toward negative infinity.

Declaration
```
PROCEDURE Entier(x: REAL): INTEGER;
PROCEDURE Entier(x: LONGREAL): LONGINT;
```

Usage
: `Y:= Entier(X)`

Argument X is REAL or LONGREAL.

Result Y is INTEGER or LONGINT.

Comments
: The Turbo Modula-2 predefined function *INT* can convert any numeric type to INTEGER; for example:

```
Entier(-3.5) = -4
INT(-3.5) = -3
```

Example   **MODULE** EntierTest;

```
IMPORT MathLib;
IMPORT LongMath;
VAR
 Round: INTEGER;
 Number: REAL;
 LongRound: LONGINT;
 LongNumber: LONGREAL;
BEGIN
 Round:= MathLib.Entier(Number + 0.5);
 LongRound := LongMath.Entier(LongNumber + 0.5
END EntierTest.
```

EOF function (Files)   L

Description   *EOF* returns true if the file pointer is at the end of tl

Declaration   **PROCEDURE** EOF(f: FILE): BOOLEAN;

Usage   endOF := EOF(f)

          EOF must be imported from the library module *File:*

          f must be of type *FILE* imported from *Files*.

Comments   This is the normal way to detect the end of defined dat
          a file.

See Also   Close
          Files
          Open

Example   Open a file and process it:

```
MODULE EndOfFile;
FROM Files IMPORT FILE, Open, ReadRec, EOF;
VAR
 f: FILE;
 data : ARRAY [0..99] OF REAL;
BEGIN
 IF Opn(f,"Numbers.dat") THEN
  WHILE NOT EOF(f) DO
   ReadRec(f,data);
   (* Code to process data *)
  END
 END
END EndOfFile.
```

EOL constant (Texts)   L

Description   *EOL* holds the value of the end-of-line character, 36C or 30 decimal.

Declaration   **CONST**
                   EOL = 36C;

Usage   ch := EOL;

                   ch must be defined as a **CHAR**.

                   EOL must be imported from the module *Texts*.

Comments   The procedures in the module *Texts* convert the carriage-return/line-feed sequence to and from the *EOL* character. When input, the carriage-return/line-feed sequence is converted to an *EOL* character; and when output, the *EOL* character is converted to a carriage-return/line-feed pair. Thus in a program-processing text file, the end of line is marked with the *EOL* character.

See Also   Texts

Example   **MODULE** EndOfTheLine;
                  FROM Texts **IMPORT** EOL;
                  VAR ch: CHAR;
                  BEGIN
                   REPEAT
                    READ(ch)
                   UNTIL ch=EOL;
                  END EndOfTheLine.

EOLN procedure (Texts)   L

Description   *EOLN* returns TRUE if the *TEXT* file is at the end of a line.

Declaration   **PROCEDURE** EOLN(t: TEXT): BOOLEAN;

Usage   endOL := EOLN (t)

                  EOLN must be imported from the library module *Texts*.

                  t must be of type *TEXT* imported from *Texts*.

                  endOL must be of type BOOLEAN.

Comments   *EOLN* returns TRUE if the last character read from *t* is an *EOL*
                  character.

See Also   EOT
                  Texts

Example   Read all characters until the end of the line is encountered:

                  REPEAT
                   ReadChar(t,ch);
                  UNTIL EOLN(t);

EOT procedure (Texts)   L

Description   *EOT* returns TRUE if the *TEXT* file is at its end.

Declaration   **PROCEDURE** EOT(t: TEXT): BOOLEAN;

Usage   EndOfText := EOT (t)

EOT must be imported from the library module *Texts*.

t must be of type *TEXT* imported from *Texts*.

EndOfText must be of type BOOLEAN.

Comments   *EOT* returns TRUE if the last character read from *t* is an *EOT* character (for example, Control-Z or 32C).

Notice there is no *EOT* character defined in Turbo Modula-2; it is reserved for the function described here. Note that sometimes the 32C character is referred to as *EOT*. Also note that the EOT character on other systems may not be 32C.

See Also   EOF
                 EOLN
                 Texts

Example   Read all **characters until the end of the** *TEXT* file *t* is encountered:

```
REPEAT
 ReadChar(t,ch);
UNTIL EOT(t)
```

EXCEPTION   R

Description   **EXCEPTION** is used to declare exception identifiers **and** declare code to trap exceptions when they occur.

Usage   Exception Declarations

```
MODULE deviceHandler;
EXCEPTION ex1, ex2, ex3; (* Exception declaration *

PROCEDURE dev1;
BEGIN
(* code with RAISEs *)
END dev1;

END devicehandler;
```

Exception Handlers

```
MODULE deviceMonitor;
IMPORT ex1,ex2,ex3;
BEGIN
 dev1; (* Invoke a procedure with RAISEs *)
EXCEPTION
 | ex1:    (* ex1  code *)
 | ex2:    (* ex2  code *)
 | ex3:    (* ex3  code *)
 ELSE     (* other code *)
END deviceMonitor.
```

The first module declares exceptions and code that will ra the exceptions in error situations. The second module call procedure that may raise an error; therefore, it declares exc tion handlers to catch and process the errors.

Comments	Exceptions are particularly useful in the event of a program crash; for instance, if you need to turn off a laser, or switch out of graphics mode. It is possible to do this with the Turbo Modula-2 system, because the runtime system can take over (in most cases) when something catastrophic happens to your logic. The **ELSE** clause in the exception handler catches all possible exceptions whether imported or not.

See Also	RAISE

Example	Declare *PowerFailureImminent* as an exception:

EXCEPTION PowerFailureImminent

Then raise it:

IF ((* some hint *)) **THEN RAISE**

PowerFailureImminent;

Then handle it:

EXCEPTION
 PowerFailureImminent: WRITE('Help !)';
END someMainModule;

EXCL standard procedure   S

Description	*EXCL* excludes **element *I* from the set *S*.**
Usage	EXCL(S,I)
	Argument *S* is of **set type.**
	Argument *I* is of the set's base type.
Comments	This standard procedure allows you to remove an element from a set. This has the effect of turning off a bit in the word that represents the set.

See Also	BITSET INCL SET
Example	**MODULE** Exclude; **TYPE** Day = (Sun,Mon,Tues,Wed,Thu,Fri,Sat); **VAR** Work: **SET OF** Day; **BEGIN** Work:= Day{Mon..Fri}; (* Set Work contains five **elements Mon to Fri** *) EXCL(Work,Fri); (* Set Work contains four elements, Wed excluded **END** Exclude.

EXIT statement   R

Description	**EXIT** enables you to exit from a **LOOP** statement.
Usage	**EXIT**
Comments	**EXIT** jumps to the statement following the **END of the end** ing **LOOP** statement.
See Also	LOOP
Example	**LOOP** IF DoneProcessing **THEN** EXIT **END** ; **END** ; HALT

ExitScreen procedure (Terminal)   L

Description	*ExitScreen* sends a terminal reset string after a series of screen operations, if needed.
Declaration	**PROCEDURE** ExitScreen;
Usage ·	ExitScreen;
	ExitScreen must be imported from the library module *Terminal*.
Comments	This function will work only if the Turbo Modula-2 system has been installed on your terminal.
	Many terminals don't need or have a reset string. If this is the case with your terminal, the function will do nothing.
Example	Send a terminal reset string:

```
BEGIN
 (* Lots of screen operations *)
 ExitScreen
END
```

Exp function (MathLib, LongMath)   L

Description	Exp returns the natural exponential of X.
Declaration	**PROCEDURE** Exp(x: REAL): REAL; **PROCEDURE** Exp(x: LONGREAL): LONGREAL;
Usage	Y:= Exp(X)
	Argument X and result Y are both **REAL** or both LONGREAL.

Comments	· Be certain that argument X is less than 87.4 for single- precision numbers and less than 710.47 for double-precision numbers; otherwise, the ArgumentError exception is raised. The inverse of the exponential function is the natural logarithmic function *Ln*. If you need to calculate X to the power of Y, you can use the *Exp* and *Ln* functions in the following expression: `XPowerY:= Exp(Ln(X) * Y)`
See Also	LongMath MathLib
Example	```MODULE Exp;FROM MathLib IMPORT Exp;VAR Argument,Result:REALBEGIN IF Argument < 87.4 THEN Result:= Exp(Argument) END ;END Exp.```

EXPORT   R

Description	**EXPORT** specifies which identifiers inside a module are to be visible in the scope surrounding the module.
Usage	`EXPORT var1,var2, type1,type2, proc1,proc2`

Comments   IMPORT and EXPORT lists must appear immediately after
                       the MODULE statement and before any constant, variable,
                       type, or procedure declarations.

                       Names identifying constants, variables, types, exceptions, and
                       procedures may be exported. Modules may not be exported.

                       Everything in a definition module is exported; thus no export
                       list is required.

                       Nothing in an implementation module may be exported, unless
                       it has been previously exported by its definition module.

                       Nothing may be exported by a main module since there is
                       nothing to export to.

                       Consequently, export lists really only make sense in local
                       modules or in modules that are embedded inside of a main
                       module, procedure, or an implementation module.

Example   *local*'s *beans* is visible to *main*, but its *rice* is not:

```
MODULE main;

  MODULE local;
  EXPORT beans;
  VAR
   rice,beans: BOOLEAN;
  BEGIN
  (* code to set rice and beans *)
  END local;

  (* Any code here cannot see rice, only beans *)
  END main.
```

FALSE standard value   S

Description	FALSE denotes the **Boolean state of falsity.**
Usage	`Finished := FALSE;` *Finished* is a variable **of type BOOLEAN.**
Comments	The ordinal value of FALSE is 0; thus, *WRITE(CAR-DINAL(FALSE))* will print a 0. In contrast, the ordinal value of TRUE is 1; thus, truth is greater than falsity.
See Also	BOOLEAN TRUE
Example	```MODULE Falsity;
VAR
b: BOOLEAN;
BEGIN
REPEAT b := FALSE UNTIL b; (* repeats forever *)
END Falsity.```

FILE type (Files)   L

Description	*FILE* is an opaque type representing a low-level file. Variables of type *FILE* may represent illegible streams and random access files.
Declaration	```TYPE
FILE;```
Usage	```VAR
f: FILE;``` *FILE* must **be imported from the module** *Files*.
Comments	Use *FILE* when you are working with files that are not strictly ASCII characters or if you do not want to interpret the contents of the file.

See Also	Files
	TEXT
	Texts

Example
```
MODULE FileExists;
FROM Files IMPORT FILE, Open, Close;
VAR f: FILE;
BEGIN
 IF Open(f,'testfile.dat') THEN
  Close(f); WRITELN('File exists.');
 ELSE
  WRITELN('File does not exist.');
 END ;
END FileExists.
```

Files module   L

Description *Files* facilitates the handling of disk files.

Declaration
```
DEFINITION MODULE Files;
FROM SYSTEM IMPORT BYTE WORD, ADDRESS;

TYPE FILE;

PROCEDURE Open      (VAR f: FILE;
                     name: ARRAY OF CHAR): BOOLEAN
PROCEDURE Create    (VAR f: FILE; name:
                     ARRAY OF CHAR);
PROCEDURE Close     (VAR f: FILE);
PROCEDURE Delete    (VAR f: FILE);
PROCEDURE Rename    (VAR f: FILE; name:
                     ARRAY OF CHAR);

PROCEDURE GetName   (f: FILE;
                      VAR name: ARRAY OF CHAR);
```

```
PROCEDURE FileSize   (f: FILE): LONGINT;
PROCEDURE EOF        (f: FILE): BOOLEAN;
PROCEDURE ReadByte   (f: FILE; VAR b:BYTE);
PROCEDURE ReadWord   (f: FILE; VAR w: WORD);
PROCEDURE ReadRec    (f: FILE;
                      VAR rec: ARRAY OF WORD);
PROCEDURE ReadBytes  (f: FILE; buf: ADDRESS;
                      nbytes: CARDINAL): CARDINAL;


PROCEDURE WriteByte  (f: FILE; b:BYTE);
PROCEDURE WriteWord  (f: FILE; w: WORD);
PROCEDURE WriteRec   (f: FILE;
                      VAR rec: ARRAY OF WORD);
PROCEDURE WriteBytes (f: FILE; buf: ADDRESS;
                      nbytes:CARDINAL);


PROCEDURE Flush      (f: FILE);
PROCEDURE NextPos    (f: FILE): LONGINT;
PROCEDURE SetPos     (f: FILE; pos:LONGINT);


PROCEDURE NoTrailer  (f: FILE);

PROCEDURE ResetSys( );

EXCEPTION EndError, StatusError, UseError,
          DeviceError, DiskFull;

END Files.
```

Comments   This library module performs low-level disk access.

See Also   Texts
                InOut
                Also see individual identifiers declared in the module *Files*.

FileSize procedure (Files)   L

Description	*FileSize* gets the exact size of the file in bytes.
Declaration	**PROCEDURE** FileSize(f: FILE): LONGINT;
Usage	nbytes := FileSize(f)

FileSize must be imported from the library module *Files*.

nbytes must be of type LONGINT.

f must be of type *FILE* imported from *Files*.

Comments   Turbo Modula-2 uses the last byte of the last record (all CP/M files have 128 bytes per record) to store the number of bytes actually used in this record. This number is added to the number of bytes in the previous records (*(nrecs-1)*128*) to determine the actual file size in bytes.
This processing of the last byte can be turned off by a call to *NoTrailer*, but then *FileSize* will only return sizes in 128-byte multiples.

See Also   FILE
Files
NoTrailer

Example   To see just how big *widgetFile* is, do the following:

```
MODULE FileSizes;
FROM Files IMPORT FILE, Open;
VAR
 nbytes: LONGINT;
 widgetFile: FILE;
BEGIN
 IF Open(widgetFile, "widgets.dat") THEN
  nbytes := FileSize(widgetFile)
 END
END FileSizes.
```

FILL procedure (SYSTEM)   L

Description   *FILL* fills a block of memory with a given byte.

Declaration   **PROCEDURE** FILL(adr: ADDRESS; len: CARDINAL;
                   val: BYTE);

Usage   FILL(start, nbytes, ch)

                   FILL must be imported from the library module *SYSTEM*.

                   start must be of type *ADDRESS* imported from *SYSTEM*.

                   nbytes must be of type CARDINAL.

                   ch must be of type *BYTE* imported from *SYSTEM*.

Comments   The block of memory between *start* and the address **start +**
                   nbytes -1 is filled with the byte value *ch*.

See Also   MOVE
                   SYSTEM

Example   Fill the array *Sieve* with the value TRUE:

                   VAR Sieve: **ARRAY** [0..8191] **OF** BOOLEAN;
                   BEGIN
                   . . .
                   FILL(ADR(Sieve), 8192,TRUE);

firstDrive variable (Loader)   L

Description   *firstDrive* contains the first disk drive to be searched when
                   looking for overlay files.

Declaration   **VAR**
                   firstDrive: [0..15];

Usage   firstDrive := currentDrive;

firstDrive must be imported from the module *Loader.*

currentDrive is a CARDINAL in the range 0 to 15.

Comments   Once the location of the overlay files has been found you may speed up load time by ensuring that *firstDrive* is set to the correct drive number.

ee Also   Loader

Example

```
MODULE LoadOverlay;
FROM Loader IMPORT firstDrive, Call;
BEGIN
 firstDrive := 13; (* Search ram disk (M:) first *)
 Call('overlay.fil')
END LoadOverlay.
```

FLOAT standard function   S

Description   *FLOAT* converts argument X to REAL.

Usage   Y:= FLOAT(X)

Argument X can be of type INTEGER, CARDINAL, LONGINT, or REAL.

The argument must be in the REAL range -1E38 to 1E38.

Result Y is of type REAL.

Comments   Be certain that the argument is in the REAL range -6.80564E38 to +6.80564E38; otherwise, an overflow error will occur, causing the exception OVERFLOW to be raised.

Example   **VAR** Count:CARDINAL
                      Time:REAL
                     BEGIN
                      Time:= Time + 0.1 * FLOAT(Count);
                     FND

Flush procedure (Files)   L ˙

Description   *Flush* flushes the file's internal buffer to disk.

Declaration   **PROCEDURE** Flush(f: FILE);

Usage   Flush(f)

                     Flush must be imported from the library module *Files*.

                     f must be of type *FILE* imported from *Files*.

Comments   In order to speed up the input and output to disk files, most
                     computers maintain a file buffer for each open file. In a write
                     operation this buffer is not normally »flushed« until the buffer
                     is full. The *Flush* command forces this process.

                     This function can be used to detect write errors (like DiskFull)
                     sooner than they would otherwise appear. (They do not usually
                     appear until the internal file buffer is full and the Turbo
                     Modula-2 sytem orders a flush.)

See Also   Files

Example   **Force a DiskFull error at the record where it occurs:**

                     FOR i := 1 **TO** nrecs **DO**
                      WriteRec(f, widget[i]);
                      Flush(f);
                     END ;

FOR statement   R

Description
: FOR repeats the execution of a sequence of instructions a specified number of times.

Usage
: **FOR** ControlVar := StartExp **TO** EndExpr **[BY Step] DO**
    <Statement> {; <Statement> }
  END

  ControlVar is of type CARDINAL, INTEGER, CHAR, enumeration, or subrange. It must not be imported, nor be a procedure parameter or a structured variable's component.

  StartExp and *EndExp* are single values or expressions of the same type as the control variable.

  Step is a constant expression of **type** CARDINAL or INTEGER.

Comments
: The **FOR** statement repeatedly executes the statement sequence for a progression of control variable values. It begins with the value of the start expression and increments by the value *Step* until the control variable is equal to or greater than the value given by the end expression. If you do not specify a step value, it is assumed to be 1.

  As defined in the syntax section, the start and end values can be the results of expressions.

  FOR Y:= X **TO** X + N **BY** Ramp **DO**

  A FOR statement cannot be exited before its normal termination. If you want to do this, you can use the REPEAT or WHILE statements instead.

Example   **FOR** X:= 0 **TO** 10 **BY** 2 **DO**
                 (* Some statement sequence *)
                 END

                 This example repeats the statement sequence six times, with *X*
                 equal to the values 0, 2, 4, 6, 8, and 10. You can use *X* in the
                 statement sequence, but you cannot alter its value. For exam-
                 ple, you can do the following:

                 FOR Count:= 5 **TO** 3 **BY** -1 **DO**
                 T:= 2 * Count + 4
                 END

                 This example is repeated three times for descending values of
                 Count equal to -5, -4, and -3.

FORWARD statement   R

Description   **FORWARD** allows a procedure to be referenced before it is
                 defined.

Usage   PROCEDURE foo( ): CARDINAL; **FORWARD** ;

                 foo is a procedure or a function heading followed by the
                 reserved word **FORWARD**.

                 The body of *foo* is given after a normal procedure heading for
                 foo.

Comments   The **FORWARD** statement allows Turbo Modula-2 to imple-
                 ment mutual recursion in a one-pass compiler environment.
                 This statement was not defined in the original definition, but
                 is needed for a one-pass implementation of the compiler.

Example

```
MODULE Recurring;
PROCEDURE one; FORWARD ;

PROCEDURE two;
BEGIN
 one;  (* Call of procedure before it is defined *)
END two;

PROCEDURE one;
BEGIN
 two;
END one;

BEGIN
 one;
END Recurring.
```

FREEMEM procedure (STORAGE)   L

Description   *FREEMEM* finds out how much contiguous dynamic memory (heap space) is available.

Declaration   **PROCEDURE** FREEMEM( ): CARDINAL;

Usage   freeman := FREEMEM( )

              FREEMEM must be imported from the pseudomodule *STORAGE*.

              freeman must be of type **CARDINAL.**

Comments   This function returns the number of bytes between the heap pointer and the current top of the runtime stack. The sizes of holes in the allocated heap is not taken into account.

See Also   ALLOCATE
              DEALLOCATE
              STORAGE
              OUTOFMEMORY

Example   WRITELN('There are ',FREEMEM( ),' bytes free on the
                   heap');
                   ALLOCATE(p, 2000);
                   WRITELN('and now there **are 2000 less:**',**FREEMEM( ))**;

GetName procedure (Files)   L
_____

Description   *GetName* retrieves the name of an open file.

Declaration   **PROCEDURE** GetName(f: FILE; **VAR** name:
                   ARRAY OF CHAR);

Usage   GetName(f, fname)

                   GetName must be imported **from the library module *Files*.**

                   f must be of **type** *FILE* imported from *Files*.

                   fname must be a character array variable.

Comments   This procedure allows library modules to determine the name
                   of a file when only the file variable has been passed.

See Also   Create
                   FILE
                   Files
                   Open

Example   Get the **name** of file *f*:

                   GetName(**f,** fname)

GotoXY procedure (Terminal)   **L**

Description	*GotoXY* positions **the cursor on the screen.**
Declaration	**PROCEDURE** GotoXY(x,y: CARDINAL);
Usage	GotoXY(col, row)

GotoXY must be imported from the library module *Terminal*.

col and *row* must be of type CARDINAL.

Comments   This function will only work if Turbo Modula-2 has been properly installed on your system.

0, 0 is the upper left-hand corner
79,23 is the lower right-hand corner

(0,0)   col   (79,0)

row

(0,23)   (79,23)

See Also   ClearScreen
ClearToEOL
DeleteLine
InsertLine
Highlight
Normal
Terminal

Example   Put an 'X' in the upper right-hand corner:

                   GotoXY(79,0);WriteChar('X');

HALT standard procedure   S

Description   *HALT* stops program execution.

Usage   HALT

Comments   The *HALT* procedure terminates execution of the program and
                   returns you to Turbo Modula-2 or the operating system, depen-
                   ding on where the program execution originated from.

                   HALT is a standard Modula-2 procedure; a nonstandard way
                   to stop a program and deliver an error message to the terminal
                   is available with the *RAISE* command.

See Also   RAISE

Example   **VAR** ErrorCondition:BOOLEAN
                   BEGIN
                    IF ErrorCondition **THEN**
                     HALT
                    END ;

haltOnControlC variable (Texts)   L

Description   haltOnControlC holds the current mode of operation o
                   Control-C during console input and output.

Declaration   **VAR**
                   haltOnControlC: BOOLEAN;

Usage

        `haltOnControlC := FALSE;`

        haltOnControlC is a predefined **variable that must be imported** from the module *Terminal*.

Comments

        This variable is useful for preventing a user from stopping a program.

        The default value of *haltOnControlC* is TRUE, which means that the user can press $\boxed{\text{Ctrl}}$ $\boxed{\text{C}}$ during an *input* statement or a *WRITELN* statement to halt the program.

        Note that the only output procedure that checks for the Control-C character is the *WriteLn* procedure. Thus, programs that do not use a *WriteLn* (or *WRITELN*) statement cannot be interrupted.

See Also

        Texts

Example

```
MODULE HaltTheProgram;
FROM Texts IMPORT haltOnControlC;
VAR  I: CARDINAL;
BEGIN
 FOR I := 1 TO 20 DO
  WRITELN('Press control C to stop the program ')
 END ;
 haltOnControlC := FALSE;
 FOR I := 1 TO 20 DO
  WRITELN('You cannot stop me now . . . ')
 END ;
END HaltTheProgram.
```

HIGH standard function   S

Description	To *HIGH* returns the high-index bound of array *A*.

Usage   Y:= HIGH(A)

Argument *A* is **an array of any index and component** type.

Result *Y* is of the same type as the index type.

Comments   This function allows you to find the upper index **of any array** for example, an array defined as

Move: **ARRAY** (Left,Right,Stop) **OF** INTEGER;

has an upper index *Stop*.

Note that for multidimensional arrays, the upper index of th first dimension is returned. For example, if you have define an array as

MultiDim: **ARRAY** [10..20],(Left,Right,Stop) **OF**
                                                            CHAR

the *HIGH* function will return the result 20, which is the hig index of the first dimension.

You can also find the other dimensions as well; for exampl if you have defined

MultiDim: **ARRAY** [0..10] **OF ARRAY** [0..20] **OF WORD:**

then *HIGH(MultDim)* = *10* and *HIGH(MultiDim[0])* = *2(*

HIGH is very useful for finding the upper bound of an op(array parameter. The lower bound is always zero and the upp bound is *HIGH(a)* where *a* is an open array parameter.

See Also   ARRAY

Example

```
MODULE High;
TYPE
 Weather = (Clear,Rain,Wind,Snow);
 Day   = (Mon,Tue,Wed,Thu,Fri,Sat,Sun);
 Hour   = [1..24]
VAR
 Forecast: ARRAY Day,Hour OF Weather;
 HighForecast: Day;
 HighHour: Hour
BEGIN
 HighForecast:= HIGH(Forecast);
(* HighForecast is value Sun *)
 HighHour:= HIGH(Forecast[Sun]);
(* HighHour is 23 *)
END High.
```

Highlight procedure (Terminal)   L

Description

Highlight turns on highlighting (brightens text) for terminal output.

Declaration

PROCEDURE Highlight;

Usage

Highlight

Highlight **must be imported from the library module** *Terminal*.

Comments

Highlight is only possible on systems with screens capable of displaying text in at least two of the following modes: brightness, inverse video, or underlining.

Highlight will work on your system only after the Turbo Modula-2 system has been properly installed.

Stand-alone programs can see if the terminal they are running on has this capability by checking *OpSet* for the element *highlightNormal*.

See Also	Normal
	Terminal
	OpSet
	highlightNormal

Example	Highlight the words *»This text is high«*:

```
Highlight;
WriteLn( 'This text is high ');
Normal;
WriteLn( 'And this text is normal ')
```

---

highlightNormal enumerated value (Terminal)   L

Description	*highlightNormal* is the third value of the enumerated type *Terminal.SpecialOps*.

Declaration	`SpecialOps = (clearEol, insertDelete, highlightNormal);`

Usage	`IF highlightNormal IN available THEN Highlight END;`

SpecialOps must be imported from the library module *Terminal*.

Comments	If this value is a member of the set *Terminal.available*, then the program may use the procedure *Highlight* to enhance text written to the screen.

To use this identifier, include *SpecialOps* in your import list. The identifiers of each of *SpecialOps'* values become visible automatically.

See Also	Highlight
	available
	insertDelete
	clearEol
	Terminal

Example   Write some enhanced text:

```
MODULE CheckAvailableOperations;
FROM Terminal IMPORT
available,SpecialOps,Highlight,Normal;
FROM Strings IMPORT CAPS;
VAR s: ARRAY [0..20] OF CHAR;
BEGIN
 IF highlightNormal IN available THEN
  Highlight;
  WRITELN(s);
  Normal;
 ELSE
  CAPS(s)
  WRITELN(s);
 END ;
END CheckAvailableOperations.
```

HLRESULT variable (SYSTEM) L

Description *HLRESULT* holds the contents of the *HL* register after a BDOS or BIOS call.

Declaration **VAR**
     HLRESULT: CARDINAL;

Usage   c := HLRESULT;

     HLRESULT must be imported from the pseudomodule *SYSTEM*.

     c must be of type CARDINAL;

Comments This variable is used to access the register after operating system calls.

See **Also**	BDOS
	BIOS
	HLRESULT
	SYSTEM

Example

```
MODULE OperatingSystemCalls;
FROM SYSTEM IMPORT BDOS,BIOS,IORESULT,HLRESULT;
VAR  ReadOnlyVector: CARDINAL;
BEGIN
 BDOS(1,0); WRITE('The character read is
 ',CHR(IORESULT));
 BDOS(29,0);
 ReadOnlyVector := HLRESULT;
END OperatingSystemCalls.
```

IF statement   R

Description   IF executes a sequence of statements that depend on the result
                  of a BOOLEAN expression.

Usage   IF BooleanExpression **THEN** StatementSequence
                  { **ELSIF** BooleanExpression **THEN** StatementSequence }
                  [**ELSE** StatementSequence]
                  END

                  StatementSequence = Statement {; Statement}

Comments   Each Boolean expression is evaluated until one results in
                  TRUE. The associated statement sequence is then executed
                  and the statement ends. If none of the IF or ELSIF expressions
                  yield a TRUE result, the optional ELSE statement sequence
                  is executed.

                  Note that Boolean expressions are only evaluated until one is
                  TRUE.

Example

```
VAR
 XAxis,YAxis: REAL;
BEGIN
 IF XAxis < 10.0 THEN
  YAxis:= 2.0 * XAxis
 ELSIF XAxis < 20.0 THEN
  YAxis:= XAxis + 10.0
 ELSE
  YAxis:= 30.0
 END ;
```

IMPLEMENTATION declaration   R

Description   IMPLEMENTATION marks the begining of the implementation code of a module.

Usage

```
IMPLEMENTATION MODULE moduleName;
<declarations>
BEGIN
<body>
END moduleName.
```

Comments   An implementation module always has a definition module, and together they form a library module from which objects can be imported. The definition module contains the declarations of all the exported objects. The implementation module contains the actual code that defines the objects and procedures that are exported.

An implementation module is a separate compilation unit. In general, it is possible to make a change to an implementation module and recompile it without changing the client modules. Other .MCD files that import objects from the changed module will incorporate those changes without having to be recompiled. This is possible because Turbo Modula-2 programs are linked at runtime when run from the menu shell.

An implementation module is not normally a stand-alone program (though it can be); it is usually executed by importing its code into a so-called »main module«.

The main body of code in an implementation module is executed at load time. This is referred to as the initialization section. It is used to set up variables before any client uses that library's services.

See Also   DEFINITION
                  Linker
                  MODULE

Example   Here's a definition module in the file COMPUTE.DEF:

```
DEFINITION MODULE Compute;
VAR
 PI: REAL;

 PROCEDURE Add( Y,Z :INTEGER):INTEGER
 PROCEDURE Sub( Y,Z :INTEGER):INTEGER
END Compute.
```

And here's the corresponding implementation module in file COMPUTE.MOD:

```
IMPLEMENTATION MODULE Compute;

PROCEDURE Add( Y,Z :INTEGER):INTEGER;
 RETURN Y + Z
END Add;

PROCEDURE  Sub( Y,Z :INTEGER): INTEGER;
 RETURN Y - Z
END Sub;

BEGIN
 PI := 3.14159;
END Compute.
```

IMPORT declaration   R

Description   IMPORT specifies which identifiers a module is able to »see
                    from another module or the surrounding environment.

Usage   **IMPORT** identifiers;

                    or

                    FROM moduleName **IMPORT** identifiers;

                    The first form of IMPORT imports identifiers as a whole. I
                    the identifier is a module, then that module's identifiers mus
                    be qualified.

                    The second form de-qualifies identifiers as they are imported

Comments   IMPORT lists must come immediately after the MODUL:
                    declaration and before any constant, variable, type, or pro
                    cedure declarations.

                    In local modules, the IMPORT declarations must come befor
                    the EXPORT declarations.

                    Qualified identifiers can be accessed by prefixing the identifi
                    with   the   module   name,   followed   by   a   ».«,   as   :
                    ModuleName.IdentifierName.

                    Note that you need not name the module the same as the fi
                    name. However, the compiler, the linker, and the loader u:
                    the first eight characters of the name given in the MODUL
                    heading when searching for symbol files, library code file
                    and linked overlay files. Thus, we recommend that you alwa
                    use the first eight letters of the module name for the actual fi
                    name.

See Also   EXPORT
                    QUALIFIED

Example   Input the *Append* procedure from *Strings* and use it in the
                 following:

```
MODULE SayHello;
FROM Strings IMPORT Append;
VAR
 s : ARRAY [0..20] OF CHAR;
BEGIN
 s := 'Hello';
 Append(' Folks', s);
 WRITELN(s);
END SayHello.
```

                 Import all of *Strings* and do the same thing you did in the
                 previous example:

```
MODULE SayHello;
IMPORT Strings;
VAR
 s : ARRAY [0..20] OF CHAR;
BEGIN
 s := 'Hello';
 Strings.Append( ' Folks', s); (* Qualified *)
 WRITELN(s);
END SayHello.
```

INC standard procedure   S

Description   *INC* returns a successor of argument *X*.

Usage   INC(X [,N])

                 Argument *X* may be of type INTEGER, CARDINAL, CHAR,
                 BOOLEAN or enumeration.

                 Optional argument *N* must be a CARDINAL; it is the incre-
                 ment that will be added to *X*.

Comments   If you add 1 to a CARDINAL value, the result is the value im-
mediately after the original value. With *INC*, you can do the
exact same thing with any ordinal type. For example, suppose
you have an enumeration variable

VAR Energy: (Kinetic,Potential,Heat);

You can assign any of the three named values to the variable
Energy. To increment the value of *Energy* to a higher value, use
INC(Energy).

Now, you have performed an addition operation on the
enumeration variable. You can increment by more than one
value as shown in the following examples.

The reverse operation selecting the previous **value** is perform-
ed by the standard procedure *DEC*.

When incrementing a value, you may reach its *MAX* value; for
example, the *MAX(CARDINAL)* is 65535. You may choose
whether or not this situation generates an error by using the
overflow compiler switch. This can be done globally or local-
ly. Thus, with overflow checking turned off, as in

```
VAR
 i: CARDINAL;
BEGIN
 i := MAX(CARDINAL);
 (*$O-*)
 INC(i);
 (*$O+*)
 WRITELN(i);
END
```

the *i* would wraparound to *MIN(CARDINAL)*, which is zero,
and no error will be generated. If overflow checking has been
turned on, then the exception OVERFLOW will be raised.
This can be trapped as follows:

```
MODULE inc;
FROM SYSTEM IMPORT OVERFLOW;
VAR
i: CARDINAL;
BEGIN
 (*$O+*)
 i := MAX(CARDINAL);
 INC(i);
EXCEPTION
 OVERFLOW: WRITELN('Trapped overflow error');
END inc.
```

See Also   **DEC**

Example   **TYPE Color** = (Red,Orange,Yellow,Green,Blue,
                        Indigo,Violet);
             VAR Rainbow: Color;
               Increment:CARDINAL

             BEGIN
              Rainbow:= Red;
              INC(Rainbow);
             (* Rainbow has the value Orange *)

              INC(Rainbow,2);
             (* Rainbow has the value Green *)
              Increment:= 1;
              INC(Rainbow,Increment);
             (* Rainbow has the value Blue *)

INCL standard procedure   S

Description	INCL includes element I into the set S.
Usage	`INCL(S,I)`
	Argument *S* is of set type.
	Argument *I* is of the set's **base type**.
Comments	This standard procedure allows you to include an element into a set. It has the effect of turning on a bit that represents the element in the word representing the set.
See Also	EXCL
Example	**TYPE** Day = (Mon,Tues,Wed,Thu,Fri,Sat,Sun); **VAR** Work: **SET OF** Day;

```
BEGIN
 Work:= Day{Mon..Fri};
(* Set Work contains five elements Mon to Fri *)

 INCL(Work,Sun);
(* Set Work contains six elements, Sun included *)
```

Init procedure (Processes)   L

Description	*Init* initializes a *SIGNAL* variable.
Declaration	**PROCEDURE** Init(**VAR** s: SIGNAL);
Usage	`Init(s)`
	Init must be imported from the library module *Processes*.
	s must be of type *SIGNAL* imported from *Processes*.

Comments   This is part of a module that gives a standardized set of cor-outine facilities, as well as a rudimentary form of interprocess communications upon which to base scheduling.

The three routines *Init*, *SEND*, and *WAIT* work together in the following way: *Init* initializes a queue belonging to a given signal. Every time a process calls the procedure *WAIT* with this signal, it will be entered in the queue. When a process calls the procedure *SEND* with this signal, the first process in the queue will be resumed and will be purged from the queue. A fourth routine, *Awaited*, returns TRUE if the queue is not empty.

The initialization of a *SIGNAL* by *Init* is mandatory.

See Also   Awaited
Processes
SEND
SIGNAL
StartProcess
Wait

Example   Initialize the signal *DonePrinting*:

```
INIT(DonePrinting)
```

InitScreen procedure (Terminal)   L

Description   *InitScreen* sends an initialization string to the terminal.

Declaration   **PROCEDURE** InitScreen;

Usage   InitScreen

InitScreen must be imported from the library module *Terminal*.

Comments   *InitScreen* will send the initialization string specified by t
                 user upon installation of the Turbo Modula-2 system. Ma
                 terminals don't need an initialization string, in which ca
                 nothing happens.

See Also   EditScreen
                 Terminal

Example   Initialize the screen:

                 ```
                 InitScreen;
                 ```

---

inName variable (ComLine)   L

Description   *inName* is a string variable that contains a string found af
                 the input redirection symbol (<) on the command line.

Declaration   **VAR**
                 inName: **ARRAY [0..19] OF** CHAR;

Usage   ```
                 WRITE('Input redirected from ',inName);
                 ```

Comments   The redirection symbol and *inName* will not **appear** in the t
                 commandLine.

                 inName contains the standard output string CON: if
                 redirection argument has been found on the command lin

See Also   ComLine
                 inName
                 RedirectOutput

Example

```
MODULE RedirectInput;
FROM ComLine IMPORT inName, RedirectInput;
FROM Texts IMPORT console;
BEGIN
 RedirectInput;
 IF inName < > "CON:" THEN
  WRITE(console,'Input redirected from ',inName);
 END
END RedirectInput.
```

InOut module   L

Description   *InOut* performs input and output.

Usage   **DEFINITION MODULE** InOut;

```
CONST EOL=36C;
VAR Done:BOOLEAN
termCH: CHAR;

PROCEDURE OpenInput(defext: ARRAY OF CHAR);
PROCEDURE OpenOutput(defext: ARRAY OF CHAR);
PROCEDURE CloseInput;
PROCEDURE CloseOutput;

PROCEDURE Read(VAR ch: CHAR);
PROCEDURE ReadString(VAR s: ARRAY OF CHAR);
PROCEDURE ReadInt(VAR x: INTEGER);
PROCEDURE ReadCard(VAR x: CARDINAL);

PROCEDURE Write(ch: CHAR);
PROCEDURE WriteLn;
PROCEDURE WriteString(s: ARRAY OF CHAR);
PROCEDURE WriteInt(x: INTEGER; n: CARDINAL);
PROCEDURE WriteCard(x,n: CARDINAL);
PROCEDURE WriteHex(x,n: CARDINAL);
PROCEDURE WriteOct(x,n: CARDINAL);
```

```
PROCEDURE ReadReal(VAR x: REAL);
PROCEDURE WriteReal(x: REAL; n,digits: CARDINAL);

END InOut.
```

Comments   This is the original input and output module defined by Wirtl
                 It is postulated to be present in every implementation of
                 Modula-2 and is included here to maintain compatibility wi
                 other implementations.

                 The module *InOut* is usually implemented using the **modu**
                 Texts with the *input* or *output* text streams.

                 WriteHex and *WriteOct* can sometimes prove useful for writii
                 addresses.

IORESULT variable (SYSTEM)   L

Description   IORESULT holds the contents of the *A* register after a BD(
                 or BIOS call.

Declaration **VAR**
                   IORESULT: CARDINAL;

Usage   c := IORESULT;

                 IORESULT must **be** **imported** **from** **the** pseudomodi
                 SYSTEM.

                 c must be of **type** CARDINAL;

Comments   This variable is used to access **the register after** operati
                 system calls.

See Also   BDOS
                 BIOS
                 HLRESULT
                 SYSTEM

Example
```
MODULE OperatingSystemCalls;
FROM SYSTEM IMPORT BDOS,BIOS,IORESULT,HLRESULT;
VAR  ReadOnlyVector: CARDINAL;
BEGIN
 BDOS(1,0); WRITE('The character read is
 ',CHR(IORESULT));
 BDOS(29,0);
 ReadOnlyVector := HLRESULT;
END OperatingSystemCalls.
```

INP procedure (SYSTEM)   L

Description   *INP* reads a byte from a given I/O port.

Declaration   **PROCEDURE** INP(port: WORD): CARDINAL;

Usage   inByte := INP(port)

               INP must be imported from the library module *SYSTEM*.

               inByte must be of type INTEGER or CARDINAL.

               port can be any scalar type compatible with *WORD*.

Comments   The port, whose number is given by *port*, is read and the value is placed in *inByte*.

See Also   OUT
               SYSTEM

Example   Get a byte by doing the following:

               KbdStatus := INP(KbdStatusPort);

input (Texts)   L

Description   *Input* is the standard *TEXT* **variable that can** be redefined.

Declaration   **VAR** input: TEXT;

Usage   input

                     input **must** be **imported** from the library module *Texts*.

                     input is used anywhere an input text is needed.

Comments   *input* is declared in *Texts* as a variable of type *TEXT* and
                     read-only.

                     input is always open and is connected to the terminal l
                     default. It operates as if the call

                     OpenText(input, CON:)

                     has taken place before the execution of every main modul
                     The call

                     CloseText(input)

                     reestablishes this connection after input has been redirecte

                     READ and READLN always use input as their default t∈
                     stream (users may use input without knowing it). On occasio
                     when the user wants to explicitly manipulate the input t∈
                     string, he will have to import input from Texts.

See Also   input
                     console
                     Texts

Example   Read aNumber from input:

                     ReadInt(input, aNumber) |

Insert procedure (Strings)   L

Description	*Insert* inserts a substring into a string.

Declaration	**PROCEDURE** Insert(substr: **ARRAY OF** CHAR; 　　　　　　　**VAR** str: **ARRAY OF** CHAR; inx: CARDINAL);

Usage	Insert(substr, str, at)

substr and *str* must be of type **ARRAY OF CHAR**. *substr* may be a literal, but *str* must be a variable.

at is the index of the destination byte and it must be of type CARDINAL.

Comments	*Insert* actually inserts; the old data in *str* is not overwritten but is moved over in the string.

See Also	Append Copy Delete Length Pos Strings

| Example | ```
str := 'This is Modula-2';
  substr := 'Turbo ';
  Insert(substr, str, 8)
  (* str now is 'This is Turbo Modula-2' *)
``` |

insertDelete enumerated value (Terminal)   L

Description	*insertDelete* is the second value of the enumerated type *Terminal.SpecialOps*.

Declaration	SpecialOps = (clearEol, insertDelete, highlightNormal);

Usage   **IF** `insertDelete` **IN** `available` **THEN** `InsertLine` **END** ;

                    SpecialOps must be imported from the library module *Terminal*.

Comments   If this value is a member of the set *Terminal.available*, then
                    the program may use the procedures *InsertLine* and *DeleteLine*
                    to manipulate text on the screen.

                    To use this identifier, include *SpecialOps* in your import list.
                    The identifiers of each of *SpecialOps'* values become visible
                    automatically.

See Also   InsertLine
                    DeleteLine
                    available
                    highlightNormal
                    clearEol
                    Terminal

Example   Insert and delete a line on the screen:

```
MODULE CheckAvailableOperations;
FROM Terminal IMPORT
 available,SpecialOps,
 InsertLine,DeleteLine,GotoXY;
VAR
 s: ARRAY [0..20] OF CHAR;
BEGIN
 IF insertDelete IN available THEN
 GotoXY(0,10);
 InsertLine;
 WRITELN(s);
 GotoXY(0,10);
 DeleteLine;
 ELSE
 GotoXY(0,10);
 (* Clear the line and move everything down *)
 WRITELN(s);
 GotoXY(0,10);
 (* Move everything back up one line *)
 END ;
END CheckAvailableOperations.
```

InsertLine procedure (Terminal)   L

Description   *InsertLine* inserts a blank line onto the screen at the current cursor position.

Declaration   **PROCEDURE InsertLine;**

Usage   InsertLine

                 InsertLine must be imported from the library module *Terminal*.

Comments	This operation is not available on all systems. To determine if the operation is on the current system, inspect the variable *Available* imported from *Terminal*. If *insertDelete* is an element of the set *Available*, then insertion and deletion of screen lines can be performed.
See Also	DeleteLine GotoXY Terminal
Example	Insert the **text** *»Here I am«* **at the sixth line from the top of the** screen.

```
GotoXY(0,5)
InsertLine;
GotoXY(0,5);
WRITE('Here I am');
```

INT standard function   S

Description	*INT* converts the argument to type INTEGER.
Usage	`Y:= INT(X)`
	Argument *X* can be of type INTEGER, CARDINAL, LONGINT, REAL, and LONGREAL.
	The argument must be in the INTEGER range -32768 to 32767.
	Result *Y* is type INTEGER.
Comments	When you use this function with a REAL argument, the value is truncated (the fractional part removed); thus, a value 12.35 becomes 12.

Example   **MODULE** ConvertToInteger;
               VAR
                r: REAL;
                i: INTEGER;
               BEGIN
                x := 39.489;
                i := INT(x) * 10;
               END ConvertToInteger.

\TEGER standard type   S

Description   INTEGER is a standard type with variables that can assume
               whole values between -32768 and 32767. In addition, IN-
               TEGER can be used as a type-transfer function.

Usage   Variable declarations:

               VAR
                i,j: INTEGER;

               Type transfer:

               i := INTEGER(65535)

               where *i* is an integer. This statement assigns -1 to the integer *i*.

Comments   You can use INTEGER variables whenever you know that the
               possible values are limited to whole numbers. Note that when
               the possibilities include only positive whole numbers, you are
               better off using CARDINAL types. In practice, you rarely
               need to use INTEGER variables.

               You may use any arithmetic operator in INTEGER expres-
               sions, and INTEGER variables may take part in relational ex-
               pressions.

               The *INT* standard function will convert other numeric data
               types into INTEGER.

An INTEGER value requires 2 bytes for storage. The bytes a: stored as two's complement, with the least-significant by stored first.

INTEGER is compatible with *WORD*. Since *BYTE* is subrange of *WORD*, INTEGER is compatible with *BYTE* well. However, the exception BoundsError can occur if negative integer is assigned to a byte (or a positive integ greater than 255).

See Also   CARDINAL
                  LONGINT
                  WORD

Example   **MODULE** Integer;
                  VAR
                   Deviation: INTEGER;
                   Angle: REAL;
                  BEGIN
                   FOR Deviation:= -3 **TO** 3 **DO**
                    Angle:= 10.0 * Float(Deviation);
                   END
                  END Integer.

IntToStr procedure (Convert)   L

Description   *IntToStr* converts an integer variable to a string.

Declaration   **PROCEDURE** IntToStr(i: INTEGER; **VAR s: ARRAY OF**
                  CHAR);

Usage   IntToStr(i, s)

                  IntToStr must be imported from the library **module** *Conve*

                  i must be of type INTEGER.

                  s must be of type **ARRAY OF CHAR.**

Comments	The string will be right-justified, and the left-most characters padded with blanks.
	If the number is too large to fit into the string, the exception TooLarge will be raised.
See Also	CardToStr
	Convert
	Doubles
	LongToStr
	RealToStr
	StrToInt
Example	Put the string '5000' into *s*:

```
i := 5000;
IntToStr (i,s);
```

IOTRANSFER procedure (SYSTEM)   L

Description	*IOTRANSFER* sets up an interrupt vector to point to the next line of code and transfers control to another coroutine.
Declaration	**PROCEDURE** IOTRANSFER(**VAR** source, dest: PROCESS; n: CARDINAL);
Usage	IOTRANSFER(inthandler, background, intvector)
	IOTRANSFER must be imported from the library module *SYSTEM*.
	inthandler and *background* must be of type *PROCESS* imported from *SYSTEM*.
	intvector must be of type CARDINAL.

Comments	*inthandler* and *background* must have been previously set u with calls to *NEWPROCESS*.
	intvector must be a legal and available interrupt vector numbe
See Also	SYSTEM TRANSFER
Example	IOTRANSFER(KbdIntHandler,MainRoutine,KbdIntVec- tor);

---

legal variable (Doubles)   L

Description	*legal* monitors the input of a LONGREAL during a call *ReadDouble*.
Declaration	**VAR** legal: BOOLEAN;
Usage	finished := legal; *finished* must be of type BOOLEAN.
Comments	The variable is usually checked after input of a LONGREA *ReadDouble* is the only procedure that can affect it.
See Also	Texts done

Example	**MODULE** InvalidInput;
	FROM Doubles **IMPORT** legal, ReadDouble;
	FROM Texts **IMPORT** input;
	VAR d: LONGREAL;
	BEGIN
	 REPEAT
	  ReadDouble(input,d);
	 UNTIL legal;
	END InvalidInput.

Length procedure (Strings)   L

Description	*Length* finds the current length of a string.
Declaration	**PROCEDURE** Length(**VAR** str: **ARRAY OF** CHAR): CARDINAL;
Usage	size := Length(str)

Length must be imported from the library module *Strings*.

size must be of type CARDINAL.

str must be a character array variable.

Comments	*Length* returns the position of the first byte containing a zero, which is defined to be the end of the string.

size will not be greater than the originally declared size of *str*.

See Also	Append Copy Delete Insert Pos
Example	Find the length of *str*:

size := Length(str)

LoadError exception (Loader)   L/E

Description	LoadError is raised by the module *Loader* when the loading of an overlay is unsuccessful.
Declaration	**EXCEPTION** LoadError;

Usage	**EXCEPTION** LoadError: WRITELN('Overlay Error');
	LoadError must be imported from the module *Loader.*
Comments	This exception is raised if the overlay file is not found, if there is not enough memory to hold the overlay, or if a version conflict occurs.
See Also	Loader
Example	**MODULE** LoadOverlay; **FROM** Loader **IMPORT** firstDrive, Call, LoadError; **BEGIN** firstDrive := 13; (* Search ram disk (M:) first *) Call('overlay.fil') **EXCEPTION** LoadError: WRITELN(' Load error occured '); RAISE **END** LoadOverlay.

LongMath module   L

Description	*LongMath* provides commonly used double-precision mathematical functions.
Declaration	**DEFINITION MODULE** LongMath;

```
PROCEDURE Sqrt    (x:LONGREAL):LONGREAL;
PROCEDURE Exp     (x:LONGREAL):LONGREAL;
PROCEDURE Ln      (x:LONGREAL):LONGREAL;
PROCEDURE Cos     (x:LONGREAL):LONGREAL;
PROCEDURE Arctan  (x:LONGREAL):LONGREAL;
PROCEDURE Entier  (x:LONGREAL):LONGINT;

EXCEPTION ArgumentError;

END LongMath.
```

Comments   These routines are identical to the routines for single- precision real numbers.

The ArgumentError exception is raised when an argument is outside the permitted range. The following ranges apply:

Sqrt   Argument must be positive or zero

Exp   Argument must be less than 710.47

Ln   Argument must be positive (greater than zero)

If you want to use the same routine from *MathLib* and *LongMath*, the statements

```
FROM MathLib IMPORT Sqrt;
FROM LongMath IMPORT Sqrt;
```

will lead to a conflict. Thus, qualified import must be used.
```
IMPORT MathLib,LongMath;
BEGIN
  dSqr := LongMath.Sqrt( arg );
  rSqr := MathLib.Sqrt( arg );
```

LONGREAL standard type   S

Description   LONGREAL is a standard type with variables that can assume any value between -3.5953862697246D+308 and +3.5953862697246D+308.

Usage   **VAR** x,y: LONGREAL;

Comments   LONGREALs are particularly useful in arithmetic expressions
                  that concern a small difference between large numbers or a
                  sum that results from many (a 1000 or so) additions and sub-
                  tractions. In both cases, the round-off error tends to ac-
                  cumulate; however, it can be kept smaller by using a
                  LONGREAL. Don't use them indiscriminately, though, since
                  your program will pay the price in slower execution times.
                  You may use any arithmetic operator in LONGREAL expres-
                  sions, and LONGREAL variables may take part in relational
                  expressions.

                  The *DOUBLE* standard function will convert other numeric
                  data types into LONGREAL.

                  String and I/O support for LONGREALs are provided by the
                  library module *Doubles*.

                  Mathematical support similar to *MathLib* for REALs is pro-
                  vided by the library module *LongMath*.

                  Turbo Modula-2 uses the IEEE 8-byte, double-precision real-
                  number format; that is, in the order of their significance, 1
                  sign bit, 11 exponent bits, and 52 mantissa bits:

63 62		52 51		0
s	exponent		mantissa	

                  The mantissa is in binary format but is offset by 1.0, represen-
                  ting only those digits to the right of the decimal point.

                  The bytes are stored with the least-significant bytes in the
                  lower-numbered addresses.

                  If the exponent is zero, the floating-point value is considered
                  zero.

Sample hex real numbers:

3FF0000000000000H = 1.0
4000000000000000H = 2.0

See the example in *WORD* for printing out other values.

Example   **VAR**
      SquareArea, SquareSide: LONGREAL;
     BEGIN
      SquareSide:= 4.21D0;
      SquareArea:= SquareSide * SquareSide;

Ln function (MathLib, LongMath) L

Description   Ln returns the natural logarithm of X.

Declaration **PROCEDURE** Ln(x: REAL): REAL;
      PROCEDURE Ln(x: LONGREAL): **LONGREAL;**

Usage   Y:= Ln(X)

      Argument X **and result** Y **are either both REAL or both**
      LONGREAL.

      The argument X **must be positive.**

Comments   You must always make sure that the argument X is a positive
      value; otherwise, the ArgumentError exception is raised. The
      inverse of the logarithmic function is the natural exponential
      function *Exp*.

See Also   Exp
      LONGMATH
      MATHLIB

Example

```
MODULE LnTest;
FROM MathLib IMPORT Ln;
VAR
 Argument, Result: REAL;
BEGIN
 IF Argument > 0.0 THEN
  Result:= Ln(Argument)
 END
END LnTest;
```

Loader module   L

Description   *Loader* is a library module that handles overlays.

Declaration   **DEFINITION MODULE** Loader;

```
VAR
firstDrive: [0..15];

PROCEDURE Call(modName: ARRAY OF CHAR);

EXCEPTION LoadError;

END Loader.
```

Comments   *Loader* is used by a program to explicitly overlay certair modules. The overlay code may not export anything to the caller; however, the overlay code may communicate with the caller by importing from the caller a buffer where data anc messages can be left.

See Also   Refer to »The Loader **Module**« in **Chapter 11.**

LONG standard function   S

Description	*LONG* converts the **argument** to type LONGINT.

Usage   Y:= LONG(X)

Argument X can be of type INTEGER, CARDINAL, LONGINT, REAL, or LONGREAL.

The argument must be in the range from $-(2^{31}-1)$ to $2^{31}-1$.

The result Y is type LONGINT.

Comments   When you use this function with a REAL argument, the value is truncated (the fractional part removed); thus, the value 12.35 becomes 12.

Do not confuse this function with the LONGINT-type transfer function, which transfers data without converting it.

See Also   FLOAT
                 INT
                 TRUNC

Example   **VAR**
                  Convert: LONGINT;
                 BEGIN
                  Convert:= 2000000L * LONG(-12);
                 (* Convert is -24000000L *)
                 END

LONGINT standard type   S

Description	LONGINT is a standard type with variables that can **assume** whole values between -2147483648 to 2147483647.
Comments	You can use LONGINT types when you need a larger range of whole numbers than that offered by the CARDINAL and INTEGER types.
	You may use any arithmetic operator in LONGINT expressions, and LONGINT variables may take part in relational expressions.
	The LONG standard function will convert other numeric data to type LONGINT. Note that even INTEGERtypes must be converted.
	A LONGINT value requires 4 bytes for storage. The bytes are stored as two's complement, with the least-significant byte stored first.
	LONGINT is compatible with ARRAY OF WORD, but no with WORD or BYTE.
See Also	CARDINAL INTEGER WORD
Example	

```
MODULE Longs;
CONST
 N = 1000000L;
VAR
 BigNumber:LONGINT;
BEGIN
 BigNumber:= N;

 BigNumber:= 2L * BigNumber;
END Longs.
```

LongMath module

Description	Provides commonly used mathematical functions with double precision accuracy.

Declaration

DEFINITION MODULE LongMath;

```
PROCEDURE Sqrt    (x:LONGREAL): LONGREAL;
PROCEDURE Exp     (x:LONGREAL): LONGREAL;
PROCEDURE Ln      (x:LONGREAL): LONGREAL;
PROCEDURE Sin     (x:LONGREAL): LONGREAL;
PROCEDURE Cos     (x:LONGREAL): LONGREAL;

PROCEDURE Arctan (x:LONGREAL): LONGREAL;
PROCEDURE Entier (x:LONGREAL): LONGINT;

EXCEPTION ArgumentError;

END LongMath.
```

For a description of each of these procedures and exceptions, see Chapter 11 or the appropriate reference in this chapter.

Comments

The *LongMath* module exports a number of mathematical procedures and functions. You can import any of the procedure and exception identifiers into your own modules, which allows you to use the predefined facilities.

The *ArgumentError* exception is raised when an argument is outside the permitted range. The following ranges apply:

Sqrt   Argument must be positive or zero.

Exp   Argument must be less than 710.475D00

Ln   Argument must be positive (greater than zero).

Example   If you want to cáll the *Cos* and *Sin* functions from a modul
                 called *Problem*, you would start the module with:

                 MODULE Problem;
                 FROM LongMath **IMPORT** Cos,Sin;

                 If *Y,Z*, *Angle*, and *Radians* are defined as LONGREA
                 variables then the library functions can then be used normall$

                 Y:= Cos(Angle);
                 Z:= Sin(Radians);

LongToStr procedure (Convert)   L

Description   *LongToStr* converts a long integer variable to a string.

Declaration   **PROCEDURE** LongToStr(l: LONGINT; **VAR s: ARRAY OF**
                 CHAR);

Usage   LongToStr(l, s)

                 LongToStr must be imported from the library module *Conver*

                 l must be of type LONGINT.

                 s must be a character array variable.

Comments   The string will be right-justified, and the left-most characte
                 will be padded out with blanks.

                 If the number is too large to fit into the string, the exceptic
                 TooLarge will be raised.

See Also   Doubles
                 IntToStr
                 CardToStr
                 RealToStr
                 Strings
                 StrToLong

Example   Put the string '300000' into *s*:

```
VAR
 s : ARRAY [0..6] OF CHAR;
 l :LONGINT;
BEGIN
 l := 300000;
 LongToStr(l,s);
END
```

LOOP statement   R

Description   **LOOP** repeatedly executes a statement sequence until
                   terminated by an **EXIT** statement.

Usage   **LOOP** StatementSequence **END**

                   StatementSequence = Statement {; Statement}

Comments   There are two main reasons for using a **LOOP** statement: (1)
                   when programming a continually cycling process, such as
                   many real-time systems and concurrent processes, or (2) when
                   the **FOR, REPEAT** and **WHILE** statements are unsuitable
                   because loop termination can only be determined in the middle
                   of the loop.

                   Termination of the loop requires an EXIT statement.

See Also   EXIT

Example   **BEGIN**
                   LOOP
                    IF Found **THEN EXIT END**
                   END ;

MARK procedure (STORAGE)   L

Description	LOOP marks the beginning of a **block of dynamic variable**:
Declaration	**PROCEDURE** MARK(**VAR** a: ADDRESS);
Usage	MARK(a)
	MARK must be imported from the library module *STORAGI*
	a must be compatible with type *ADDRESS* (any pointer).
Comments	*MARK* and *RELEASE* provide an alternate method of handlir heap data. The use of these procedures allows the heap to t treated in a stack-like manner.
	A call to *MARK* creates a new heap that consists of the spac between the previous top of Heap and the stack pointer. A variables allocated on the heap prior to the call to *MARK* a no longer accessible until a *RELEASE* with the same point is called.
See Also	DISPOSE NEW RELEASE
Example	Allocate a **mess of variables and then throw them away:**

```
MARK(MessStart);
NEW(Mess1);
NEW(Mess2);
NEW(Mess3);
RELEASE(MessStart);
```

MathLib module   L

Description   *MathLib* provides commonly used mathematical functions.

Declaration   **DEFINITION MODULE** MathLib;

                      PROCEDURE Sqrt(x:REAL):REAL;
                      PROCEDURE Exp(x:REAL):REAL;
                      PROCEDURE Ln(x:REAL):REAL;
                      PROCEDURE Sin(x:REAL):REAL;
                      PROCEDURE Cos(x:REAL):REAL;
                      PROCEDURE Arctan(x:REAL):REAL;
                      PROCEDURE Entier(x:REAL):INTEGER;

                      PROCEDURE Randomize(n:CARDINAL);
                      PROCEDURE Random( ):REAL;

                      EXCEPTION ArgumentError;

                      END MathLib.

                      For a description of each of these procedures and exceptions, see Chapter 11 or the appropriate reference in this chapter.

Comments   The *MathLib* module exports a number of mathematical procedures and functions. You can import any of the procedure and exception identifiers into your own modules, allowing you to use the predefined facilities.

                      The exception ArgumentError is raised when an argument is outside the permitted range. The following ranges apply:

                      Sqrt   Argument must be positive or zero.

                      Exp   Argument must be less than 87.4.

                      Ln   Argument must be positive (greater than zero).

Example   If you want to call the *Cos* and *Sin* functions from a
                 called *Problem*, you would start the module with the fo

```
MODULE Problem;
FROM MathLib IMPORT Cos,Sin;
```

                 The predefined functions can then be used normally,

```
Y:= Cos(Angle);
Z:= Sin(Radians);
```

MAX standard function   S

Description   *MAX* returns the largest element of type *T*.

Usage   `Y:= MAX(T);`

                 Argument *T* is of type CARDINAL, INTEGER, BO
                 CHAR, enumeration, LONGINT, REAL, or LONGI
                 short, any unstructured scalar type.

                 Result *Y* is of the same type.

Comments   This function allows you to find the last elemer
                 unstructured type. Note that the argument is the typ
                 a variable of the type.

                 The maximum values of the predefined types are

CARDINAL	65535
INTEGER	32767
BOOLEAN	TRUE
CHAR	377C
REAL	2147483647
LONGINT	3.5953862697246D+308
LONGREAL	6.80565E+38

                 The smallest element of a type is returned y the *MIN*

module
lowing:

like so:

(

Example   **MODULE** Max;
                 TYPE
                  Weather = (**Clear,Rain,Wind,Snow**);
                 VAR
                  MaxWeather: Weather;
                  MaxCardinal: CARDINAL
                 BEGIN
                  MaxWeather:= MAX(Weather);
                 (* MaxWeather is value Snow *)
                  MaxCardinal:= MAX(CARDINAL);
                 (* MaxCardinal is value 65535 *)
                 END Max.

MIN standard function   S

Description   *MIN* returns the smallest element of type *T*.

ᵎLEAN,   Usage   Y:= MIN(T);
EAL; in

                 Argument *T* is of type CARDINAL, INTEGER, BOOLEAN,
                 CHAR, enumeration, LONGINT, REAL, or LONGREAL; in
                 short, any unstructured type.

ᵗ in any   Result *Y* is of the same type.
ᵉ and not

Comments   This function allows you to find the first element in any
                 unstructured type. Note that the argument is the type and not
(a variable of the type.

                 The minimum values of the predefined types are

CARDINAL	0
INTEGER	-32768
BOOLEAN	FALSE
CHAR	0C
REAL	-6.80565E+38
function.	LONGINT	-2147483648
LONGREAL	-3.5953862697246D+308

The largest element of a type is returned by **the** *MAX* function.

Example

```
MODULE Min;
TYPE
 Weather = (Clear,Rain,Wind,Snow);
VAR
 MinWeather: Weather;
 MinCardinal:CARDINAL
BEGIN
 MinWeather:= MIN(Weather);
 (* MinWeather is value Clear *)
 MinCardinal:= MIN(CARDINAL);
 (* MinCardinal is value 0 *)
END Min.
```

MOD standard operator   R

Description   **MOD** is an integer modulus operator; it is a binary operator that works on operands of the same type. The **MOD** operator returns the remainder of integer division. The operands may be both CARDINAL, both INTEGER, or both LONGINT.

Usage   i := i **MOD** j;

where *i* and *j* are of the same type.

The result of the expression is the same **type as** the operands.

Comments   The modulus operator can be used to insure that a variable remains within a certain defined range.

See Also   DIV

Example
```
MODULE Modulus;
VAR
 i: INTEGER;
BEGIN
 LOOP
 i := (i + 1) MOD 100; (* i will range from 0 to 99 *)
 END
END Modulus.
```

(ꓭDULE declaration   R

Description   *MODULE* **serves to declare the beginning of a module.**

Usage   **MODULE** moduleName

         or

DEFINITION MODULE libraryName

         or

IMPLEMENTATION MODULE libraryName

Comments   The first form declares a main module, which is a stand-alone
              program. It can also start the beginning of a local module.

              A local module limits visibility of global objects (variables,
              types, constants, procedures). Thus, an object buried inside a
              local module can retain its value throughout the program's
              lifetime, while simultaneously remain safe from outside
              tampering.

              The second and third forms are for libraries, separately com-
              piled sets of frequently used routines.

See Also   DEFINITION
              IMPLEMENTATION

Example   The following file is an example of a main module in
                 'MAIN.MOD':

```
MODULE main;
FROM SimpleMath IMPORT add;
BEGIN
 WRITELN(' 2 + 2 = ',add(2,2));
END main.
```

The next example is a local module in a main module:

```
MODULE Main;

 MODULE local;
 EXPORT p1;

 PROCEDURE p1;
 BEGIN
 END p1;

 PROCEDURE p2;
 BEGIN
 END p2;

 END local;

 BEGIN  (* p1 is visible here, but not p2 *)
 END Main.
```

NEW standard procedure (STORAGE)   S

Description   *NEW* allocates a dynamic variable from the heap.

Usage   NEW(p);

               or

         NEW(p, tag1, tag2, ..., tagn);

         p must be of type *POINTER*.

         ALLOCATE must be imported from the pseudomodule *STORAGE*.

         tag1, ..., tagn are allowed in the case where *p* points to a variant record, and the tags are defined in that record.

Comments   *NEW* is translated into an appropriate call to *ALLOCATE*.

         Turbo Modula-2 expects *ALLOCATE* to be defined the way it is in the pseudomodule *STORAGE*; however, you may substitute your own allocation scheme by importing *ALLOCATE* from your own module instead of *STORAGE*.

See Also   ALLOCATE
         DISPOSE
         STORAGE

Example   Allocate a new Name:

```
MODULE Allocate;
FROM STORAGE IMPORT ALLOCATE;
VAR
 n: POINTER TO Name;
BEGIN
 NEW(n);
END Allocate.
```

         The *NEW(n)* statement would be translated into a call of the form *ALLOCATE(n, TSIZE(Name))*.

NEWPROCESS procedure (SYSTEM)   L

Description   *NEWPROCESS* sets up an independent process (or coroutine).

Declaration   **PROCEDURE** NEWPROCESS(p: PROC; a: ADDRESS;
                   n: CARDINAL; **VAR** q: PROCESS);

Usage   NEWPROCESS(newProc, wrkAdr, wrkSize, P);

                   NEWPROCESS must be imported from the pseudomodule
                   SYSTEM.

                   newProc must be of the standard type *PROC* (a parameterless
                   procedure).

                   wrkAdr must be compatible with type *ADDRESS* (any pointer).

                   wrkSize must be of type CARDINAL. It specifies the number
                   of bytes of storage.

                   P must be of type *PROCESS* imported from *SYSTEM*.

Comments   The procedure *newProc* contains the code for the new process.

                   The workspace must be big enough to accommodate any local
                   stack and heap activity; 100 words is a good size to start with.

                   Local stack activity consists of local variables, procedure
                   parameters, and some space for working out expressions.

                   Local heap activity consists of the user's calls to *NEW* and
                   ALLOCATE and any file control blocks that are active. If the
                   allocated workspace is not enough, the exception
                   OutOfMemory will be raised.

                   NEWPROCESS allocates space and sets the corresponding
                   PROCESS variable, but it does not actually start up the
                   routines--*TRANSFER* accomplishes that.

The procedure *StartProcess* in *Processes* is implemented using this routine.

See Also   Processes
PROCESS
PROC
TRANSFER
SYSTEM

Example   Start up a driver called *Driver*:

```
MODULE NewProcess;
FROM SYSTEM IMPORT ADR, PROCESS, NEWPROCESS,
    TRANSFER;
VAR
 work: ARRAY [ 0..100 ] OF WORD;
 M,P: PROCESS;

PROCEDURE Driver;
BEGIN
 (* Driver's code *)
END Driver;

BEGIN
 NEWPROCESS(Driver, ADR(work), SIZE(work), P);
 TRANSFER(M,P); (* Driver now has control *)
END NewProcess.
```

NextPos procedure (Files)   L

Description   *NextPos* returns the current byte position in a file.

Declaration   **PROCEDURE** NextPos(f: FILE): LONGINT;

Usage   n := NextPos(f);

                         NextPos must be imported from the library module *Files*.

                         n must be of type LONGINT.

                         f must be of type *FILE* imported from *Files*.

Comments   The result of *NextPos* is a LONGINT because file can be
                         larger than *MAX(CARDINAL)*.

See Also   FileSize
                         SetPos

Example   Keep track of the percentage of file you have completed:

```
MODULE NextPosPercent;
FROM Files IMPORT FILE, Open, NextPos, ReadRec;
VAR
 f:File;
 totsize: LONGINT;
 wreck: SomeRecord;
BEGIN
 IF NOT Open(f,"reckfile") THEN HALT END ;
 totsize := FileSize(f);
 WHILE NOT EOF(f) DO
  ReadRec(f, wreck);
  n := (100L*NextPos(f)) / totsize;
  WRITE('Finished ',n,' % of the file ')
  (* Process wreck *)
  END
END NextPosPercent.
```

NIL standard value   S

Description	*NIL* designates an unused pointer variable.
Usage	`BufferPtr := NIL;`
	BufferPtr is any pointer type.
	NIL is compatible with all **pointer variables**.
Comments	The actual value of *NIL* is zero; thus, *WRITE(CARDINAL(NIL))* will print a 0.
See Also	POINTER

Example

```
MODULE NilList;
FROM STORAGE IMPORT ALLOCATE;
TYPE
 NodePtr = POINTER TO Node;
 Node =   RECORD
            element: CARDINAL;
            Next: NodePtr;
            END ;
VAR
 l: NodePtr;
BEGIN
 NEW(l); l^.Next := NIL;
END NilList.
```

Normal procedure (Terminal)   L

Description	*Normal* turns off highlighting.
Declaration	**PROCEDURE** Normal;
Usage	`Normal;`
	Normal must be imported from the library module *Terminal*.

Comments　　　The highlighting will only work if Turbo Modula-2 has been properly installed. If highlighting has not been turned on, then the effect of *Normal* is undefined (it depends on your terminal, but it is probably harmless).

Example　　　　Highlight the word 'high':

```
WRITE( 'This is a example of ' );
Highlight;
WRITE( 'High' );
Normal;
WRITE( 'lighting' );
```

NoTrailer procedure (Files)　　　L

Description　　　*NoTrailer* turns off the special Turbo Modula-2 end-of-file handling.

Declaration　　　**PROCEDURE** NoTrailer(f: FILE);

Usage　　　　　NoTrailer(f);

　　　　　　　NoTrailer must be imported from the library **module *Files*.**

　　　　　　　f must be of type *FILE* imported from *Files*.

Comments　　　Turbo Modula-2 handles the last byte of the last record of a CP/M file so that a program can determine exactly how many bytes are in a file.

　　　　　　　The last byte of the last record contains the length of the last record plus 128. Thus, if the last byte is 127 or less, then the record is completely filled; otherwise, the number of bytes in the last record is found by subtracting 128 from the last byte. This is how the procedure *FileSize* works.

　　　　　　　If *NoTrailer* is used on a file, then *FileSize* will always return a multiple of 128.

NoTrailer should be used immediately after *Open* when reading; it can be used immediately after *Create* as well.

In general, this function should only be used if you are having trouble with file communication involving a non- Turbo Modula-2 program.

See Also   Files
                  FileSize

(ımple   Declare a file to have no trailer:

```
IF Open(f,filename)THEN
 NoTrailer(f);
 (* process the file *)
END ;
```

---

numCols variable (Terminal)   L

Description   *numCols* holds the number of columns available on the currently installed terminal.

Declaration   **VAR**
                  numCols: CARDINAL;

Usage   IF CurCol > numCols **THEN** CurCol := 0 **END** ;

                  numCols must be imported from the module *Terminal.*

Comments   This variable is useful for writing terminal-independent programs. The value of *numCols* is set by the installation program INSTM2.

See Also   numRows
                  Terminal

Example   **MODULE** ColumnNumber;
                        FROM Terminal **IMPORT** numCols;
                        VAR
                         I: CARDINAL;
                        BEGIN
                         I := 0;
                         WHILE I < numCols **DO** WRITE('-') **END** ;
                        END ColumnNumber.

numRows variable (Terminal)   L

Description   *numRows* holds the number of rows available on the current
                        installed terminal.

Declaration   **VAR**
                         numRows: CARDINAL;

Usage   **IF** CurRow1 > numRows **THEN** CurRow := 0 **END** ;

                        numRows must be imported from the module *Terminal.*

Comments   This variable is useful for writing terminal-independent p:
                        grams. The value of *numRows* is set by the installation progr;
                        INSTM2.

See **Also**   numCols
                        Terminal

Example   **MODULE** RowNumber;
                        FROM Terminal **IMPORT** numRows;
                        VAR
                         I: CARDINAL;
                        BEGIN
                         I := 0;
                         WHILE I < numRows **DO** WRITELN('|') **END** ;
                        END RowNumber.

ODD standard function   S

Description   *ODD* returns TRUE if the ordinal value of *X* is odd.

Usage   Y:= ODD(X);

Argument *X* is of type CARDINAL or INTEGER.

Result *Y* is of type BOOLEAN.

Comments   This function allows you to find out if a value is odd.

Note that for BOOLEAN types, the first value is FALSE with an ordinal value of 0.

Example   **MODULE** Odd;
```
TYPE
 Shift = (Early,Late,Night);
VAR
 TimeSheet    : ARRAY Shift OF CARDINAL
 ShiftToday   : Shift;
 OddNumber    :BOOLEAN;
BEGIN
 TimeSheet[Early]:= 11;
 OddNumber:= ODD(TimeSheet[Early]);
(* OddNumber is TRUE*)
END Odd.
```

Open procedure (Files)   L

Description   *Open* opens a binary disk file for input or output.

Declaration   **PROCEDURE** Open(**VAR** f: FILE; name: **ARRAY OF CHAR**):
BOOLEAN;

Usage   `okay := Open(f, name);`

Open must be imported from the library module *Files*.

okay must be declared as type BOOLEAN.

f must be declared as type *FILE* imported from *Files*.

name must be declared as ARRAY OF CHAR.

Comments   *f* is the internal file identifier. Further operations on this file
must specify *f*.

name must represent a legal CP/M file name that is on the disk
you are using or a standard CP/M device like RDR: or CON:.

okay is returned TRUE when the file can be opened; it is
FALSE if the file could not be found.

Use *Create* to start a new file.

Readable files are better handled using routines from the
module *Texts*.

See Also   Close
Create
Files

Example   Open and close a disk file 'SCRATCH.SCR':

```
MODULE openandclose;
FROM Files IMPORT FILE, Open, Close;
VAR
 f  : FILE;
BEGIN
 IF Open(f,'SCRATCH.SCR') THEN
  Close(f)
 END
END openandclose.
```

OpenInput Procedure (InOut)   L

Description   *OpenInput* asks the user for a disk file name, then opens it as standard input.

Declaration   **PROCEDURE** OpenInput(defext: **ARRAY OF** CHAR);

Usage   OpenInput(ext);

    OpenInput must be imported from the library module *InOut*.

    ext must be a character array.

Comments   Notice that calling this procedure causes the program to halt and ask the user for a file name. The *ext* parameter will be used only if the user does not specify an extension.

Example
```
MODULE Input;
FROM InOut IMPORT
 OpenInput, CloseInput, ReadString,
 WriteString, WriteLn;
VAR
 s: ARRAY [0..255] OF CHAR;
BEGIN
(* Prompt user for filename *)
(* Use default extention MOD *)
 OpenInput("MOD");
 (* Read a string from the file *)
 ReadString(s);
 WriteString(s);
(* Display it *)
 WriteLn;
 CloseInput;
(* Sever the link to the file *)
 (* Now input comes from the console *)
 ReadString(s);
 WriteString(s);
(* Display it *)
 WriteLn;
END Input.
```

OpenOutput procedure (InOut)   L

Description	*OpenOutput* asks the user for a disk file name, then opens it as the standard output.
Declaration	**PROCEDURE** OpenOutput(defext: **ARRAY OF** CHAR);
Usage	OpenOutput(ext);
	OpenOutput must be imported from the library module *InOut*.
	ext must be a character array.
Comments	Notice that calling this procedure causes the program to halt and ask the user for a file name. The *ext* parameter will be used only if the user does not specify an extension.

Example

```
MODULE Output;
FROM InOut IMPORT
 OpenOutput, CloseOutput, ReadString,
 WriteString, WriteLn;
VAR
 s: ARRAY [0..255] OF CHAR;
BEGIN
 WriteString('Enter output file and then the data:');
 WriteLn;
 OpenOutput("MOD");
 (* Read data from the console *)
 ReadString(s);
 (* Place it in the file *)
 WriteString(s);
 WriteLn;
 CloseOutput;   (* Sever the link to the file *)
 WriteString('The file has been closed.');
 WriteLn;
END Output.
```

OpenText procedure (Texts)   L

Description	*OpenText* associates a *TEXT* with an external medium and opens it for reading.
Declaration	**PROCEDURE** OpenText(**VAR** t: TEXT; name: **ARRAY OF** CHAR): BOOLEAN;
Usage	okay := OpenText(t, fname);

OpenText must be imported from the library module *Texts*.

t must be declared as type *TEXT* imported from *Texts*.

fname must be a character array.

okay must be declared as type BOOLEAN.

Comments   *fname* must either be a valid CP/M file name and be present on the appropriate disk or represent an existing CP/M device like 'CON:'.

okay is returned FALSE if the file is not found; otherwise, the file is sucessfully opened and *okay* is returned TRUE.

t can only be used for input.

See Also   CreateText
                TEXT
                Texts

Example   Open a text file 'MYTEXT.TXT' and then close it:

```
MODULE openandclose;
FROM Texts IMPORT TEXT, OpenText, CloseText;
VAR
 t  : TEXT;
BEGIN
 IF OpenText(t, 'MYTEXT.TXT') THEN
  CloseText(t)
  END
END openAndClose.
```

OpSet type (Terminal)   L

Description   *OpSet* holds the set of screen operations available on the
                      rently installed terminal.

Declaration   ```
TYPE
 SpecialOps = (clearEol, insertDelete,
 highlightnor mal);
 OpSet : SET OF SpecialOps;
```

Usage   ```
VAR
 available : OpSet;
```

                      *available* is a predefined variable declared as type *OpS*
                      must be imported from the module *Terminal*.

Comments              This variable is useful for writing terminal-independent
                      grams. The value of *available* is set by the installation pro
                      INSTM2.

See Also              SpecialOps
                      available
                      Terminal

Example           

```
MODULE TermStuff;
FROM Terminal IMPORT
 available, SpecialOps, Highlight, Normal;
BEGIN
 IF highlightNormal IN available THEN
  Highlight
 ELSE
 WRITE('"');
 END ;
 WRITELN('This is highlighted text!');
 IF highlightNormal IN available THEN
  Normal
 ELSE
 WRITE('"');
 END ;
END TermStuff.
```

OR   R

Description	OR is a binary logical operator resulting in a Boolean value.
Usage	IF InputError OR TimeIsUp THEN Beep END ;
	OR is a reserved word.
	InputError and *TimeIsUp* are Boolean expressions.
Comments	OR is used to concatenate two logical expressions. An OR expression is TRUE if at least one of its subexpressions is TRUE.
	Note that the second subexpression in an OR expression is only evaluated if the first subexpression is FALSE. This is called short-circuit evaluation.
See Also	AND NOT

Example   REPEAT statement using **OR** to test either of two terminatin
                conditions.

```
MODULE Delay;
FROM Terminal IMPORT BusyRead;
VAR
 ch: CHAR;
 x : CARDINAL;
BEGIN
 x := 0;
 REPEAT
  INC(x);
  BusyRead(ch);
 UNTIL ch # 0C OR x > 1000;
END Delay.
```

ORD standard function   S

Description   *ORD* returns the ordinal number of *X*.

Usage   Y:= ORD(X);

                Argument *X* is of type enumeration, INTEGER, CARDINA
                BOOLEAN, or CHAR.

                Result is of type CARDINAL.

Comments   Ordinal types have an ordered sequence of values, with
                position in the sequence being the ordinal number; thus,
                first value has an ordinal of 0, the next has ordinal of 1, ǎ
                so forth.

                Note that INTEGER values are written in two's complem
                format so that the lower ordinals correspond to positive va
                and higher ordinals to negative values.

                Ordinals   0 to 32767   Values   0 to 32
                Ordinals   32768 to 65535   Values   -32768t

For BOOLEAN types, the first value is FALSE with ordinal 0.

The inverse operation from ordinal to value uses the *VAL* function, or you can use the *CHR* function for character variables.

Example

```
TYPE Direction: (Forward,Backward,Up,Down);
VAR Move: Direction;
    Value:INTEGER
    Ordinal:CARDINAL

BEGIN
 Move:= Up;
 Ordinal:= ORD(Move);
(* Ordinal is value 2 *)
 Value:= 10; Ordinal:= ORD(Value);
(* Ordinal is value 10 *)

 Ordinal:= ORD("?")
(* Ordinal is value 63 *)
```

OUT procedure (SYSTEM)   L

Description   *OUT* outputs a word or a byte to a given I/O port.

Declaration   **PROCEDURE** OUT(port,data: WORD);

Usage   OUT(port, outWord);

OUT must be imported from the library module *SYSTEM*.

outWord must be compatible with type *WORD* imported from *SYSTEM*.

port can be any scalar type compatible with *WORD*.

Comments   The value given by *outWord* is output to the port whose number is given by *port*.

See Also	INP
	SYSTEM

Example   Output a byte to port 23 hex:

```
OUT(023H, outWord);
```

outName variable (ComLine)   L

Description	*outName* is a string variable that contains a string found at the output redirection symbol (>) on the command line.
Declaration	**VAR** outName: **ARRAY** [0..19] **OF** CHAR;
Usage	WRITE('Output redirected to ',outName);
Comments	The redirection symbol and *outName* will not appear in text *commandLine*.
	outName contains the standard output string »CON:« if redirection argument has been found on the command lin
See Also	ComLine
	inName
	RedirectOutput

Example
```
MODULE RedirectOutput;
FROM ComLine IMPORT outName, RedirectOutput;
FROM Texts IMPORT console;
BEGIN
 RedirectOutput;
 IF outName < > "CON:" THEN
  WRITE(console,'Output redirected to ',outName
 END
END RedirectOutput
```

OUTOFMEMORY exception (SYSTEM)   L/E

Description   OUTOFMEMORY is raised by the module *SYSTEM* when an allocation operation makes the stack and heap collide.

Declaration   **EXCEPTION**
                     OUTOFMEMORY;

Usage   **EXCEPTION**
                     OUTOFMEMORY:
                       WRITELN('Out of memory.');
                     END

                     OUTOFMEMORY must be imported from **the module** *SYSTEM*.

Comments   This exception **may occur during explicit allocation** or procedure calls.

See Also   SYSTEM
                     BADOVERLAY
                     OVERFLOW
                     REALOVERFLOW

Example   **MODULE** OutOfMemory;
                     FROM SYSTEM **IMPORT** OUTOFMEMORY;
                     FROM STORAGE **IMPORT** ALLOCATE, FREEMEM;
                     VAR
                      p: **POINTER TO** CHAR;
                     BEGIN
                      ALLOCATE(p,FREEMEM( )+100);
                     EXCEPTION
                      OUTOFMEMORY:
                        WRITELN('Out of **memory.**');
                     END OutOfMemory.

output (Texts)   L

Description   *output* is a standard *TEXT* variable that can be redirected.

Declaration   **VAR** output: TEXT;

Usage   output

output must be imported from the library module *Texts* and ca
be used anywhere a *TEXT* can be used.

Comments   *output* is declared in *Texts* as a variable of type *TEXT* and
write-only.

output is always open and by default connected to the termin
It is as if the call *CreateText(output, »CON:«)* has taken pla
before the execution of every main module. The ca
CloseText(output) reestablishes this connection.

WRITE and *WRITELN* always use *output* as their default te
stream.

See Also   console
input
Texts

Example   Write the number 5A5A (=23130 decimal) to the output:

```
MODULE StandardOut;
FROM Texts IMPORT output;
BEGIN
 WRITELN(output,5A5AH );
END StandardOut.
```

OVERFLOW exception (SYSTEM)   L/E

Description	OVERFLOW is raised by the module *SYSTEM* when an integer or cardinal expression results in a value that is too large or too small to be represented by the result type.

Declaration

```
EXCEPTION
  OVERFLOW;
```

Usage

```
EXCEPTION
  OVERFLOW:
   WRITELN('Overflow.');
END
```

OVERFLOW must be imported from the module *SYSTEM*.

Comments

This exception occurs only during evaluation of integer expressions. Real expressions generate the exception REALOVERFLOW.

See Also

SYSTEM
BADOVERLAY
OUTOFMEMORY
REALOVERFLOW

Example

```
MODULE Overflow;
FROM SYSTEM IMPORT OVERFLOW;
VAR
 r: REAL;
BEGIN
 r := MAX(REAL) * 2.2;
EXCEPTION
 OVERFLOW:
  WRITELN('Overflow');
END Overflow.
```

POINTER type   R

Description	A pointer references a dynamic variable that can be created, operated on, and discarded during program execution.
Usage	`Pointer Type = POINTER OF Referenced Type;`
	Referenced Type can be any type.
Comments	If the precise data structure can only be defined while running a program, you could use pointers to enable the program to create the data structure. For example, suppose you want to record the name and job of each employee in a company. Assuming there will never be more than 100 employees, you could declare a fixed data structure of 100 records, or you could use pointer variables. The program creates new employee records as you enter data at the keyboard.

A pointer is bound to the referenced type and cannot point to any other type. For example, if a pointer type is declared as

`ArrayPointer = POINTER TO ARRAY [1..10] OF CHAR`

then a pointer of type ArrayPointer can only create, reference, and discard arrays of 10 CHAR components.

The standard procedures NEW and DISPOSE create and discard referenced variables.

It should be noted that NEW and DISPOSE are translated into calls to the standard procedures ALLOCATE and DEALLOCATE, which must be imported from the module STORAGE.

The referenced variable is designated by the ^ symbol, the dereferencing operator.

A pointer's reference can be assigned to another identical pointer; in fact, several pointers can reference the same variable.

A pointer can also be given the value NIL, which means the pointer does not reference anything; this is not the same as discarding variables.

A pointer requires 2 bytes of memory. The value NIL corresponds to 2 zero bytes.

Dynamic variables are stored in the heap area that is handled by procedures in module STORAGE.

Example

```
MODULE Company;
FROM STORAGE IMPORT ALLOCATE,DEALLOCATE;

TYPE
PersonPointer = POINTER TO Person;
Person = RECORD
            Name: ARRAY [1..20] OF CHAR;
            Job:  ARRAY [1..20] OF CHAR;
            Next: PersonPointer;
            END ;

VAR
     FirstPerson,LastPerson,NewPerson:
     PersonPointer;
  Name: ARRAY [1..20] OF CHAR;

BEGIN
 FirstPerson:= NIL;
REPEAT
  WRITE('Enter name: ');
  READLN(Name);

IF Name < > '' THEN
(* Create a person record if name entered *)
   NEW(NewPerson);
   NewPerson ^ .Name:= Name;
   WRITE('Enter profession: ');
   READLN(NewPerson ^ .Job);
   WRITELN;
```

```
    IF FirstPerson = NIL THEN
    FirstPerson:= NewPerson
(* FirstPerson references first in list *)
    ELSE
     LastPerson ^ .Next:= NewPerson
(* Reference to next in list *)
    END ;
    LastPerson:= NewPerson;
(* LastPerson references last in list *)
    LastPerson ^ .Next:= NIL
   END
 UNTIL Name = '';
 WRITELN;
(* Write all records on screen *)
 WHILE FirstPerson < > NIL DO
  WITH FirstPerson ^ DO
   WRITELN(Name,' is a ',Job);
   FirstPerson:= Next;
  END
 END
END Company.
```

Pos procedure (Strings)   L

Description   *Pos* finds the position of a substring in a string.

Declaration   **PROCEDURE** Pos(substr,str: **ARRAY OF** CHAR): CARDINAL

Usage   p := Pos(substr, str);

Pos must be imported from the library module *Strings*.

substr and *str* must be character arrays.

p must be of type CARDINAL.

Comments   The first character has the position 0, the second 1, and so on.

If the substring is not found, the value *HIGH(str)+1* is returned.

See Also   Copy
Delete
Insert
Length
Strings

Example

```
VAR
 str: ARRAY [0..30] OF CHAR;
 index:CARDINAL
BEGIN
 str    := 'This is a little test string';
 index  := Pos('This', str);
(* index now equals 0 *)

 index   := Pos('little', str);
(* index now equals 10 *)

 index   := Pos('big', str);
(* index now equals 31 *)
```

PROC standard type   S

Description   *PROC* is a predefined parameterless procedure type

Usage   PROC = PROCEDURE ;

Comments   *PROCESS* is an opaque type used to represent an independen process. In reality, it is nothing more than a storage area fo a program counter and local heap and stack pointers. The use can manipulate variables of type *PROCESS* only through routines exported from *SYSTEM* and *Processes*. Notice that th module *Processes* is implemented using the primitives defined in *SYSTEM*.

The procedure *NEWPROCESS* sets up variables of type *PRO CESS*, and the routines *TRANSFER* and *IOTRANSFER* provid cooperative and interrupt driven control functions.

See Also   TRANSFER
NEWPROCESS
Processes

Example

```
MODULE Process;
VAR
 M,P: PROCESS;
 WorkSpace: ARRAY [0..99] OF CARDINAL;
PROCEDURE PP;
VAR
 i: CARDINAL
BEGIN
 FOR i:= 1 TO 10000 DO END ;
 TRANSFER (P,M);
END PP;
BEGIN
 NEWPROCESS(PP,ADR(WorkSpace),SIZE(WorkSpace),P)
 TRANSFER (M,P);
END Process.
```

Comments   PROCEDURE may be nested to any depth and may enclose
             modules. If a procedure has a local module, then that module
             is only in existence when the procedure is active. Thus, the
             module's initialization part is executed every time the pro-
             cedure is called.

Example   A procedure:

```
PROCEDURE AmI;
BEGIN
 WRITELN ( 'I am ' );
END AmI;
```

             A function:

```
PROCEDURE NotZero ( i :INTEGER) :BOOLEAN
BEGIN
 RETURN ( i # 0 );
END NotZero;
```

PROCEDURE type   R

Description   **PROCEDURE allows procedures to be assigned to variables.**

Usage   **TYPE = PROCEDURE** [Parameter Types] [:Result Type];

             Parameter types and the result type can be any type.

Comments   Procedure types allow you to assign arguments to a variable.
             You can use procedure-type variables if you want to call a pro-
             cedure that has another procedure as a parameter. For exam-
             ple, suppose you define a procedure type as

             TYPE StringProc = **PROCEDURE(ARRAY OF** CHAR);

             This declares a procedure type *StringProc*. Any variable of
             type *StringProc* can take as argument an **ARRAY OF** CHAR.
             Note that nothing is mentioned about the contents of the pro-
             cedure.

Suppose you have a procedure with *StringProc* as an argument:

PROCEDURE Example(X:CARDINAL P: StringProc);
 :
END Example;

This procedure expects two arguments: one is a CARDINAL, the other is a procedure of type *StringProc* with **ARRAY OF CHAR** parameter.

The procedure *Example* can be called with any parameters of these types. For example:

Example(12,OpenInput(".INP"));

Example(21,OpenOutput(".DTA"));

Example(3,WriteString("Input data: "))

These three examples all call procedure *Example* with the two expected arguments: a CARDINAL and a procedure with an **ARRAY OF CHAR** argument. The procedures *OpenInput, OpenOutput*, and *WriteString* are from the module *InOut*.

Note that standard procedures such as *EXCL* or *HALT* cannot be assigned to procedure variables, nor can procedures that are local to other procedures.

The predefined procedure type *PROC* denotes a procedure with no parameters.

Example

```
MODULE Procedures;
FROM InOut IMPORT WriteCard,WriteHex;
TYPE
 ListPointer = POINTER TO List;
 List = RECORD
          Key:CARDINAL
          Data:CARDINAL
          Next: ListPointer
        END ;
 ProcType = PROCEDURE (CARDINAL,CARDINAL);
VAR First: ListPointer;
PROCEDURE Search(X:CARDINAL Q: ProcType);
(* Search for record with Key X *)
 VAR P: ListPointer;
BEGIN
 P:= First;
 WHILE (P # NIL) AND (P ^ .Key # X) DO
  P:= P ^ .Next
 END ;
 IF P # NIL THEN
  Q(P ^ .Data,6)
 END
END Search;
BEGIN
 Search(3,WriteCard);
(* Write data from record with Key 3 asCARDINAL*)

 Search(3,WriteHex);
(* Write same data as a Hexadecimal value *)
END Procedures.
```

PROCESS type (SYSTEM)   L

Description   *PROCESS* declares processes or coroutines.

Usage   p : PROCESS;

PROCESS must be imported from the library module *SYSTEM*.

Comments   *PROCESS* is an opaque type used to represent an independe
                  process. In reality, it is nothing more than a storage area 1
                  a program counter and local heap and stack pointers. The us
                  can manipulate variables of type *PROCESS* only throu
                  routines exported from *SYSTEM* and *Processes*. Notice that t
                  module *Processes* is implemented using the primitives defin
                  in *SYSTEM*.

                  The procedure *NEWPROCESS* sets up variables of type *PR
                  CESS*, and the routines *TRANSFER* and *IOTRANSFER* provi
                  cooperative and interrupt driven control functions.

See Also   TRANSFER
                  NEWPROCESS
                  Processes

Example   **MODULE** Process;
                  VAR
                   M,P: PROCESS;
                   WorkSpace: **ARRAY** [0..99] **OF** CARDINAL;
                  PROCEDURE PP;
                  VAR
                   i: CARDINAL
                  BEGIN
                   FOR i:= 1 **TO** 10000 **DO END** ;
                   TRANSFER (P,M);
                  END PP;
                  BEGIN
                   NEWPROCESS(PP,ADR(**WorkSpace**),SIZE(**WorkSpace**),P)
                   TRANSFER (M,P);
                  END Process.

Processes module   L

Description	*Processes* provides standard multiprogramming capabilities.

Declaration

```
DEFINITION MODULE Processes;

TYPE SIGNAL;

PROCEDURE StartProcess(P: PROC; n:CARDINAL;
PROCEDURE SEND(VAR s: SIGNAL);
PROCEDURE WAIT(VAR s: SIGNAL);
PROCEDURE Awaited(s: SIGNAL):BOOLEAN;
PROCEDURE Init(VAR s: SIGNAL);

EXCEPTION DeadLock;

END Processes.
```

Comments

Notice that there are no scheduling facilities associated with this module; the burden of providing these falls on the programmer.

Processes is implemented using the type *PROCESS* and the routines *NEWPROCESS* and *TRANSFER* in pseudomodule *SYSTEM*.

See Also

TRANSFER
PROCESS
NEWPROCESS
Also see the individual identifiers exported from *Processes*.

progName variable (ComLine)   L

Description	*progName* is a string variable that contains the name of program currently running.

Declaration   **VAR**
               progName: **ARRAY** [0..7] **OF** CHAR;

Usage   WRITE('The currently running program is ',progNa

Comments   *progName* allows for an eight-character name (assum
               .COM extension).

               progName is only valid if the program is started from w
               the Turbo Modula-2 shell. If the program is started from
               operating system, the string is empty.

See Also   ComLine

Example   **MODULE** RunFromTheShell;
               FROM ComLine **IMPORT** progName;
               VAR
               x,y: REAL;
               BEGIN
               IF progName < > '' **THEN**
                WRITELN('This program is called ',progName);
               END
               END RunFromTheShell.

PromptFor procedure (ComLine)   L

Description	*PromptFor* looks for an input string from the comman if there is not one present, it prompts the user to provid

Declaration   **PROCEDURE** PromptFor(prompt: **ARRAY OF**
               CHAR; **VAR** s: **ARRAY OF** CHAR);

Usage	`PromptFor(prompt, answer);`

PromptFor must be imported from the library module *Com-Line*.

prompt and *answer* must be of type ARRAY OF CHAR.

Comments	This procedure essentially performs a *ReadString* from the *TEXT commandLine* exported by *ComLine*. If an *EOT* (32C) is encountered in *commandLine*, then the user is prompted with *prompt* to provide additional input.

Note that in the Turbo Modula-2 system a command line can be specified by the user with the Run command from inside the Turbo Modula-2 shell.

Example	

```
MODULE ptest;
FROM ComLine IMPORT PromptFor;
BEGIN
 FOR i := 1 to 3 DO BEGIN
  WRITE(i:3,' ');
  PromptFor('Need more>', str);
  WRITELN(str);
 END ;
END ptest.
```

If the user starts the program with

`Run MCD file: Ptest arg1 arg2`

then the output will be as follows:

```
1 ARG1
2 ARG2
3 Need more> arg3
```

QUALIFIED   R

Description   QUALIFIED forces an identifier exported from a lo
                   module to be qualified in the surrounding environment.

Usage   **EXPORT QUALIFIED** this, that, theNextThing;

Comments   When an identifier is qualified, it means that every refere
                   must be prefixed by the name of the module it has been defi
                   in, followed by a period (.). For example, these identifiers
                   qualified: *Texts.ReadChar, MathLib.Cos, Files.EOF. /*
                   these are not: *ReadChar, Cos, EOF.*

                   This command only makes sense for local modules, si
                   main modules and implementation modules can't ex|
                   anything , and in definition modules everything is exp
                   qualified by default.

Example   *pl* in *local* is EXPORT QUALIFIED:

                   MODULE main;

                     MODULE local;
                     EXPORT QUALIFIED pl;

                     PROCEDURE pl;
                     BEGIN END ;

                     END local;

                   BEGIN
                     local.pl; (* pl is visible, but must be qualifi‹
                   END main.

RAISE statement   R/E

Description	*RAISE* causes an exception to be raised, passing control up the calling chain to an exception handler or the runtime system.
Usage	**RAISE** [SomeError] [, message];
	message must be a character array.
Comments	When an exception is raised, the current procedure is searched for the appropriate handler. If one is found, the specified code is executed. If a handler for the exception is not found, the caller of the current procedure is searched. This search process continues until a handler is found or the runtime supervisor is reached. The supervisor prints the exception name on the terminal along with the optional message and the calling chain, control is then returned to the operating system.
	The short form of the exception statement is **RAISE**, which may only be used within an exception handler. If this statement occurs in an exception handler, then the same exception that the exception handler is currently handling will be raised again in the next highest procedure of the calling sequence.
See Also	EXCEPTION
Example	Raise an exception to be caught somewhere:
	RAISE MyError
	Raise an exception with a message:
	RAISE TooLateAtNight, 'Turn me off and go home !'

Do a little bit of exception-handling and then pass it on:

```
EXCEPTION
 | DiskFull:
   WRITELN('DiskFull in thisProc');
   Close(f);
   RAISE ;
 ELSE
   RAISE ;
END thisProc.
```

Random function (MathLib)   L

Description   *Random* returns a random number between 0 and 1.

Declaration   **PROCEDURE** Random( ): REAL;

Usage   Y:= Random( );

                  Result *Y* is REAL.

                  There is no argument, but the parentheses are required.

Comments   The function returns the next element from a pseudo-ran(
                  number sequence. The same sequence is repeated each
                  your program is executed unless you select a new sequ
                  with the procedure *Randomize*.

Example   Two ways to randomize the random number generator:

```
FROM MathLib IMPORT Randomize;
VAR Sequence:CARDINAL
BEGIN
 WRITELN('Input sequence number, 0 to 65535');
 READ(Sequence);
 Randomize(Sequence);

    or

i := 0;
WRITE('Enter any character');
REPEAT
 i := i + 1;
 BusyRead( ch );
UNTIL ch # 0C;
Randomize( i );
```

Randomize procedure (MathLib)   L

Description   *Randomize* initializes a pseudo-random-number sequence. The same random sequence can be reproduced by passing the same seed value to *Randomize*.

Declaration   **PROCEDURE** Randomize(n:CARDINAL);

Usage   Randomize(1986);

                 Randomize must be imported from the module *MathLib*.

Comments   After randomizing the sequence, call *Random* to get the random numbers.

See Also   MathLib
                 Random

Example   **MODULE** RandomMath;

```
MODULE RandomMath;
FROM MathLib IMPORT Randomize, Random;
VAR x: REAL;
BEGIN
 Randomize(1986);
 x := Random( )*10.0/Random( )*20.0;
END RandomMath.
```

READ standard procedure   S/E

Description   *READ* is a generalized read statement.

Usage   READ(t, vlist)

          or

          READ(vlist);

          t is of type *TEXT* imported from TEXTS. *vlist* is simply a
          of variables and constants.

Comments   *READ* will be translated by the compiler into the approp
          calls to the module *Texts*.

          Notice that you do not have to import procedures from ?
          in order to use these commands, but *Texts* must be somew
          on disk.

          Although *READ*, *READLN*, *WRITE*, and *WRITELN* are n
          the standard Modula-2 definition, programming without t
          is much more tedious. However, if a program is to be port
          do not use these commands. A good strategy would b
          develop the program using these statements for debugging
          put and explicitly typed calls to *Texts* for the user's outpu
          move the program to another enviroment, only the mc
          Texts must be duplicated in the new enviroment.

See Also	READLN
	Texts
	WRITE
	WRITELN

Example	Read an integer *i*:

READ(i)

Read a string *str* followed by 4 integers *i,j,k,l*:

READ(str, i,j,k,l)

Read a real number *r* from *TEXT t*:

READ(t, r)

ReadAgain procedure (Terminal)   L

Description	*ReadAgain* causes the last character read from the terminal to be repeated.
Declaration	**PROCEDURE** ReadAgain;
Usage	ReadAgain;
	ReadAgain must be imported from the library module *Terminal*.
Comments	The last character read before this procedure is called will be picked up by the next read operation from the terminal.
	This procedure works with all the input procedures in *Terminal*, but only works if at least one character has already been processed since the program began.
	With this routine you can achieve a certain amount of lookahead without having to set up a global buffer and complicated logic to use is.

See also	BusyRead
	ReadChar
	Terminal

Example   Read a character and then read it again:

```
ReadChar(ch);
ReadAgain;
ReadChar(ch);
```

ReadAgain procedure (Texts)   L

| Description | *ReadAgain* causes the last character read from a *TEXT* to repeated. |

| Declaration | **PROCEDURE** ReadAgain(t: TEXT); |

| Usage | ReadAgain(t); |

ReadAgain must be imported from the library module *T*

t must be of type *TEXT* imported from *Texts*.

| Comments | The last character read from *t* before this procedure is ca will be picked up by the next read operation from *t*. |

This procedure works with all the input procedures in *T* but only works if at least one character has already been cessed since the program began.

With this routine, you can achieve a certain amoun lookahead without having to set up a global buffer and plicated logic to use it.

| **See Also** | ReadChar |
| | Texts |

Example Read a character **from the *TEXT* t and then read it** again:

```
ReadChar (t, ch);
ReadAgain(t);
ReadChar (t, ch);
```

Or, using the *ReadAgain's* potential lookahead to good advantage, you can read a value as an integer or a string depending on the first nonblank character:

```
REPEAT
 ReadChar(t,ch);
UNTIL ch # '' '' ;
ReadAgain(t);
IF (ch > = '' 0 '') and (ch < = '' 9 '') THEN
 ReadInt(t,x)
ELSE
 ReadString(t,s)
END ;
```

ReadByte procedure (Files) L

Description *ReadByte* reads a variable of any type compatible with *BYTE* from *FILE*.

Declaration **PROCEDURE** ReadByte(f: FILE; **VAR** ch: BYTE);

Usage ReadByte(f, ch);

ReadByte must be imported from the library module *Files*.

f must be of type *FILE* imported from *Files*.

ch must be of a type compatible with *BYTE*.

Comments	Types compatible with *BYTE* are those whose storage uses exactly 1 byte. These include CHAR, BOOLEAN, and enumeration types with at most 256 elements; subrange types with bounds within the range 0 to 255, and word- sized variables as well (but only 1 byte is read).
See Also	Files ReadWord ReadRec WriteByte
Example	Open a file 'ONEBYTE.DAT' and write out a byte; close it, and read it back:

```
MODULE writebyte;
FROM Files IMPORT FILE, Create, Close, ReadByte,
                  WriteByte;
VAR  f  : FILE;
     ch : CHAR;
BEGIN
 Create(f, 'ONEBYTE.DAT');
 WriteByte(f,69);
 Close(f);
 IF Open(f, 'ONEBYTE.DAT') THEN
  ReadByte(f, ch);
  (* ch now contains the character 'E'
  ( ascii code 69 ) *)
  Close(f);
END writebyte.
```

ReadBytes procedure (Files)   L

Description	*ReadBytes* reads a number of bytes into a block of memor beginning at an address.
Declaration	**PROCEDURE** ReadBytes(f: FILE; buf: ADDRESS; nbytes CARDINAL):CARDINAL;

Usage	`nread := ReadBytes( f, bufadr, nbytes );`

ReadBytes must be imported from the library module *Files*.

f must be of type *FILE* imported from *Files*.

buf must be compatible with type *ADDRESS* (any pointer).

nread and *nbytes* must be of type CARDINAL.

Comments   *ReadBytes* will attempt to read *nbytes* from *f.* It returns the number of bytes successfully read. It will not raise the esception *EndError*, even if the end of the file is encountered before *nbytes* bytes are processed.

The variables are read into **memory starting at the location** specified in *bufadr.*

ReadBytes and *WriteBytes* are low-level routines with which you can take snapshots of memory to and from the disk, copy files, and so on. They must be used with care, since you can easily destroy your program or its variables with them. When transferring variables, we recommend using an explictly typed procedure like *ReadRec*, *ReadWord*, and the like.

See Also   FILE
Files
WriteBytes

Example   Read 1024 bytes from *f* into memory starting at 8000 Hex:

`uread := ReadBytes(f, 8000H, 1024)`

The previous example is not very safe since the programmer doesn't always know what is at a given address in memory. The following is more useful:

`nbytes := ReadBytes(f, ADR(area), SIZE(area));`

ReadCard procedure (Texts)   L

Description	*ReadCard* reads a cardinal number from a *TEXT* file.
Declaration	**PROCEDURE** ReadCard(t: TEXT; **VAR** c: CARDINAL);
Usage	ReadCard(t, card);
	ReadCard must be imported from the library module *T*
	t must be of type *TEXT.*
	card must be of type **CARDINAL.**
Comments	In general, *READ* is easier to use.
	This function works by reading a string into an int variable and then converting it via *StrToCard* in *Conver*
	Leading blanks are skipped, and the range must be within to *MAX(CARDINAL).*
	The BOOLEAN variable *Done* (exported by *Texts*) can be to determine whether input was syntactically correc within the legal range. If *Done* is FALSE, the number ret by *ReadCard* is undefined.
See Also	READ Texts WriteCard
Example	Read a cardinal number called *c* from the file *t*:
	ReadCard(t, c)

ReadChar procedure (Terminal)　　L

Description	*ReadChar* reads a character from the terminal without echo.
Declaration	**PROCEDURE** ReadChar(**VAR** ch: CHAR);
Usage	ReadChar(c);
	ReadChar must be imported from the library module *Terminal*.
	c must be of type **CHAR**.
Comments	The character is not echoed to the screen.
	Line ends are returned as carriage returns, CHR(13).
See Also	BusyRead
	Terminal
	WriteChar
Example	Read a character *ch* from the terminal without echo:
	ReadChar(ch)

ReadChar procedure (Texts)　　L

Description	*ReadChar* reads a character from a *TEXT* file.
Declaration	**PROCEDURE** ReadChar(t: TEXT; **VAR** ch: CHAR);
Usage	ReadChar(t, c);
	ReadChar must be imported from the library module *Texts*.
t	must be of type *TEXT*.
c	must be of type *CHAR*.

cts.

(

ernal

t.

zero

used
and
rned

(

Comments   In general, *READ* is easier to use than this procedure.

                A single character is read in; if the stream is *input*, then the
                character is echoed to the screen. For input without echo, use
                ReadChar from the module *Terminal*.

                Input from the console must always be terminated with a
                return.

See Also   READ
                WriteChar

Example   Read a character *c* from the terminal with echo:

                ReadChar(input, c)

                Read a character from the *TEXT* file *t*:

                ReadChar(t, c)

ReadDouble procedure (Doubles)   L

Description   *ReadDouble* reads a double-precision real variable from a
                TEXT file.

Declaration   **PROCEDURE** ReadDouble(t: TEXT; **VAR** d: LONGREAL);

Usage   ReadDouble(t, d);

                ReadDouble must be imported from the library module
                Doubles.

                t must be of type *TEXT* imported from *Texts*.

                d must be of type LONGREAL.

Comments	This procedure works by reading a string from *t* and then converting it to a LONGREAL using *StrToDouble*.
	The Boolean variable *legal* (exported from *Doubles*) is set to TRUE if a legal double-precision number is read from *t*.
	The *READ* and *READLN* procedures use *ReadDouble* to read LONGREALs.
See Also	WriteDouble
Example	Read a double-precision *d* out of a text *t*:

```
ReadDouble(t, d);
```

ReadInt procedure (Texts)   L

Description	*ReadInt* reads an integer from a *TEXT* file.
Declaration	**PROCEDURE** ReadInt(t: TEXT; **VAR** i: INTEGER);
Usage	ReadInt(t, i);
	ReadInt must be imported from the library module *Texts*.
	t must be of type *TEXT*.
	i must be of type INTEGER.

Comments	In general, *READ* is easier to use than this procedure.

This routine works by reading a string from text *t* and converting it to an INTEGER using *IntToStr* in *Convert*.

Leading blanks are skipped, and the range must be within *MIN(INTEGER)* to *MAX(INTEGER)*.

The BOOLEAN variable *Done* (exported by *Texts*) can be used to determine whether input was syntactically correct and within the legal range. If *Done* is FALSE, the number returned by *ReadInt* is undefined.

See Also   READ
Texts
WriteInt

Example   Read an integer from the text *t*:

ReadInt(t, i)

ReadLine procedure (Texts)   L

Description   *ReadLine* reads a whole line of text into a string variable.

Declaration   **PROCEDURE** ReadLine(t: TEXT; **VAR s: ARRAY OF** CHAR);

Usage   ReadLine(t, line);

ReadLine must be imported from the library module *Texts*.

t must be of type *TEXT* imported from *Texts*.

line must be an ARRAY OF CHAR.

Comments   *ReadLine* and *ReadString* are similar except that *ReadString* skips over leading blanks and *ReadLine* doesn't, and *ReadLine* is only terminated by an EOL or an EOT.

See Also	ReadString
	Texts

Example   Read from *TEXT t* a string *str* with leading blanks:

```
ReadLine(t, str)
```

ReadLine procedure (Terminal)   L

Description   *ReadLine* reads a line of text into a string variable.

Declaration   **PROCEDURE** ReadLine**(VAR** s: **ARRAY OF** CHAR);

Usage   `ReadLine(line);`

               ReadLine must be imported from the library module *Terminal*.

               line must be an **ARRAY OF CHAR**.

Comments   *ReadLine* is similar to *Texts.ReadLine*, but *ReadLine's* input is terminated by a carriage return instead of an EOT character.

See Also   Terminal

               Texts

Example   Read a string *str* from the console:

```
MODULE ReadLineEx;
FROM Terminal IMPORT ReadLine;
VAR
 s: ARRAY [0..20] OF CHAR;
BEGIN
 WRITELN('Enter a string and press return ');
 ReadLine(str);
 WRITELN('The string you entered is ',str);
END ReadLineEx;
```

READLN standard procedure   S/E

Description   *READLN* is a generalized read statement that terminates with
                 a new line.

Usage   READLN({t}, vlist)

                       or

                 READLN(vlist);

                 t is an optional *TEXT* file; the default is *input*.

                 vlist is simply a list of variables.

Comments   If *t* is omitted, the standard input will be used.

                 READLN will be translated by the compiler into the ap-
                 propriate calls to the module *Texts*. Thus do not change the
                 definition of the module *Texts* if you want to use this sort of
                 input and output.

                 READLN works just like *READ* except it skips over the text file
                 being read until an *EOL* is encountered.

See Also   READLN
                 Texts
                 WRITE
                 WRITELN

Example   Get a number from the user at the terminal:

                 WRITE('Gimme a number, doesn't matter which one > ');
                 READLN(number);

                 Read a bunch of numbers from a *TEXT* file:

                 READLN(t, num1, num2, num3, numlast);

ReadLong procedure (Texts)   L

Description	*ReadLong* reads a long integer from a *TEXT* file.
Declaration	**PROCEDURE** ReadLong(t: TEXT; **VAR** l: LONGINT);
Usage	ReadLong(t, l);

ReadLong must be imported from the library module *Texts*.

t must be of type *TEXT*.

l must be of type LONGINT.

Comments	In general, *READ* is easier to use than this procedure.

ReadLong works by reading a string from a *TEXT* file and converting it to a LONGINT using the procedure *StrToLong* from *Convert*.

Leading blanks are skipped, and the range must be within *MIN(LONGINT)* to *MAX(LONGINT)*.

The BOOLEAN variable *Done* (exported by *Texts*) can be used to determine whether input was syntactically correct and within the legal range. If *Done* is FALSE, the number returned by *ReadLong* is undefined.

See Also	READ
	Texts
	WriteLong
Example	Read from *TEXT* file *t* a long integer *l*:

ReadLong(t, l)

ReadReal procedure (Texts)   L

Description   *ReadReal* reads a real number from a *TEXT* file.

Declaration   **PROCEDURE** ReadReal(t: TEXT; **VAR** r: REAL);

Usage   ReadReal(t, r);

                   ReadReal must be imported from the library module *Texts*.

                   t must be of type *TEXT*.

                   r must be of type REAL.

Comments   In general, *READ* is easier to use than this procedure.

                   Leading blanks are skipped, and the range must be within the
                   legal real-number range *MIN(REAL)* to *MAX(REAL)*.

                   The BOOLEAN variable *Done* (exported by *Texts*) can be used
                   to determine whether input was syntactically correct and
                   within the legal range. If *Done* is FALSE, the number returned
                   by *ReadLong* is undefined.

See Also   READ
                   Texts
                   WriteReal

Example   Read a real number *r* from text file *t*.

                   ReadReal(t, r)

ReadRec procedure (Files)   L

Description   *ReadRec* reads a record, array, or real from a *FILE* (a binary
                   disk file).

Declaration   **PROCEDURE** ReadRec(f: FILE; **VAR rec: ARRAY OF** WORD);

Usage   ReadRec(f, r);

ReadRec must be imported from the library module *Files*.

f must be of type *FILE* imported from *Files*.

r can be of any type.

Comments   This routine can read any variable not covered by *ReadByte* and *ReadWord*. It reads variables compatible with **ARRAY OF WORD**; for example, REAL, LONGREAL, LONGINT, ARRAY, or RECORD.

See Also   Files
ReadByte
ReadWord
WriteRec

Example   Open a file 'SOMEWREK.DAT' and write out this record, then read it back:

```
MODULE writefile;
FROM Files IMPORT FILE, Create, Close, WriteRec,
ReadRec;
FROM wreck IMPORT wreckRec;
VAR
 f  : FILE;
 wr : wreckRec;
BEGIN
 Create(f, 'SOMEWREK.DAT');
 WriteRec(f, wr);
 Close(f);
 IF Open(f, 'SOMEWREK.DAT') THEN
  ReadRec(f, wr);
  (* wr has whatever it had before in it *)
  Close(f);
 END ;
END writefile.
```

ReadString procedure (Texts)   L

Description   *ReadString* reads a string, which in this case is a sequence of characters that does not contains blanks or control characters.

Usage   ReadString(t, s);

                     ReadString must be imported from the library module *Texts*.

                     t must be of type *TEXT* imported from *Texts*.

                     s must be of type **ARRAY OF CHAR.**

Comments   In general, *READ* is easier to use than this procedure.

                     The routine works by skipping over blanks until it gets to a nonblank character, at which point it starts to read. Input is then terminated by a blank or a control character (like *EOL* or *EOT*). Thus, strings cannot be separated by commas.

                     If you want to read in a line as it is, with leading blanks and all, use *ReadLine*.

See Also   ReadLine
                     TEXTS
                     WriteString

Example   Read a string, **ignoring leading blanks,** from a *TEXT* file *t*:

                     ReadString(t, str)

ReadWord procedure (Files)   L

Description   *ReadWord* reads a variable of any type compatible with *WORD* from a *FILE*.

Declaration   **PROCEDURE** ReadWord(f: FILE; **VAR** w: WORD);

Usage	`ReadWord(f, w);`

ReadWord must be imported from the library module *Files*.

f must be of type *FILE* imported from *Files*.

w must be of a type compatible with *WORD*.

Comments	Types compatible with *WORD* are those whose storage uses exactly one word (2 bytes); these include INTEGER, CARDINAL, BITSET and all pointers (ADDRESS is a pointer).

(

See Also	Files
	ReadByte
	ReadRec
	WriteWord

Example — Open a file 'SOMEYEAR.DAT' and write out this year for later:

```
MODULE writefile;
FROM Files IMPORT FILE, Create, Open, Close,
WriteWord,
ReadWord;
VAR
 f     : FILE;
 okay  :BOOLEAN
 y     : CARDINAL;
BEGIN
 Create(f, 'SOMEYEAR.DAT');
 WriteWord(f, 1985);
 Close(f);
 IF Open(f, 'SOMEYEAR.DAT') THEN
  ReadWord(f, y);
  (* y now has the value 1985 *)
 END ;
END writefile.
```

REAL standard type   S

Description   REAL is a standard type with variables that can assume any
                   value between -6.80565E+38 and 6.80565E+38.

Comments   You will use REAL variables whenever your computation is
                   likely to produce a fraction. For example, to record the mass
                   of liquid emitting from a pipe, use *MassLiquid:= RateOfFlow
                   * Density*.

                   In this case, none of the variables are likely to be whole
                   numbers so REAL types are appropriate.

                   It should be remembered that REAL computations are not
                   always exact; a certain amount of rounding occurs. For exam-
                   ple, an expression such as $Y:= 3.0 * (1.0/3.0)$ will not give the
                   expected result of 1.0, but a value slightly less. In practice, the
                   error is so small that it can usually be ignored. But don't make
                   the following mistake:

```
Y:= 3.0 * (1.0/3.0);
IF Y = 1.0 THEN
  :
```

                   The correct solution is

```
IF (Abs(Y) - 1.0) < 0.01 THEN
  :
```

                   If *Y* is between 0.99 to 1.01, the statement sequence will be ex-
                   ecuted.

                   You may use any arithmetic operator in REAL expressions,
                   and REAL variables may take part in relational expressions.

                   The *FLOAT* standard function will convert other numeric data
                   types into REAL.

                   It should be noted that the REAL type is not an ordinal type
                   and cannot be used to index arrays, define subranges or sets,

control **FOR** or **CASE** statements, or take arguments in any procedure expecting an ordinal type.

Turbo Modula-2 uses the IEEE 4-byte real-number format; that is, in the order of their significance, 1 sign bit, 8 exponent bits, and 23 mantissa bits.

```
31   30                    23   22                      0
-----------------------------------------------------------
| s |        exponent       |         mantissa          |
-----------------------------------------------------------
```

The mantissa is in binary format but offset by 1.0, representing only those digits to the right of the decimal point. Thus,

```
000000H  = 1.0
4000C0H  = 1.5
7FFFFFH  = 1.9999999
```

The exponent is in binary format but with a 80H offset. For example, an exponent 84H indicates the mantissa is to be multiplied by

$2 \wedge (84H - 80H) = 2 \wedge 4 = 16$

If the exponent is zero, the floating-point value is considered zero.

The least-significant bytes are stored first, at the lower addresses.

Sample Hex **real numbers**:

```
3F800000H  = 1.0
40000000H  = 2.0
40490FDBH  = 3.141592
```

See the example in *WORD* for printing out other values.

Example   **VAR**
                 SquareArea, SquareSide: REAL;
                 BEGIN
                 SquareSide:= 4.21;
                 SquareArea:= SquareSide * SquareSide;

REALOVERFLOW exception (SYSTEM)   L/E

Description   REALOVERFLOW is raised by the module SYSTEM when a
                 real expression results in a value that is too large or too small
                 to be represented by the result type.

Declaration   **EXCEPTION**
                 REALOVERFLOW;

Usage   **EXCEPTION**
                 REALOVERFLOW:
                   WRITELN('Overflow.');
                 END

                 REALOVERFLOW must be imported from the module
                 SYSTEM.

Comments   This exception occurs only during evaluation of real expres-
                 sions. Integer and cardinal expressions generate the exception
                 OVERFLOW.

See Also   SYSTEM
                 BADOVERLAY
                 OUTOFMEMORY
                 OVERFLOW

Example

```
MODULE RealOverflow;
FROM SYSTEM IMPORT REALOVERFLOW;
VAR
 r: REAL;
BEGIN
 r := MAX(REAL) * 2.2;
EXCEPTION
 REALOVERFLOW:
  WRITELN('Real Overflow');
END RealOverflow.
```

(

RealToStr procedure (Convert)   L

Description   *RealToStr* writes a real number into a string.

Declaration   **PROCEDURE** RealToStr(r: REAL; **VAR** s: **ARRAY OF** CHAR; digits: INTEGER);

Usage   RealToStr(r, s, digits);

RealToStr must be imported from the library module *Convert*.

r   must be of type *Real*.

s   must be of type **ARRAY OF CHAR.**

digits must be of type CARDINAL.

Con(ts   The number *r* will be converted to the string *s*. This procedure is similar in form and action to *WriteReal* in *Texts*. Here, the declared length of *s* is substituted for the *width* parameter in *WriteReal*.

The *digits* parameter indicates the number of digits to the right of the decimal point. The sign of digits determines whether or not scientific notation will be used, with plus for normal notation and minus for scientific.

If *digits* is **zero,** the decimal point will be dropped.

If *s* is too small to contain the string representing *r*, the exception TooLarge will be raised.

See Also   CardToStr
Convert
Doubles
IntToStr
LongToStr
StrToReal

Example   Convert a real to a string:

```
VAR sstr : ARRAY [0..4] OF CHAR;
 lstr : ARRAY [0..9] OF CHAR;
 r :REAL
BEGIN
 r := 3.14159;
 RealToStr( r, sstr, 3 );
 (* sstr now has the value ' 3.141' *)

 RealToStr( r, sstr, 1 );
 (* sstr now has the value ' 3.1' *)

 RealToStr( r, sstr, 0 );
 (* sstr now has the value ' 3' *)

 RealToStr( r, sstr, -3 );
 (* causes the exception TooLarge to be raised *)

 RealToStr( r, lstr, -3 );
 (* lstr now has the value '3.141E+00' *)
```

RECORD type   R

Description	A record consists of a fixed number of bytes that contain *fields* of possibly different types. Each field has a name, the *field identifier*, which is used to select the field. A record may have variant parts that allow the record to have more than one field occupying the same memory.

Usage

```
RecordType = RECORD
                 Field sequence
             END

Field sequence = Field { ; Field }
```

A *field* can be either a *fixed field* or a *variant field*.

```
Fixed field = Field identifier: Field type

Variant field = CASE [Tag identifier]: Tag type OF
                    [Case  label:  Field  sequence]  {|
                    [Case label: Field sequence] }

                    [ELSE Field sequence]
                END
```

The *field type* can be any type.

The *tag type* can be of type INTEGER, CARDINAL, BOOLEAN, CHAR, or enumeration.

Each *case label* can be either a single value or a range of values of the tag variable separated by commas.

Comments   You can use a record to handle several related items of different types; for example, a record of the date could be

```
Date : RECORD
        Day: [1..31];
        Month: (Jan,Feb,Mar,Apr,May,June,
                July,Aug,Sept,Oct,Nov,Dec);
        Year: [1900..1999]
      END
```

Each field in a record has an identifier that is used to access the field. Thus, in the example, a field is specified as *Date.Day*, *Date.Month*, and so on.

A field can take values defined by the field type. In the previous example, the field *Month* can take any value from the enumeration *Jan* to *Dec*.

For variant records, the tag variable is used to select the different cases depending on its value. For example:

```
TYPE
 Lights = (Red,Orange,Yellow,Green,Blue);
VAR
 Example: RECORD
            Name: ARRAY [1..20] OF CHAR;
            CASE Color: Lights OF
             Red..Yellow: Test : CARDINAL |
             Green, Blue: Today: Date |
            END
          END
```

In this example, the value of *Color* selects the form of the record. Thus, if *Select* is a record of type *Example*, altering the value of the tag *Select.Color* changes the record's structure. If the tag has a value of *Red*, *Orange*, or *Yellow* and the record is accessed, the field *Select.Test* exists and can be given a value.

No operators are directly applicable to records (except assignment), but any operator applicable to a field type can operate on that field. Thus, if a record *Value* has a field *RealNumbers* of type REAL, then the following is valid:

```
Value.RealNumbers:= 2.31 * Value.RealNumbers
```

The assignment of values to record fields can either be explicitly stated, specifying both the record identifier and the field:

```
Value.CardNumbers:= 34
```

or can use a **WITH** statement:

```
WITH Value DO
 CardNumbers:= 34;
 RealNumbers:= 2.31 * RealNumbers
END
```

The memory requirements for a record with fixed fields is the sum of the individual field lengths, but the fields that have 1 byte types are allocated in 2 bytes. For example, a record with two character fields actually occupies 4 bytes. A variant field requires enough memory for the largest of the cases.

Example

```
MODULE Records;
TYPE
 Origin = (Citizen,Alien);
 Name   = ARRAY [1..20] OF CHAR;
 Date   =RECORD
          Day: [1..31];
          Month: (Jan,Feb,Mar,Apr,May,June,
                 July,Aug,Sept,Oct,Nov,Dec);
          Year: [1900..1999]
         END ;
 Person = RECORD
          PersonName: Name;
          BirthDate: Date;
          CASE CitizenShip: Origin OF
            Citizen: BirthPlaceName: Name
            |
            Alien:   CountryOfOrigin: Name;
                     DateOfEntry: Date
          END
         END ;
```

```
VAR
 Passenger: Person;
BEGIN
 WITH Passenger DO
  WITH BirthDate DO
   Day:= 15;
   Month:= Sept;
   Year:= 1958
  END ;
  CASE CitizenShip OF
   | Citizen: BirthPlaceName:= 'New Orleans'
   | Alien: CountryOfOrigin:= 'UK';
     WITH DateOfEntry DO
     Day:= 22;
     Month:= Aug;
     Year:= 1984
  END
  END
 END ;
 WRITELN(Passenger.BirthDate.Year);
END Records.
```

RedirectInput procedure (ComLine)   L

Description	*RedirectInput* allows a filename defined in the command line to be designated as standard output.
Declaration	**PROCEDURE** RedirectInput;
Usage	RedirectInput;
	RedirectInput must be imported from the library module *ComLine*.

Comments   The file name preceded by a » < « in the command line will be
                  reassigned to the standard stream *input*. No space is allowed
                  between the » < « and the file name.

                  If no standard input file is specified in the command line, then
                  the standard input is defined to be at the terminal console.

                  The name of the file specified to be the standard input stream
                  is retrievable in the variable *inName* exported by *ComLine*.

See Also   Comline
                  RedirectOutput

Example   Redirect the input:

```
MODULE RedirectTheInput;
FROM ComLine IMPORT RedirectInput,RedirectOutput;
BEGIN
 RedirectInput;
 (* Now input may be redirected from the
    command line *)
END RedirectTheInput.
```

RedirectOutput procedure (ComLine)   L

Description   *RedirectOutput* allows a filename defined in the command line
                  to be designated as standard output.

Declaration   **PROCEDURE** RedirectOutput;

Usage   RedirectOutput;

                  RedirectOutput must be imported from the library module
                  ComLine.

Comments	The file name preceded by a » > « in the command line will be reassigned to the standard stream *output*.

If no standard output file is specified in the command line, then output is defined to be the terminal screen.

The name of the file specified to be the standard output stream is retrievable in the variable *outName* exported by *ComLine*.

See Also	Comline RedirectInput
Example	Redirect the output:

```
MODULE RedirectTheOutput;
FROM ComLine IMPORT RedirectInput,RedirectOutput;
BEGIN
 RedirectOutput;
 (* Now output can be redirected from the
    command line *)
```

RELEASE procedure (STORAGE)   L

Description	*RELEASE* releases a block of allocated memory from the heap.
Declaration	**PROCEDURE RELEASE(VAR** a: ADDRESS);
Usage	RELEASE(a);

RELEASE must be imported from the pseudomodule *STORAGE*.

a must be compatible with type *ADDRESS* (any pointer).

Comments	All of the memory from address *a* to the top of the heap will be released; *a* will become the new top of the heap.
	You can only release a block of memory that has **been** previously specified with *MARK*.
	After the call to *RELEASE*, the value of *a* will be set to *NIL*, and any variable allocated on the heap prior to the *MARK* call will again be accessible.
	RELEASE should be used with care in conjuction with *DISPOSE*, *DEALLOCATE*, and file handling.
See Also	MARK
	NEW
	STORAGE
Example	Allocate a block of variables and then throw them all away:

```
MARK(blockStart);
NEW(blockVar1);
NEW(blockVar2);
NEW(blockVar3);

(* Now use these variables, and when done: *)

RELEASE(blockStart);
```

Rename procedure (Files)   L

Description	*Rename* **changes the name of an open file and then closes it.**
Declaration	**PROCEDURE** Rename(**VAR** f: FILE; name: **ARRAY OF** CHAR);

Usage   Rename(f, newname);

Rename must be imported **from the library module** *Files*.

f must be a variable of type *FILE*, imported from the module *Files*.

newname must be a character array.

Comments   The file *f* must have already been opened under its old name.

newname must be a valid CP/M file name.

This function has the side effect of closing the renamed file. If you want to use it again, you must reopen it with the new name.

See Also   Close
Delete
Files
Open

Example
```
MODULE renameExample;
FROM Files IMPORT FILE, Open, Rename, Close
VAR
 f : FILE;
 newname : ARRAY [0..12] OF CHAR;
 oldname : ARRAY [0..12] OF CHAR;
BEGIN
 oldname := 'oldtest.txt';
 newname := 'newtest.txt';
 IF Open(f, oldname) THEN
  Rename(f, newname)
 END ;
END renameExample.
```

REPEAT statement   R

Description   **REPEAT** repeatedly executes a sequence of statements until a
                     Boolean expression returns a TRUE result.

Usage   **REPEAT**
                     Statement sequence
                     UNTIL BooleanExpression

                     Statement sequence = Statement {; **Statement**}

Comments   The Boolean expression is evaluated only after the statement
                     sequence has been executed at least once.

                     The repetition is terminated when the expression gives a
                     TRUE result.

Example   **MODULE** Repeats;
                     VAR
                      Answer: CHAR;
                     BEGIN
                      REPEAT
                       WRITE('Enter "Q" and <RET> to continue . . . ');
                       READ(Answer);
                      UNTIL CAP(Answer) = 'Q';
                     END Repeats.

ResetSys procedure (Files)   L

Description   *ResetSys* resets **the disk drive system.**

Declaration   **PROCEDURE** ResetSys( );

Usage   ResetSys;

                     ResetSys must be imported from the library module *Files*.

Comments	*ResetSys* asks CP/M to update the file directory. Thus, after a call to *ResetSys*, it is possible to write to drives where disks have been swapped, but all open output files will then be lost. Consequently, make sure all your disk files are closed before using this procedure.
See Also	Files
Example	Get the user to change disks:

```
WRITELN('Change disk and type return when ready');
READLN(ch);
ResetSys;
```

RETURN statement   R

Description	**RETURN** terminates execution and specifies the return value of a function procedure.
Usage	**RETURN ;**
	or
	RETURN answer;
	answer must be compatible with the type with which the function procedure has been declared.
Comments	Normally, there will be only one **RETURN** in a function procedure, just before the final **END**. However, it is allowable to put as many **RETURN**s as you like throughout the procedure. When a **RETURN** is encountered, the procedure will terminate, and the value specified at that point will be returned to the calling procedure.

Example

```
PROCEDURE maxi3(i1, i2, i3 :INTEGER) :INTEGER
BEGIN
 IF ((i1>i2) & (i1>i3)) THEN
   RETURN i1
 ELSIF (i2>i3) THEN
   RETURN i2
 ELSE
   RETURN i3
 END
END maxi3;
```

SEND procedure (Processes)   L

Description   *SEND* transfers **control to any process waiting for** a given signal.

Declaration   **PROCEDURE SEND(VAR** s: SIGNAL);

Usage   SEND(s);

                     SEND must be imported from the library module *Processes*.

                     s must be of type *SIGNAL* imported from *Processes*.

Comments   *SEND* is used in conjunction with the other routines in the module *Processes*: *Init*, *StartProcesses*, *Wait*, and *Awaited*.

                     If no process is waiting for signal *s*, then *SEND* has no effect.

                     If a process is waiting for *s* (that is, has terminated execution with the routine *WAIT*), then control will be transferred to it.

                     If several processes are waiting, then the first one in the queue will get control.

                     Before a signal is used it must be initialized using the procedure *Init* in *Processes*.

                     To find out if processes are **waiting for a signal, use the routine** *Awaited* in *Processes*.

See Also   Awaited
Init
Processes
StartProcess
WAIT

Example   Send the signal done:

```
SEND(done);
```

SET type   R

Description   A set-type variable can assume as a value a set of up to 16 different elements with ordinal values in the range of 0 to 15.

Usage   Set Type = **SET OF** Base Type;

Set Assignment := Base Type ø{ø **Base Type Elements** ø}ø ;

Base type can be enumeration or subrange.

Base-type elements are elements of the base type.

Comments   A *set* is a collection of *elements* that you handle as a whole.

Besides assigning a value to a set, you can also include another element from the base type into the set, or exclude (or remove) an element from the set using the standard procedures *INCL* and *EXCL*.

The operators applicable to set expressions are the set operators and the relational operator **IN**. These operators allow you to check if a particular element is present in a set, to find the difference between two sets, and so forth.

There is a predefined set type called BITSET that is defined as *BITSET = SET OF [0..15]*. With this set type, you can define sets with values made up of combinations of CARDINAL numbers in the range 0 to 15. A set type value requires 2 bytes for storage in memory.

See Also   INCL
                      EXCL   .

Example   Suppose you are an automobile manufacturer and wish to record the state of completion of each vehicle. You could define a set of the main parts required, like so:

```
TYPE
 Components = (Body,Engine,Chassis);
Compset = SET OF Components
VAR
 State: Compset
```

The enumeration *Components* is called the base type of the set *State*. The set *State* can take as a value any combination of elements from the base type. For example, all of the following are possible assignments:

```
State:= Compset{Body,Chassis};
(* State has two elements Body and Chassis *)

State:= Compset{Body..Chassis};
(* State has all three elements *)

State:= Compset{ };
(* State has no elements *)

MODULE Sets;
TYPE
 Components = (Body,Engine,Chassis);
 Compset= SET OF Components;
VAR
 State: Compset;
BEGIN
 State := Compset{Body,Engine};
END Sets.
```

SetCol procedure (Texts)   L

Description	*SetCol* advances the write position of a *TEXT* file to a given column.
Declaration	**PROCEDURE** SetCol(t: TEXT; column: CARDINAL);
Usage	SetCol(t, column);
	SetCol must be imported from the library module *Texts*.
	t must be of type *TEXT* imported from *Texts*.
	column must be of type CARDINAL.
Comments	*SetCol* has no effect on input. The current column can be queried by use of the function *Col* exported from *Texts*.

See Also   Col
                  Texts

Example   Cause the string »Hello world !« to be output in the twentiet
                  column:

                  ```
                  SetCol(output, 20);
                  WRITE("Hello world !");
                  ```

                  Example of tabbing to next multiple of 10:

                  ```
                  SetCol(output, (Col(output) DIV 10 + 1) * 10)
                  ```

SetPos procedure (Files)   L

Description   Sets the current byte position of a file.

Declaration   **PROCEDURE** SetPos(f: FILE; pos: LONGINT);

Usage   ```
                  SetPos(f, bytepos);
                  ```

                  SetPos must be imported from the library module *Files*.

                  f must be of type *FILE* imported from *Files*.

                  bytepos must be of type LONGINT.

Comments   The procedure is used for random access of data. If all the fil
                  elements of a certain file are of type *t*, then the file can be posi
                  tioned at the *n*th element by the call

                  ```
                  SetPos(f, LONG(n)*LONG(SIZE(t)) )
                  ```

See Also   Files
                  NextPos

Example   Read the fifteenth widget from a file that has a header:

```
MODULE RandomAccess;
TYPE
 widget  = RECORD .... END ;
 header  = RECORD .... END ;
VAR
 currentWidget : widget;
 pos : LONGINT;
BEGIN
 pos := 15L*LONG(SIZE(widget)) + LONG(SIZE(header));
 SetPos(widgetFile, pos);
 ReadRec(widgetFile, currentWidget);
END RandomAccess.
```

SIGNAL type (Processes)   L

Description   *SIGNAL* references control loosely coupled processes or coroutines. A *SIGNAL* may be *Sent* or *Awaited*.

Declaration
```
TYPE
 SIGNAL;
```

Usage
```
VAR
 s: SIGNAL;
```

SIGNAL must be imported from the module *Processes*.

Comments   It is mandatory that a *SIGNAL* variable be initialized by a call to the procedure *Init* in *Processes*.

See Also   Processes
SYSTEM

Example   **MODULE** Signals;
                FROM Processes **IMPORT** Init, SIGNAL;
                VAR s: SIGNAL;
                BEGIN
                 Init(s);
                END Signals.

Sin function (MathLib, LongMath)   L

Description   *Sin* returns the sine of X, where X is expressed in radians.

Declaration   **PROCEDURE** Sin(x: REAL): REAL;
                PROCEDURE Sin(x: LONGREAL): LONGREAL;

Usage   Y:= Sin(X);

                Argument X **and result** Y **are either both REAL or bot**
                LONGREAL.

Comments   If you want the argument **expressed in degrees instead of** ra
                dians, use the expression

                Result:= Sin(X * 3.141592/180.0)

                If you need to calculate the inverse of sine, **you can use th**
                ArcTan function in the following expression:

                Arcsin:= Arctan(X/Sqrt(1.0 - X*X).

See Also   Cos
                MathLib

Example   **MODULE** Sine;
                 FROM MathLib **IMPORT** Sin;
                 CONST
                  DegsToRads = 3.141592/180.0;
                 VAR
                  Degs, Radians, sinAngle: REAL;
                 BEGIN
                  Radians:= Degs * DegsToRads;
                  sinAngle:= Sin(Radians);
                 END Sine.

SIZE standard procedure   S

Description   *SIZE* returns the minimum required storage in bytes of a type
                 or variable.

Usage   SIZE(x);

                 x can be a variable of **any type or a type** identifier.

Comments   The *SIZE* of a CHAR, BOOLEAN, or BYTE variable is 1;
                 WORD, INTEGER, and CARDINAL variables are 2; REAL
                 and LONGINT variables are 4; LONGREAL variables are 8.

                 Note that byte-size variables are only packed in arrays and thus
                 always take up one word of storage, unless they are in an array.

See Also   TSIZE

Example   Multiply 2 x 4:

```
MODULE Sizes;
TYPE
 rec = RECORD
         a,b: CHAR;
         END ;
 aray= ARRAY [0..1] OF CHAR;
VAR
 r: rec;
 a: aray;
BEGIN
 WRITELN(SIZE(r), SIZE(a));
END Sizes.
```

The output is

4   2

which shows that arrays are packed and records are not.

SpecialOps type (Terminal)   L

Description   *SpecialOps* is an enumeration of terminal operations that are
                  not available on all terminals. It is used to describe a set of
                  operations available on the currently installed terminal.

Declaration   **TYPE**
                  SpecialOps = (clearEol, insertDelete, highlightnor-
                  mal);
                  OpSet : SET OF SpecialOps;

Usage   **VAR**
                  available : OpSet;

                  available is a predefined variable declared as type *OpSet*; it
                  must be imported from the module *Terminal*.

Comments   This variable is useful for writing terminal-independent pro-
           grams. The value of *available* is set by the installation program
           INSTM2.

See Also   available
           OpSet
           Terminal

Example   **MODULE** TermStuff;
           FROM Terminal **IMPORT**
            available, SpecialOps, ClearToEOL, **GotoXY**, numCols;
           VAR I, CurCol, CurRow: CARDINAL;
           BEGIN
            IF clearEol **IN** available **THEN** **ClearToEOL**
            ELSE
             FOR I := CurCol **TO** numCols **DO** WRITE(' ') **END** ;
             GotoXY(CurCol,CurRow)
            END ;
            WRITE('A line of text.');
           END TermStuff.

Sqrt function (MathLib, LongMath)   L

Description   *Sqrt* returns the square root of positive *X*.

Declaration   **PROCEDURE** Sqrt(x: REAL): REAL;
              PROCEDURE Sqrt(x: LONGREAL): LONGREAL;

Usage   Y:= Sqrt(X);

              Argument *X* **and result** *Y* are either both REAL or both
              LONGREAL.

              The argument **must be positive or zero.**

Comments   The argument *X* must be positive or zero; otherwise, the ex-
              ception ArgumentError is raised.

See Also   MathLib
                  LongMath

Example   **MODULE** SquareRoot;
                  FROM MathLib **IMPORT** Sqrt;
                  CONST
                   Pi = 3.141592;
                  VAR
                   CircleArea, CircleRadius: REAL;
                  BEGIN
                   READ (CircleArea);
                   IF CircleArea $>$ = 0.0 **THEN**
                    CircleRadius:= Sqrt(CircleArea/Pi)
                   END
                  END SquareRoot.

StartProcess procedure (Processes)   L

Description   *StartProcess* allocates space to a process and transfers control
                  to it.

Declaration   **PROCEDURE** StartProcess(P: PROC; n: CARDINAL);

Usage   StartProcess(proc, workSpaceSize);

                  StartProcess must be imported from the library module *Pro-
                  cesses.*

                  proc must be of type *PROC*.

                  workSpaceSize must be of type **CARDINAL.**

Comments   *proc* is a procedure without **RETURNs**. Normally, its
                  statements will be enclosed in a **LOOP** statement without **EX-
                  ITs**. Control passes to and from the process by calls to *SEND*
                  and *WAIT.*

                  This routine is implemented using the procedures
                  NEWPROCESS and *TRANSFER* of the pseudomodule
                  SYSTEM.

See Also   Awaited
                  Init
                  Processes
                  SEND
                  WAIT

Example   Start the process *service1*:

                  ```
                  StartProcess(service1,200);
                  ```

StatusError exception (Files)   L/E

Description   StatusError is raised by the *Files* module when a *FILE* variable
                  is used incorrectly; for example, when reading a unopened file
                  or opening a file twice.

Declaration   **EXCEPTION**
                  ```
                   StatusError;
                  ```

Usage   **EXCEPTION**
                  ```
                   StatusError: WRITELN('File error');
                  END
                  ```

                  Status**Error must be imported** from the module *Files*.

Comments   Use the StatusError exception to prevent your program from
                  crashing due to user input.

See Also	EndError
	UseError
	DeviceError
	DiskFull

Example

```
MODULE ReadFile;
FROM Files IMPORT
 FILE, Open, ReadWord, Close, EndError, StatusError;
VAR f: FILE; c: CARDINAL;
BEGIN
 IF Open(f,'testfile.dat') THEN
  ReadWord(f,c);
 END ;
 ReadWord(f,c);
 Close(f)
EXCEPTION
 EndError: WRITELN('End of file reached.'); Close(f)
 | StatusError: WRITELN('Error file not opened');
END ReadFile.
```

STORAGE pseudomodule   L

Description   *STORAGE* is a pseudomodule for managing heap memory in general and dynamic variables in particular.

Declaration
```
DEFINITION MODULE STORAGE;
FROM SYSTEM IMPORT ADDRESS;

PROCEDURE ALLOCATE(VAR A:ADDRESS; Size: CARDINAL);
PROCEDURE DEALLOCATE(VAR
A:ADDRESS; Size: CARDINAL);

PROCEDURE MARK(VAR A: ADDRESS);
PROCEDURE RELEASE(VAR A: ADDRESS);

PROCEDURE FREEMEM( ):CARDINAL

END STORAGE.
```

Comments	The procedures *NEW* and *DISPOSE* can be considered abbreviated forms of *ALLOCATE* and *DEALLOCATE*. To use the standard procedures *NEW* and *DISPOSE*, *ALLOCATE* and *DEALLOCATE*, respectively, must be imported.
See Also	MARK NEW RELEASE

(:ing type (Strings)   L

Description	*String* is a predefined array of characters.
Declaration	**TYPE** String = **ARRAY** [0..80] **OF** CHAR;
Usage	**VAR** s1, s2: String; *String* must be imported from the library module *Strings*.
Comments	Since *String* is a character array, a variable of this type may be assigned and compared with any other character array variable. Even though the module *Strings* exports the type *String* none of the procedures require this type. Instead, they use open array parameters that will accept any array of characters.
See Also (	Strings
Example	*vl* and *v2* are defined as strings: **FROM** Strings **IMPORT** String; **VAR** v1,v2: String;

StringError exception (Strings)   L/E

Description	StringError is raised by the module *Strings* when the destination of a string operation is not large enough to hold the result.
Declaration	**EXCEPTION** · StringError;
Usage	**EXCEPTION** StringError WRITELN('String error'); **END** **StringError must be imported from the module *Strings*.**
Comments	This exception occurs only during calls to the procedures exported by *Strings*.
See Also	Strings
Example	**MODULE** StringTooShort; **FROM** Strings **IMPORT** StringError, Append; **VAR** s1, s2 : **ARRAY** [0..12] **OF** CHAR; **BEGIN** s1 := 'Hello there'; s2 := 'everyone'; Append(s2,s1); **EXCEPTION** StringError: WRITELN('Destination string too short.'); **END** StringTooShort.

Strings module   L

Description   *Strings* is a module for the manipulation of strings.

Declaration   **DEFINITION MODULE** Strings;
                   TYPE String = **ARRAY** [0..80] **OF** CHAR;
                   PROCEDURE Length   (**VAR** str: **ARRAY OF** CHAR):
                                          CARDINAL;
                   PROCEDURE Pos   (substr,str: **ARRAY OF** CHAR):
                                          CARDINAL;
                   PROCEDURE Insert   (substr: **ARRAY OF**
                                          CHAR; **VAR** str: **ARRAY OF** CHAR;
                                           inx:CARDINAL);
                   PROCEDURE Delete   (**VAR** str: **ARRAY OF** CHAR; inx,len:
                                          CARDINAL);
                   PROCEDURE Append   (substr: **ARRAY OF**
                                          CHAR; **VAR** str: **ARRAY OF** CHAR);
                   PROCEDURE Copy   (**VAR** str: **ARRAY OF** CHAR; inx,len:
                                           CARDINAL;
                                           VAR result: **ARRAY OF** CHAR);
                   PROCEDURE CAPS   (**VAR** str: **ARRAY OF** CHAR);
                   EXCEPTION StringError;
                   END Strings.

Comments   These procedures exported from *Strings* work on any array of
                   characters, not just on variables declared as type *String*.

                   These procedures use open arrays; that is, arrays of any size.
                   You should, however, restrict yourself to arrays with indexing
                   that start at zero, since the various *inx* parameters listed earlier
                   work relative to zero.

See Also   String

StrToCard procedure (Convert)   L

Description	*StrToCard* performs conversion from a string variable to a cardinal variable.
Declaration	**PROCEDURE** StrToCard(**VAR** s: **ARRAY OF** CHAR; **VAR** CARDINAL): BOOLEAN;
Usage	okay := StrToCard(s, c);

StrToCard must be imported from the library module *Convert*.

okay must be of type BOOLEAN.

s must be a character array.

c must be of type CARDINAL.

Comments	*okay* is assigned TRUE if the string represents a valid cardinal number; otherwise, it is assigned FALSE.
See Also	CardToStr Strings
Example	Read a cardinal *c* out of a string *str*:

```
str := ' 12323';
IF NOT StrToCard(str, c) THEN
 WRITELN(str,'is not convertible.');
END ;
```

StrToDouble procedure (Doubles)   L

Description	*StrToDouble* performs conversion from a string variable to a double-precision real variable.
Declaration	**PROCEDURE** StrToDouble(**VAR** s: **ARRAY OF** CHAR; **VAR** d: LONGREAL): BOOLEAN;
Usage	okay := StrToDouble(s, d);
	StrToDouble must be imported from the library module *Doubles*.
	okay must be of type **BOOLEAN**.
	s must be a character array.
	d must be of type LONGREAL.
Comments	*okay* is assigned TRUE if the string represents a valid real number; otherwise, it is assigned FALSE.
See Also	Doubles DoubleToStr
Example	Read a double-precision *d* out of a string *str*:

```
str := ' 2.99D10';
IF NOT StrToDouble(str, d) THEN
 Writeln (str,'is not convertible');
END ;
```

StrToInt procedure (Convert)   L

Description	*StrToInt* performs conversion from a string variable to an integer variable.
Declaration	**PROCEDURE** StrToInt **(VAR** s: **ARRAY OF** CHAR; **VAR** i: INTEGER): BOOLEAN;
Usage	okay := StrToInt(s, i);
	StrToInt must be imported from the library module *Conver*
	okay must be of type BOOLEAN.
	s must be a character array.
	i must be of type INTEGER.
Comments	*okay* is assigned TRUE if the string represents a valid integer otherwise, it is assigned FALSE.
See Also	IntToStr Strings
Example	Read an integer called *i* out of a string:

```
str := " 30000 - This number is almost too big ";
okay := StrToInt(str, i);

(* i has got 30000 now *)
```

StrToLong procedure (Convert)   L

Description	*StrToLong* performs conversion from a string **variable to a** long integer variable.
Declaration	**PROCEDURE** StrToLong(**VAR** s: **ARRAY OF CHAR;** **VAR** l: LONGINT): BOOLEAN;
Usage	okay := StrToLong(s, l);

StrToLong must be imported **from the library module** *Convert*.

okay must be of type BOOLEAN.

s must be a character array.

l must be of type LONGINT.

Comments	*okay* is assigned TRUE if the string represents a **valid** long integer; otherwise, it is assigned FALSE.
See Also	LongToStr Strings
Example	Read a longer integer (see the example in *StrToInt*) out of a string:

```
VAR
 str: ARRAY [0..20] OF CHAR;
 l: LONGINT;
BEGIN
 str := "40000";
 okay := StrToLong(str, l);
END
```

StrToReal procedure (Convert)   L

Description	*StrToReal* performs **conversion from a string variable to a real-**number variable.
Declaration	**PROCEDURE** StrToReal(**VAR** s: **ARRAY OF** CHAR; **VAR** r: REAL): BOOLEAN;
Usage	okay := StrToReal(s, r); *StrToReal* must be imported **from the library module** *Convert*. *okay* must be of type BOOLEAN. *s* must be a character array. *r* must be of type REAL.
Comments	*okay* is assigned TRUE if the string **represents a** valid **real;** otherwise, it is assigned FALSE.
See Also	RealToStr Strings
Example	Make Pi out of a string: stringyPi := "3.14159265"; okay := StrToReal(stringyPi, Pi); (* Pi is now Pi *)

SYSTEM pseudomodule   L

Description	*SYSTEM* is a pseudomodule that provides low-level support.

Declaration **DEFINITION MODULE** SYSTEM

TYPE
  WORD; **BYTE; ADDRESS;** PROCESS;
  VAR
    IORESULT, HLRESULT: CARDINAL;

PROCEDURE ADR**(VAR** v: AnyType): ADDRESS;
PROCEDURE TSIZE(AnyType): CARDINAL;

PROCEDURE TRANSFER**(VAR** source, dest: PROCESS);
PROCEDURE IOTRANSFER**(VAR** source, dest: PROCESS;
    n: CARDINAL);
PROCEDURE NEWPROCESS(p: PROC; a: ADDRESS;
    n: CARDINAL; VAR q: PROCESS);

PROCEDURE BIOS(n:CARDINAL w: WORD): CARDINAL;
PROCEDURE **BDOS**(n:CARDINAL w: WORD): CARDINAL;
PROCEDURE CODE(AnyStringLiteral);

PROCEDURE MOVE(source, destin: ADDRESS;
    nbytes: CARDINAL);
PROCEDURE FILL(start: ADDRESS; nbytes: **CARDINAL;**
    ch: BYTE);
PROCEDURE INP(port: WORD): CARDINAL;
PROCEDURE OUT(port: WORD; outByte: WORD);

EXCEPTION OVERFLOW, READOVERLOW, OUTOFMEMORY,
    BADOVERLAY;

END SYSTEM.

Comments   For explanations of the various procedures, see the specific en-
             tries. In general, use of these procedures make a program
             much less portable.

termCH variable (InOut)   L

Description	*termCH* always contains the last character read by the module *InOut*.
Declaration	**VAR** termCH: CHAR;
Usage	LastChar := termCH; *termCH* must be imported from the module *InOut*.
Comments	*termCH* contains the last character read by any of the five read procedures in *InOut*.
See Also	InOut Done Texts

Example	**MODULE** LastChar;
	FROM InOut **IMPORT** ReadCard, termCH;
	VAR ch: CHAR; c: CARDINAL;
	BEGIN
	  ReadCard(c);
	  WRITELN('The cardinal read is ',c);
	  WRITELN('The last character of the cardinal is
	  ',termCH);
	END LastChar.

Terminal module   L

Description	*Terminal* provides input/output to the terminal.

Declaration   **DEFINITION MODULE** Terminal;

```
(* Terminal Input *)

PROCEDURE ReadChar(VAR ch: CHAR);
PROCEDURE BusyRead(VAR ch: CHAR);
PROCEDURE ReadAgain;
PROCEDURE ReadLine(VAR s: ARRAY OF CHAR);

(* Screen output *)

VAR numRows, numCols:CARDINAL

PROCEDURE WriteChar(ch: CHAR);
PROCEDURE WriteLn;
PROCEDURE WriteString(s: ARRAY OF CHAR);
PROCEDURE ClearScreen;
PROCEDURE GotoXY(col,row:CARDINAL);

PROCEDURE InitScreen;
PROCEDURE ExitScreen;

TYPE
 SpecialOps = (clearEol, insertDelete,
 highlightNormal);
 OpSet   = SET OF SpecialOps;

VAR
 available: OpSet;

PROCEDURE ClearToEOL;
PROCEDURE InsertLine;
PROCEDURE DeleteLine;
PROCEDURE Highlight;
PROCEDURE Normal;

END Terminal.
```

Comments   This module, or at least the screen manipulation procedures,
                will work correctly only if Turbo Modula-2 has been properly
                installed on your terminal.

TEXT data structure (Texts)   L

Description   *TEXT* is a type that represents legible disk files.

Declaration   **TYPE** TEXT = [1..16];

Usage   **VAR** t: TEXT;

                TEXT must be **imported from the library module *Texts*.**

Comments   A *TEXT* is a character stream. For example, a WordStar docu-
                ment file is better handled as a *FILE*, while a nondocument file
                can be manipulated as a *TEXT.*

                There are a variety of procedures to handle *TEXT* objects.

See Also   Texts

Example   Simulate the CP/M TYPE utility:

```
MODULE type;
FROM ComLine IMPORT PromptFor;
FROM Texts    IMPORT TEXT, ReadString, OpenText,
                     EOT, ReadChar, WriteChar,
                     output;
VAR
 textName: ARRAY [0..20] OF CHAR;
 t        : TEXT;
 c        : CHAR;
 okay     :BOOLEAN;
BEGIN
 PromptFor("File to type: ",textName);
 IF OpenText(t, textName) THEN
  WHILE NOT EOT(t) DO
   ReadChar(t, c);
   WriteChar(output, c);
  END
 ELSE
  WRITELN(textName,' not found');
 END
END type.
```

TextDriver type (Texts)   L

Description   *TextDriver* must be used when installing your own character device drivers. A procedure of this type may be passed to *ConnectDriver*.

Declaration   **TYPE** TextDriver = **PROCEDURE** (TEXT,**VAR** CHAR);

Usage   **VAR**
     MyDriver: TextDriver;

    TextDriver must be imported **from the module** *Texts*.

Comments   You do not need to import the type *TextDriver*, but the procedure you pass to *ConnectDriver* must match it.

See Also   Texts
                    ConnectDriver

Example   **MODULE** CharDriver;
                    FROM Texts **IMPORT** TEXT, ConnectDriver, TextDriver;
                    VAR t:TEXT;

                    PROCEDURE MyDriver(t: TEXT; **VAR** ch: CHAR);
                    BEGIN
                    END MyDriver;

                    BEGIN
                     ConnectDriver(t,MyDriver);
                    END CharDriver.

TextFile procedure (Texts)   L

Description   *TextFile* returns a value of type *FILE*, as described in the
                    module *Files*. Since a text file is implemented as a *Files.FILE*,
                    you may obtain this handle to perform »file« operations on a
                    TEXT.

Declaration   **PROCEDURE** TextFile(T:TEXT): FILE;

Usage   f := TextFile(t);

                    TextFile must be imported from the module *Texts*.

                    t must be of type *TEXT*.

                    f must be of type *FILE*.

Comments   If the file has not been opened, *TextFile* returns *NIL*.

See Also   Texts
                    Files

Example

```
MODULE SizeOfAText;
FROM Texts IMPORT TEXT, TextFile;
FROM Files IMPORT FileSize;
VAR t: TEXT;
BEGIN
 WRITE('The size of the text file is ');
 WRITELN(FileSize(TextFile(t)),' bytes.');
END SizeOfAText.
```

TextNotOpen exception (Texts)   L/E

Description   TextNotOpen is raised by the module *Texts* when an attempt is
                    made to use an unopened text.

Declaration   **EXCEPTION**
                    TextNotOpen;

Usage   **EXCEPTION**
                    TextNotOpen: WRITELN('Text not open');
                    END

                    TextNotOpen must be imported from the module *Texts*.

Comments   This exception can be raised by all input/output procedures
                    and the procedure *SetCol*.

See Also   Texts
                    TooManyTexts

Example   **MODULE** TextNotOpen;
              FROM Texts **IMPORT** TEXT, OpenText, TextNotOpen;
              VAR
               t   : TEXT;
               filename : **ARRAY** [0..15] **OF** CHAR;
               x   : REAL;
              BEGIN
               IF OpenText(t,filename) **THEN END** ;
               READ(t,x);
              EXCEPTION
               TextNotOpen: WRITELN("Can't read an unopen text.");
              END TextNotOpen.

Texts module   L

Description   *Texts* manipulates legible disk files.

Declaration   **DEFINITION MODULE** Texts;
                 FROM Files **IMPORT** FILE;

                 CONST EOL=36C;
                 TYPE TEXT = [1..16];
                 VAR input,output,console: TEXT;

                 PROCEDURE ReadChar   (t: TEXT; **VAR** ch: CHAR);
                 PROCEDURE ReadString (t: TEXT; **VAR** s: **ARRAY OF**
                                           CHAR);
                 PROCEDURE ReadInt   (t: TEXT; **VAR** i:INTEGER);
                 PROCEDURE ReadCard   (t: TEXT; **VAR** c:CARDINAL);
                 PROCEDURE ReadLong   (t: TEXT; **VAR** l:LONGINT);
                 PROCEDURE ReadReal   (t: TEXT; **VAR** r:REAL);
                 PROCEDURE ReadLine   (t: TEXT; **VAR** s: **ARRAY OF**
                                           CHAR);
                 PROCEDURE ReadAgain   (t: TEXT);
                 PROCEDURE ReadLn   (t: TEXT);

                 PROCEDURE WriteChar   (t: TEXT; ch: CHAR);
                 PROCEDURE WriteString(t: TEXT; s: **ARRAY OF CHAR**);

```
PROCEDURE WriteInt    (t: TEXT; i:INTEGER;
                       n:CARDINAL);
PROCEDURE WriteCard   (t: TEXT; c, n:CARDINAL);
PROCEDURE WriteLong   (t: TEXT; l:LONGINT;
                       n:CARDINAL);
PROCEDURE WriteReal   (t: TEXT; r:REAL; n:CARDINAL;
                       digits:INTEGER);
PROCEDURE WriteLn     (t: TEXT);

PROCEDURE Done        (t: TEXT):BOOLEAN;
PROCEDURE EOLN        (t: TEXT):BOOLEAN;
PROCEDURE EOT         (t: TEXT):BOOLEAN;

PROCEDURE OpenText    (VAR t: TEXT; name: ARRAY OF
                       CHAR):BOOLEAN;
PROCEDURE CreateText  (VAR t: TEXT; name: ARRAY OF
                       CHAR);
PROCEDURE CloseText   (VAR t: TEXT);

PROCEDURE Col         (t: TEXT): CARDINAL;
PROCEDURE SetCol      (t: TEXT; column:CARDINAL);

PROCEDURE TextFile    (t: TEXT): FILE;

TYPE TextDriver = PROCEDURE(TEXT, VAR CHAR);

PROCEDURE ConnectDriver(VAR t: TEXT; p: Text
                       Driver);

PROCEDURE Init; (* used only by system *)

VAR haltOnControlC:BOOLEAN; (*TRUE by default *)

EXCEPTION TextNotOpen, TooManyTexts;

END Texts.
```

Comments   Rather than use the various read and write procedures in this
                module directly, the user is encouraged to use the standard
                procedures *READ, READLN, WRITE,* and *WRITELN.* These
                are translated by the compiler into calls to the appropriate pro-
                cedures in *Texts.*

TooLarge exception (Convert)   L/E

Comments   Rather than use the various read and write procedures in this
                module directly, the user is encouraged to use the standard
                procedures *READ, READLN, WRITE,* and *WRITELN.* These
                are translated by the compiler into calls to the appropriate pro-
Description   TooLarge indicates that a conversion from a number to a string
                cannot be done because the string representation of the number
                is too long for the destination string.

Declaration   **EXCEPTION** TooLarge;

Usage   **EXCEPTION**
                 TooLarge:
                  WRITELN('could not convert the number to string');
                END

Comments   This exception eliminates the necessity to check user input
                after every user entry. A common exception handler can be
                written that will trap all such conversion errors.

See Also   Convert

Example   **MODULE** InputTooLarge;
                FROM Convert **IMPORT** IntToStr, TooLarge;
                VAR
                 i: CARDINAL; s: **ARRAY** [0..2] **OF** CHAR;

```
BEGIN
 WRITE('Give me an cardinal less than 100');
 READLN(i);
 CardToStr(i,s);
 WRITELN('The string is ',s);
EXCEPTION
 TooLarge: WRITELN('That was too large!');
END InputTooLarge.
```

oManyTexts exception (Texts)   L/E

Description
TooManyTexts is raised by the module *Texts* when an attempt is made to open a text and there are already 16 open.

Declaration
```
EXCEPTION
 TooManyTexts;
```

Usage
```
EXCEPTION
 TooManyTexts: WRITELN('Too many text files');
END
```

TooManyTexts must be imported from the module *Texts*.

Comments
There are three texts that are always open: *console*, *input*, and *output*. These three texts are part of the 16 allowable texts.

See Also
Texts
TextNotOpen

Example
```
MODULE TooManyTextFiles;
FROM Texts IMPORT TEXT, CreateText, TooManyTexts;
VAR
 t: ARRAY [0..15] OF RECORD
                        t: TEXT;
                        filename: ARRAY [0..15] OF
                      CHAR;
                      END ;
     I: CARDINAL;
```

```
BEGIN
 FOR I := 0 TO 15 DO
  CreateText(t[I].t,t[I].filename);
 END
EXCEPTION
 TooManyTexts: WRITELN("Can't open another text.");
END TooManyTextFiles.
```

TRANSFER procedure (SYSTEM)   L

Description	*TRANSFER* allows cooperative control between processes.

Declaration   **PROCEDURE** TRANSFER(**VAR** source, dest: PROCESS);

Usage   TRANSFER(source, destination);

TRANSFER must be imported from the pseudomodule *SYSTEM*.

source must be of type *PROCESS* imported from *SYSTEM*.

destination must be of type *PROCESS* imported from *SYSTEM*.

Comments   The *source* variable will be assigned to the process where the TRANSFER call is made. This permits the called process to return control to the caller, if desired.

Before you transfer to a process it must have first been set up by a call to *NEWPROCESS*.

We recommend using the module *Processes* where possible, since it provides rudimentary signaling support upon which to base scheduling.

See Also   IOTRANSFER
NEWPROCESS
PROCESS

Example   Mark time:

```
MODULE TickTock;
FROM SYSTEM IMPORT
 NEWPROCESS, TRANSFER, ADDRESS, PROCESS, WORD;
VAR
 tickProcess, tockProcess, main: PROCESS;
 tickWork, tockWork : POINTER TO ARRAY [0..99] OF
 WORD;
PROCEDURE tick;
BEGIN
 WRITELN('tick');
 TRANSFER(tickProcess, tockProcess);
END ;

PROCEDURE tock;
BEGIN
 WRITELN('tock');
 TRANSFER(tockProcess, tickProcess);
END tock;

BEGIN
 NEWPROCESS(tick, tickWork, SIZE(tickWork^),
 tickProcess);
 NEWPROCESS(tock, tockWork, SIZE(tockWork^),
 tockProcess);
 TRANSFER(main, tickProcess); (* Start of
 tick-tock *)
 (* This routine can be hard to stop *)
END TickTock.
```

TRUE standard value   S

Description   **TRUE denotes the Boolean state of truth.**

Usage   Finished := TRUE;

                 Finished is a variable of type **BOOLEAN.**

Comments	The ordinal value of TRUE is 1; thus, *WRITE(CAR-DINAL(TRUE))* will print a 1. In contrast, the ordinal value of FALSE is 0; thus, truth is greater than falsity.
See Also	BOOLEAN FALSE
Example	**MODULE** Truth; **VAR** b: BOOLEAN; **BEGIN** **REPEAT** b := TRUE **UNTIL** b; (* does not repeat *) **END** Truth.

TRUNC standard function   S

Description	*TRUNC* converts from **REAL** type to **CARDINAL** type.
Usage	Y:= TRUNC(X); Argument *X* is of type REAL. The argument must be in the **CARDINAL** (0 to 65535) range. Result *Y* is of type CARDINAL.
Comments	You must make sure that the argument is in the **CARDINAL** range, otherwise an overflow error will occur. The argument is truncated (any fractional part is removed); for example, the number +16.74 will become a CARDINAL +16. Note that truncating a number is not the same as rounding the number to the nearest whole value; for example, 9.99999 will be truncated to 9, not rounded to 10. If you wish to round a REAL value to the nearest whole value, use the expression *Rounded:= TRUNC(X + 0.5)*. The inverse conversion from **CARDINAL** or **INTEGER** to REAL uses the *FLOAT* function.

TRUNC returns a CARDINAL result; therefore, its argument must be nonnegative. The result can be assigned to INTEGERs as well, since INTEGER and CARDINAL are assignment compatible.

Example

```
VAR
 Values: ARRAY 1..10] OF REAL;
 Rounded: ARRAY [1..10] OF CARDINAL;
 I: CARDINAL;
BEGIN
 FOR I:= 1 TO 10 DO
  Rounded[I]:= TRUNC(Values[I] + 0.5)
 END ;
```

TSIZE procedure (SYSTEM)   L

Description   TSIZE returns the storage requirements of a data type in bytes.

Declaration   **PROCEDURE** TSIZE(AnyType): CARDINAL;

Usage   bytes := TSIZE(t);

                     TSIZE must be imported from the **pseudomodule** *SYSTEM*.

                     bytes must be of type CARDINAL.

                     t may be any type identifier.

Comments   This function returns the size of a type; if **you want the size** of a specific variable, use *SIZE*.

See Also   SIZE
                     SYSTEM

Example   realSize := TSIZE(REAL); (* **realSize** = 4 *)

TYPE declaration   R

Description	TYPE signals the start of a data-type definition section in the declaration part of a procedure or module.
Usage	**TYPE** newtype = oldTypeList;
	newtype can be any valid identifier.
	oldTypeList is a combination of previously defined data types.
Comments	This declaration allows you to define objects whether they comprise sets like Red, Green, Blue; subranges of basic types like all integers between 1950 and 2000; or combinations of any length.
Example	Define an age type:

```
TYPE
 ageRange = [0..114];
```

Define a customer record:

```
TYPE
 personRec = RECORD
                firstName,
                middleName: ARRAY [0..12] OF CHAR;
                lastName   : ARRAY [0..20] OF CHAR;
                age        : ageRange;
                married    : BOOLEAN;
                company    : ARRAY [0..30] OF CHAR;
             END ;
```

Define colors for an RGB monitor:

```
TYPE
 pixelColors = (redPix, greenPix, bluePix);
 color       = SET OF pixelColors;
```

```
CONST
  black  = color{ };
  red    = color{ redPix   };
  green  = color{ greenPix };
  blue   = color{ bluePix  };
  yellow = color{ redPix,   greenPix };
  purple = color{ redPix,    bluePix };
  cyan   = color{ greenPix, bluePix };
  white  = color{ redPix,   greenPix, bluePix };

VAR
  currentColor: color;
BEGIN
  currentColor := red; (* etc *)
```

UseError exception (Files)   L/E

Description	UseError is raised by the *Files* module when an operation on a file is impossible because the disk is write-protected.
Declaration	**EXCEPTION** UseError;
Usage	**EXCEPTION** UseError: WRITELN('File error'); **END** UseError must be imported from the module *Files*.
Comments	Use the UseError exception to prevent your program from crashing when the user swaps disks unexpectedly.
See Also	EndError StatusError DeviceError DiskFull

Example   **MODULE** WriteFile;
                   FROM Files **IMPORT**
                    FILE, Create, WriteWord, Close, EndError,
                    StatusError, UseError;
                   VAR f: FILE; c: CARDINAL;
                   BEGIN
                    Create(f,'testfile.dat');
                    WriteWord(f,c);
                    Close(f)
                   EXCEPTION
                    EndError: WRITELN('End of file reached.'); Close(f)
                    | StatusError: WRITELN('Error file not opened');
                    | UseError : WRITELN('Disk not logged in.');
                   END WriteFile.

VAL standard function   S

Description   *VAL returns the value of type T with ordinal number X.*

Usage   Y:= VAL(T,X);

                   Argument X is of CARDINAL type.

                   Argument T is a CARDINAL, INTEGER, CHAR,
                   BOOLEAN, or enumeration-type identifier.

                   Result Y is of type T.

Comments   Note that *VAL(T,ORD(X))* = X for any type T.

Example

```
TYPE
 Direction: (Forward,Backward,Up,Down);
VAR
 Move: Direction;
 Value: CARDINAL;
 Character: CHAR;
BEGIN
 Move:= VAL(Direction,2);
(* Move is value Up *)

 Value:= 43;
 Character:= VAL(CHAR,Value);
(* Character is value '+' *)
```

Notice that this example is equivalent to type conversion. The last example could have been accomplished just as well with Character:= CHR(Value);). VAL provides a generic method for all ordinal types.

VAR declaration   R

Description

VAR signals the start of a declaration section for variables inside the declaration part of a module or a procedure.

In the parameter list of a procedure, **VAR** specifies that a variable is a variable parameter.

Usage

VAR vlist : anytype;

vlist is simply a list of identifiers.

anytype is any standard, imported, **or user-defined** type.

Comments

Variable declarations may appear anywhere in the declaration section of procedures and modules.

Example	Declare an integer *i*:

VAR i :INTEGER;

Declare a couple of characters:

VAR : carney,gleason : CHAR;

WAIT procedure (Processes)   L

Description	*WAIT* causes the currently executing process to stop and wait for a signal.
Usage	WAIT(siggy);
	WAIT must be imported from the library module *Processes*.
	siggy must be of type *SIGNAL* imported from *Processes*.
Comments	*WAIT* is used in conjunction with the other routines in *Processes*.
	When *WAIT* is called from an executing *Process*, it causes the execution of that *Process* to pause. It is then inserted into a queue associated with the signal *siggy*. It will not be revived again until the signal *siggy* is sent with the procedure *SEND*.
See Also	Awaited
	Init
	SEND
	StartProcess
	WAIT
Example	Wait for a signal:
	WAIT(signal)

WHILE statement   R

Description	WHILE repeatedly executes a sequence of statements as long as a Boolean expression returns a TRUE result.

Usage

```
WHILE BooleanExpression DO
 Statement sequence
END ;

Statement sequence = Statement {; Statement };
```

Comment
> The Boolean expression is evaluated before the statement sequence is executed; thus, the sequence is not necessarily executed.
>
> The repetition is terminated by an expression that returns a FALSE result.

Example

```
MODULE While;
CONST
 N = 10;
VAR
 I, Find: CARDINAL;
 DataArray: ARRAY [0..N] OF CARDINAL;

BEGIN
 WRITE("Input value to find: ");
 READ(Find);
 I:= 0;
 WHILE (I < N) & (DataArray[I] # Find) DO
  I:= I + 1
 END
END While.
```

WITH statement   R

Description   **WITH** aids access to record fields.

Usage   **WITH** Designator **DO**
                    Statement sequence
                   END ;

                   Designator is of **record type**.

                   Statement sequence = Statement {; Statement } ;

Comments   There are two ways to access the values of record fields.

                   The first method states explicitly both the record identifier and
                   the field:

                   Person.Name:= øAlfred Smithø

                   The other method uses the **WITH** statement:

                   WITH Person **DO**
                    Name := "Alfred **Smith**";
                    Age := 34;
                   END ;

                   This second method provides a clearer notation, especially for
                   large records or records with nested record fields.

Example
```
MODULE With;
TYPE
 PersonRecord = RECORD
                     Name: ARRAY [1..20] OF CHAR;
                     Age:  [0..200]
                     END ;
VAR
 Patient: PersonRecord;
BEGIN  ♦
 WITH Patient DO
  WRITE("Enter patient's name: ");
  READ(Name);
  WRITE("Enter patient's age: ");
  READ(Age)
 END ;
 WITH Patient DO
  WRITE(Name," is ",Age," years old.")
 END
END With.
```

WORD type (SYSTEM)   L

Description   *WORD* is a low-level type with variables that occupy one word (2 bytes) of memory. All variables occupying one word of storage are assignment-compatible with type *WORD*.

Usage   PROCEDURE p1 (w : WORD);

                WORD must be imported from the pseudomodule *SYSTEM*.

Comments   *WORD* is compatible with INTEGER, CARDINAL, BITSET, SET, enumeration types, POINTER types (including *AD-DRESS*), BOOLEAN, and CHAR.

                Because *BYTE* is a subrange of *WORD*, those variables compatible with *BYTE* are also compatible with *WORD*.

ARRAY OF *WORD* is compatible with every type. It may be used as a special open array parameter type that allows any variable to be passed as the actual parameter.

Only assignment operations are allowed on variables of type *WORD*; however, type-transfer functions may be applied to type *WORD*.

Example   Dump a variable:

```
MODULE LowLevelDump;
FROM InOut IMPORT WriteHex;
FROM SYSTEM IMPORT WORD,ADR;

PROCEDURE DumpVar(VAR v:ARRAY OF WORD);
VAR
 i :INTEGER
BEGIN
 WRITELN('There are ',HIGH(v)+1,' words allocated
 for v');
 FOR i := 0 TO HIGH(v) DO
  WRITE(' Address: $');
  WriteHex(ADR(v[i]),4);
  WRITE(i:3,' Value: $');
  WriteHex(v[i],4);
  WRITELN;
 END ;
END DumpVar;

VAR
  r: REAL;
rec: RECORD
    a,b,c,d: REAL;
  END ;
BEGIN
 DumpVar(r);
 DumpVar(rec);
END LowLevelDump.
```

WRITE standard procedure   S/E

Description   *WRITE* is a generalized write statement.

Usage   WRITE(vlist)

     or

WRITE(t, vlist);

t must be of type *TEXT* imported from *Texts*.

vlist is simply a list of variables or constants and optional formatting parameters.

Comments   If *t* is not specified, the standard output text *output* will be used.

WRITE will be translated by the compiler into the appropriate calls to the module *Texts*.

WRITELN is the same as *WRITE*, except *WRITELN* starts a new line when the output is complete.

Although *WRITE*, *WRITELN*, *READ*, and *READLN* are not included in the Modula-2 definition, programming without them can be very tedious; however, they are not portable.

One strategy might be to develop the code using these statements for debugging and development output, and use the explicit calls to *Texts* for the actual programmed output.

See Also   READ
                     READLN
                     Texts
                     WRITELN

Example   Write out the variable *nthingys* with an explanation:

```
WRITE('Number of things: ',nthingys)
```

In scientific notation, write out *pi* in a field 12 spaces wide with 7 digits in the mantissa.

```
pi := 4.0*ArcTan(1.0);
WRITE(' pi =', pi:12:-7);
```

WriteByte procedure (Files)   L

Description   *WriteByte* writes one byte to a file, which is compatible with the type *BYTE*.

Declaration   **PROCEDURE** WriteByte(f: FILE; ch: BYTE);

Usage   WriteByte(f, b);

WriteByte must be imported from the library module *Files*.

f must be of type *FILE* imported from *Files*.

b must be of a type compatible with *BYTE* imported from *SYSTEM*.

Comments   With this routine you can write out a file in small pieces.

WriteByte is appropriate with types that require only 1 byte of storage, namely BOOLEAN, CHAR, subrange types in the range 0 to 255, and enumeration types with at most 256 elements.

Example   Write out a boolean value *okay* to a file *f*:

```
WriteByte(f, okay)
```

WriteBytes procedure (Files)   L

Description	*WriteBytes* writes a specified number of bytes to a file.

Declaration   **PROCEDURE** WriteBytes(f: FILE; buf: ADDRESS; nbytes:
                            CARDINAL);

Usage   WriteBytes(f, bufStart, nbytes);

WriteBytes must be imported from the library module *Files*.

f must be of type *FILE* imported from *Files*.

bufStart must be of type *ADDRESS* imported from *SYSTEM*.

nbytes must be of type CARDINAL.

Comments   *nbytes* from the file *f* will be written from the block of memory starting from the address *bufStart* and extending to *bufStart* + *nbytes - 1*.

This is a low-level routine that should only be used for special purposes. When writing out variables, we recommend using the routines *WriteByte*, *WriteWord*, and *WriteRec*.

Example   Take a snapshot of the entire memory (except for the last byte):

WriteBytes(snap, 0, 65355);

WriteCard procedure (Texts)   L

Description   *WriteCard* writes out a cardinal number to a *TEXT* file.

Declaration   **PROCEDURE** WriteCard(t: TEXT; c, n: CARDINAL);

Usage   WriteCard(t, c, width);

                    WriteCard must be imported from the library module *Texts*.

                    t must be of type *TEXT* imported from *Texts*.

                    c must be of type CARDINAL.

                    width must be of type CARDINAL.

Comments   The number is output right-justified and blank-padded in a
                    field *width* characters wide.

See Also   CARDINAL
                    ReadCard
                    Texts
                    WRITE

Example   Write the **number** *n* **to the standard** output:

                    WriteCard(output, n, 6)

WriteChar procedure (Terminal)   L

Description   *WriteChar* writes a **character to the console.**

Declaration   **PROCEDURE** WriteChar(ch: CHAR);

Usage   WriteChar(ch);

                     WriteChar must be imported from the library module *Ter-
                    minal*.

                    ch must **be of type CHAR.**

Comments ·   *WriteString* does **the** same **thing for a string.**

Example   Write out a character *X* to **the screen:**

                    WriteChar("X")

WriteChar procedure (Texts)   L

Description	*WriteChar* writes a character to a *TEXT* file.
Declaration	**PROCEDURE** WriteChar(t: TEXT; ch: CHAR);
Usage	WriteChar(t, ch);

WriteChar must be imported from the library module *Texts*.

t must be of type *TEXT* imported from *Texts*.

ch must be of type CHAR.

Comments   In general, the *WRITE* statement is easier to use than *WriteChar*.

WriteString will do the same thing for a whole string.

See Also   ReadChar
Texts
WRITE
WriteString

Example   Write out character *MrMouse* to text *comix*:

```
MODULE WriteCharacter;
FROM Texts IMPORT TEXT, WriteChar;
VAR
 MrMouse: CHAR;
  comix: TEXT;
BEGIN
 (* other code *)
 WriteChar(comix, MrMouse)
END WriteCharacter.
```

WriteDouble procedure (Doubles)   L

Description	*WriteDouble* writes a double-precision **real variable** to a *TEXT* file.
Declaration	**PROCEDURE** WriteDouble(t: TEXT; d: LONGREAL; n: CARDINAL; m: INTEGER);
Usage	WriteDouble(t, d, width, digits);

WriteDouble must be imported from **the library module** *Doubles*.

t must be of **type** *TEXT* imported from *Texts*.

d must be of type LONGREAL.

width must be of type CARDINAL.

digits must be of type INTEGER.

Comments	The number *d* is output right-justified in a field *width* characters wide with *digits* digits after the decimal point.

If *digits* is negative, then the number is output in scientific notation with *digits* digits after the decimal point.

If *digits* is zero, the decimal point is omitted.

The *WRITE* and *WRITELN* procedures use this procedure to write LONGREALs with a default for *width* of 21 and a default for *digits* of -14.

See Also	Doubles ReadDouble

Example	Write a double-precision *d* to a text *t*:

```
WriteDouble(t, 2.99, 5,  2); yields " 2.99"
WriteDouble(t, 2.99, 5, 0); yields "    3"
WriteDouble(t, 2.99, 9, -2); yields " 2.99D+01"
```

WriteHex procedure (InOut)   L

Description	WriteHex writes a hexadecimal number out to output.
Declaration	**PROCEDURE** WriteHex(x,n: CARDINAL);
Usage	WriteHex(num, digits);
	WriteHex must be imported **from InOut.**
	num must be of type CARDINAL.
	digits must be of type CARDINAL.
Comments	The number will be converted to hexadecimal and printed out in a field that is at least *digits* wide. If the number doesn't need that many digits, it will be blank-padded.
	This and *WriteOct* are useful procedures to print out addresses and storage requirements in their natural form.
See Also	InOut
	WriteOct
Example	Write in hex **the number 32 in a field four digits wide:**
	WriteHex(32, 4) yields ' 20'

WriteInt procedure (Texts)   L

Description	WriteInt writes out an integer number to a TEXT file.
Declaration	**PROCEDURE** WriteInt(t: TEXT; i: INTEGER; n: CARDINAL)
Usage	WriteInt(t, i, width);
	WriteInt must be imported from the library module *Texts*.
	t must be of type *TEXT* imported from *Texts*.
	i must be of type INTEGER.
	width must be of type CARDINAL.
Comments	The number is output right-justified and blank-padded in field *width* characters wide.
See Also	INTEGER
	ReadInt
	Texts
	WRITE
Example	Write the number *n* to the standard output:
	WriteInt(output,n,6)

WRITELN standard procedure   S/E

Description	*WRITELN* is a generalized write statement that starts a new line when it is done.

Usage	`WRITELN(elist);`

          or

         `WRITELN(t, elist);`

         t must be of type *TEXT* imported from *Texts*.

         elist is simply a list of expressions and optional formatting parameters.

Comments   If *t* is not specified, the standard output stream *output* will be used.

WRITELN will be translated by the compiler into the appropriate calls to the module *Texts*. Thus, a change in the definition module of *Texts* could lead to *WRITELN* not working.

WRITELN is the same as *WRITE*, except *WRITELN* starts a new line when the output is complete.

Although *WRITE*, *WRITELN*, *READ*, and *READLN* are not included in the Modula-2 definition, programming without them can be very tedious; however, they are not portable.

A way around this is to develop code using these statements for debugging and development output, and use the explicit calls to *Texts* for the actual programmed output.

See Also   READ
                     READLN
                     Texts
                     WRITE

Example   Write out the variable *nthingys* with an explanation, then start
                   a new line:

                   WRITELN('Number of things: ',nthingys)

                   In scientific notation, write out *e* in a field 12 spaces wide with
                   7 digits in the mantissa, and start a new line:

                   e := Exp(1.0);
                   WRITELN(' e =', e:12:-7);

WriteLn procedure (Terminal)   L

Description   *WriteLn* starts a new line on the screen.

Declaration   **PROCEDURE WriteLn;**

Usage   WriteLn;

                   WriteLn **must be imported from the library module** *Terminal*.

Comments   The *Terminal* routines provide input and output to the screen
                   without the overhead of the stream abstraction in *Texts*.

See Also   Terminal
                   WriteChar
                   WriteString

Example   Write out an error message:

                   WriteString('Help !'); **WriteLn;**

WriteLn procedure (Texts)   L

Description	*WriteLn* writes an *EOL* to a *TEXT* file.
Declaration	**PROCEDURE** WriteLn(t: TEXT);
Usage	WriteLn(t);
	WriteLn must be imported from the library module *Texts*.
(	*t* must be of type *TEXT* imported from *Texts*.
Comments	The *WRITELN* statement is easier to use than this procedure.
	WriteLn(t) is equivalent to *WriteChar(t,EOL)*.
See Also	Texts WRITELN
Example	Write an *EOL* to the text *console*:
	WriteLn(console);

WriteLn procedure (InOut)   L

Description	Starts a new line on the current output device.
Declaration	**PROCEDURE** WriteLn;
Usage	WriteLn;
	WriteLn must be imported from the library module *InOut*.
Comments	*WriteLn* provides a standard new line routine.
	This procedure is defined in terms of the *Texts* module.

See Also   Terminal
                  Texts
                  WRITELN

Example   Write out an error message:

                  MODULE WriteMessage;
                  FROM InOut **IMPORT** WriteString, WriteLn;
                  BEGIN
                   WriteString('Help !'); WriteLn;
                  END WriteMessage.

WriteLong procedure (Texts)   L

Description   *WriteLong* writes **out a long integer number to a** *TEXT* fil∈

Declaration   **PROCEDURE** WriteLong(t: TEXT; l: LONGINT;
                  n: CARDINAL);

Usage   WriteLong(t, l, width);

                  WriteLong must be imported from the library module *Text*

                  t must be of type *TEXT* imported from *Texts*.

                  l must be of type LONGINT.

                  width must be of type **CARDINAL.**

Comments   In general, the *WRITE* statement is easier to use th
                  WriteLong.

                  The number is output right-justified and blank-padded in
                  field *width* characters wide.

See Also	LONGINT
	ReadLong
	Texts
	WRITE

Example	Write the number *n* to the standard output:

```
WriteLong(output, n, 6)
```

WriteOct procedure (InOut)   L

Description	*WriteOut* writes an octal number out to *output*.

Declaration	**PROCEDURE** WriteOct(x,n: CARDINAL);

Usage	WriteOct(num, digits);

WriteOct must be imported from *InOut*.

num must be of type CARDINAL,

digits must be of type CARDINAL.

Comments	The number will be converted to octal and printed out in a field at least *digits* wide. If the number doesn't need that many digits, it will be blank-padded.

This and *WriteHex* are useful procedures to print out addresses and storage requirements in their natural form.

See Also	InOut
	WriteHex

Example	Write out the number 8 in octal in a field four digits wide:

```
WriteOct(8, 4) yields ' 10'
```

WriteReal procedure (Texts)   L

Description   WriteReal writes out a real number to a TEXT file.

Declaration   **PROCEDURE** WriteReal(t: TEXT; r: REAL;
                              n: CARDINAL; digits: INTEGER);

Usage   WriteReal(t, r, width, digits);

                WriteReal must be imported from the library module *Texts*.

                t must be of type *TEXT* imported from *Texts*.

                r must be of type REAL.

                width must be of type **CARDINAL**.

                digits must be of type INTEGER.

Comments   In general, the *WRITE* statement is easier to use than
                WriteReal.

                The number is output in a field *width* characters wide, and the
                mantissa is *digits* long.

See Also   REAL
                ReadReal
                Texts
                WRITE

Example   Write the number *pi* to the *TEXT* file *doc*:

                WriteReal(doc, pi, 5, 2) yields " 3.14"
                WriteReal(doc, pi, 12, 2) yields "3.14"
                WriteReal(doc, pi, 5, 0) yields " 3"
                WriteReal(doc, pi, 10, -3) yields " 3.141E+00"

WriteRec procedure (Files)   L

Description	WriteRec writes a record or an array to a disk file declared as FILE.
Declaration	**PROCEDURE** WriteRec(f: FILE; **VAR** rec: **ARRAY OF** WORD);
Usage	WriteRec(f, r); *WriteRec* must be imported from the library module *Files*. *f* must be of type *FILE* imported from *Files*. *r* is any type.
Comments	This routine should be used for writing any structure larger than a byte or a word (see *WriteByte* and *WriteWord*).
See Also	Files, ReadRec, WriteByte WriteWord
Example	Write the record *widget[i]* to *newFile*: WriteRec(newFile, widget[i])

WriteString procedure (Terminal)   L

Description	*WriteString* writes a string to the terminal at the current cursor position.
Declaration	**PROCEDURE** WriteString(s: **ARRAY OF** CHAR);

Usage   `WriteString(s);`

                   WriteString must be imported from the library **module *Ter-
                   minal*.**

                   s must be of type **ARRAY OF CHAR.**

Comments   There is no stream (like console, input, or output) associated
                   with *Terminal* procedures; thus, output goes directly to the
                   console and redirection is not possible.

See Also   ReadString
                   Terminal

Example   Write a string *prompt* to the terminal:

                   `WriteString(prompt)`

WriteString procedure (Texts)   L

Description   *WriteString* writes out a string to a file declared as *TEXT.*

Declaration   **PROCEDURE** `WriteString (t: TEXT; s: ` **ARRAY OF** ` CHAR);`

Usage   `WriteString(t, s);`

                   WriteString must be **imported from the library module *Texts*.**

                   t must be of type *TEXT* imported from *Texts*.

                   s must be a character array.

Comments   In general, the *WRITE* statement offers easier output than
                   WriteString.

                   All of the characters in the string are output up until the first
                   zero byte or the end of the string is encountered.

See Also   ReadString
                    Texts
                    WRITE

Example   Write the string *name* to the *TEXT* file *doc*:

```
WriteString(doc, name)
```

WriteWord procedure (Files)   L

Description   *WriteWord* writes a variable compatible with *WORD* to a disk
                    file declared as *FILE*.

Declaration   **PROCEDURE** WriteWord(f: FILE; w: WORD);

Usage   WriteWord(f, w);

                    WriteWord must be imported from the library module *Files*.

                    f must be of type *FILE* imported from *Files*.

                    w must be of a type compatible with *WORD*, which is imported
                    from the pseudomodule *SYSTEM*.

Comments   Types compatible with *WORD* are those with storage of no
                    more than one word (2 bytes); these include INTEGER, CAR-
                    DINAL, BITSET, all pointers (*ADDRESS* is a pointer).

See Also   Files
                    ReadWord

Example   Write the cardinal *nwidgets* to *newFile*:

```
WriteWord(newFile, nwidgets)
```

APPENDIX A
Turbo Modula-2 and Turbo Pascal

This appendix examines the differences between Turbo Pascal and Turbo Modula-2 in a detailed manner. First, we will point out some of the features found only in Turbo Modula-2 and then we will compare the differences in the features of both languages.

In general, Turbo Modula-2 and Turbo Pascal are quite similar. Some features found in Turbo Modula-2 will be more familiar to Turbo Pascal programmers than they will be to regular Pascal programmers. This is because Turbo Pascal provides a rich set of standard procedures and useful extensions that are defined directly in the Turbo Modula-2 language.

Modula-2's separate compilation and library facilities provide a full and extensible set of primitives. The differences between the language implementations range from simple items like case sensitivity to complex issues involving type- and version-checking across compilation units.

Modula-2 can be used for large program development and for expressing operating system concepts like concurrency and interrupt handling. Rather than using Include files, Modula-2 separates programs textually into modules. Like Pascal procedures, modules can be used to control the scope of identifiers. Unlike procedures, the walls around a module are opaque in both directions, with only explicitly defined changes in scope. This control of scope occurs in both local (nested) and library (separately compiled) modules. In the case of a library identifier, any identifier needed from another module must be requested. And both local and library modules must explicitly make the identifier visible.

Library modules can be separately compiled. Though some other languages allow separate compilation, Modula-2 does full type- and version-checking. This helps prevent errors that occur because the program is textually broken into pieces. Thus you have the same safety as Turbo Pascal's Include files without having to recompile support procedures all the time.

Two new low-level features found in Turbo Modula-2 **are coroutines and** inter-

rupt handlers. As described in Chapter 8, coroutines simulate concurrent processes by using their own data space and sharing the processor.

While some people have expressed interrupt handlers in Turbo Pascal using in-line code, Modula-2 allows interrupt handlers to be defined in a high-level manner. This is done by initializing the machine and then having the interrupt handler install itself at some predetermined vector. The rest is handled by Turbo Modula-2.

What's the Difference?

The following overview compares the elements of Turbo Modula-2 to Turbo Pascal and should quickly acquaint the Turbo Pascal programmer with Turbo Modula-2. (For a complete explanation of any feature, refer to the body of this manual.)

To start, let's consider the following Pascal and Modula-2 programs:

```
PROGRAM prime( OUTPUT );          MODULE prime;
CONST                              CONST
 size = 8190;                       size = 8190;
VAR                               VAR
 i,k,prime,count : INTEGER;        i,k,prime,count : INTEGER;
 flags : ARRAY [0..size] OF        flags : ARRAY [0..size] OF
BOOLEAN;                          BOOLEAN;
BEGIN                             BEGIN
 count := 0;                        count := 0;
 FOR i := 0 TO size DO             FOR i := 0 TO size DO
  flags[i] := TRUE;                 flags[i] := TRUE;
                                  END;
 FOR i := 0 TO size DO BEGIN       FOR i := 0 TO size DO
  IF flags[i] THEN BEGIN            IF flags[i] THEN
   prime := i + i + 3;              prime := i + i + 3;
   k := i + prime;                  k := i + prime;
   WHILE k <= size DO BEGIN         WHILE k <= size DO
    flags[k] := FALSE;              flags[k] := FALSE;
    k := k + prime;                 k := k + prime;
   END ;                           END ;
   count := count + 1;             count := count + 1;
```

```
  END;                              END;
  END;                              END;
  WRITELN(count,' Primes');         WRITELN(count,' Primes');
END (* prime *).                  END prime.
```

Vocabulary

There are minor differences in the vocabularies of the two language implementations. For the most part, it is simple to convert a Turbo Pascal program into Turbo Modula-2.

Identifier Names

Turbo Modula-2's identifiers are written in the same manner as Pascal's. There are two major differences: (1) Turbo Modula-2 does not allow underscores in identifier names, and (2) it is case-sensitive. The latter has been made optional with a compiler switch. The following are some legal and illegal modula identifiers:

Legal   **Illegal**

AnotherIndent   An_Indent
n
N

Characters

Character constants may either be denoted by the character enclosed in single or double quotes or by the character's ordinal number written in octal notation and followed by the letter *C*. For example:

32C   A character constant representing a Control-Z
101C   The letter 'A'
'B'   The letter 'B'
'C'   The letter 'C'

Numbers

The numbers of Turbo Modula-2 are a superset of those allowed in Pascal. In ad-

dition to the integers and reals in Turbo Pascal, Turbo Modula-2 has an unsigned integer, called a CARDINAL, and two double-precision types, an integer (LONGINT) and a real (LONGREAL).

Like Turbo Pascal, Turbo Modula-2 allows you to specify the base and type of a constant number. In Turbo Pascal, a hexadecimal number is written starting with a dollar sign, such as $0F; but in Turbo Modula-2 hex numbers are written starting with a decimal digit and ending with the letter *H*, such as 0FH. The various classes are Octal, Hexadecimal, Character, Real, and Long. The following are examples of numeric constants in Turbo Modula-2:

23B   Octal cardinal   (19 decimal)
023H   Hexadecimal cardinal   (35 decimal)

Real constants are written the same as they are in Turbo Pascal--a decimal point must be present. The scale factor is preceded by *E*. For example:

1.1414   A single-precision real approximating the square root of 2
3.02E9   A single-precision real in scientific notation

Double-precision constants are just like real constants with a scale factor, but with a *D* in place of the *E*. For example:

0.0D0   A double-precision real that is **a very precise zero**

In the same way a *C* following a octal number makes a CHAR constant, an *L* following a integer makes a LONGINT constant. Of course, the integer can be larger than normal integers if followed by an *L*, as shown in the following:

1483236283L   A long integer over a trillion (1,483,236,283).
-1L   Internally, this long integer is 4 bytes of ones.

Strings

The two differences between strings in Turbo Pascal and Turbo Modula-2 are (1) strings can be enclosed in either single or double quotes, and (2) the quote used to enclose the string may not appear in the string.

In Pascal, to have a quote within a string you would write the **quote** twice. In

Turbo Modula-2, you may include whichever quote you are not using to enclose the string. Thus, the following strings yield the same result:

Modula-2   **Pascal**

" ' "   ' " '
' " '   ' " '

Here are other examples of legal strings in Modula-2

(" bba"   'The language "Modula-2"'   "Peter's programs"

Set Constants

Sets in Turbo Modula-2 are substantially different than those in Turbo Pascal. In Turbo Modula-2, sets are represented by one machine word and thus are very efficient. Tests for set membership are much faster. However, there is one drawback: A machine word has only enough bits to represent a set of 16 elements. Pascal programmers who are used to using **SET OF CHAR** may miss it; however, there are other ways to obtain the same results.

The syntactic differences are that sets are delimited by curly brackets instead of the square brackets used in Pascal. A set must be preceded by its type identifier when used in expressions, otherwise, it is assumed to be the predefined set type BITSET. For example:

```
TYPE
  UserSet = SET OF (red, green, blue);
CONST
  hibit = {7};
  CurrentColor = UserSet{green};
```

Comments

In Turbo Pascal, comments can only be nested one level deep by using one of the two forms within the other. In Turbo Modula-2, the only form of comment uses the delimiters (* and *). (As previously shown, the curly braces, { and }, are used to denote sets.) In contrast to Turbo Pascal, Turbo Modula-2 comments may be nested to any depth. This is especially useful for debugging programs. For example:

```
c = 2.997925E8; (* the speed of light in meters per second *)
(*D WriteReal(c); (* printout for debugging *) D*)
```

Declarations

In contrast to standard Pascal, Turbo Modula-2's order of declarations (like Turbo Pascal's) is not fixed. Constant, type, variable, and procedure declarations can be written in any order. This gives the programmer more freedom to group related items together. Of course, every declaration section must be preceded by the appropriate CONST, TYPE, VAR, or PROCEDURE symbol.

Constant Declarations
There are two major differences between Turbo Pascal's constants and Turbo Modula-2's: (1) Turbo Modula-2 allows constant expressions in declarations. (2) There is no equivalent for Turbo Pascal's typed constants (which are actually pre-initialized variables).

Turbo Modula-2's constant expressions may be used anyplace Pascal allows only constants. Constant expressions consist of constants connected by the usual operators. Some examples of declarations follow (constant expressions are seen in lines 2, 5 and 8):

```
CONST
 PiByTwo = 3.141592 / 2.0;
 version = "1.6 last changes: Oct 84";
 mask = {0..3,8};
 Truth = NOT FALSE;
 Size = 1000;
TYPE
 a = ARRAY [0..Size-1] OF CHAR;
```

Turbo Pascal's typed constants can be simulated with normal Turbo Modula-2 variables and the initialization part of modules. The only difference is that the values are initialized at runtime instead of at load-time.

Type Declarations

Turbo Modula-2 has all of the types offered by Turbo Pascal, whether defined directly as standard identifiers or as library types. In addition, Turbo Modula-2 has procedure types. In defining types in Turbo Modula-2, a constant expression may be used anywhere a constant is used to define a type in Pascal.

Standard types in Turbo Modula-2 are CHAR, BOOLEAN, BITSET, IN-TEGER, CARDINAL, REAL, LONGINT, LONGREAL, and PROC. Of these, INTEGER, BOOLEAN, CHAR, and REAL are used exactly as they are in Turbo Pascal. We have already mentioned the addition numeric type, CARDINAL, LONGINT, and LONGREAL; the remaining types, BITSET and PROC, will be explained shortly. The user-defined types of both languages are similar, but have slight differences.

Subrange and enumeration types are the same for both languages except Turbo Modula-2 encloses in square brackets ([]) the values defining a subrange. For example:

```
TYPE
   bitnumber  =  [0..wordlength-1];
   smallint  =  [0..255];
```

The presence of the types INTEGER and CARDINAL causes a slight ambiguity when defining a subrange. In the previous example, it's not known if the base type of the subrange is INTEGER or CARDINAL. This problem is resolved by the convention that the base type is assumed to be CARDINAL if the lower bound is not negative; otherwise, it is INTEGER. You may override this convention by explicitly specifying the base type as follows:

```
TYPE
   ismall = INTEGER[0..255];
```

Arrays

As with subranges, the syntax of an array is slightly different in each language. When the bounds of an array are specified by a subrange, brackets are not needed; for example:

```
TYPE
  color = (brown,purple,orange);
  c = ARRAY color OF CHAR;
  a = ARRAY [0..9],[0..9] OF REAL;
  charkind = ARRAY OF CHAR (letter,digit,special,illegal);
```

Records

In Turbo Modula-2, records without variant parts are identical to those of Turbo Pascal. Turbo Modula-2 variant parts have a somewhat different syntax. If the name of the tag field is omitted, the colon and the type must still be written. The vertical bar serves to separate cases. Like Turbo Pascal, label ranges (for example, 0..5) and an optional ELSE part may be used. More than one variant part is allowed and it need not be written at the end of the record, as in Turbo Pascal. Variant parts in Turbo Modula-2 must have an END statement; for example:

```
complex = RECORD x, y : REAL END ;


sneaky = RECORD
            CASE : BOOLEAN OF
             FALSE : c : CARDINAL              |
             TRUE  : p : POINTER TO CARDINAL   |
            END
            END ;


demo = RECORD
          a, b : CARDINAL;
          x : REAL;
          CASE t1 : CARDINAL OF
           0..3,7   : f1,f2 : File                       |
           4,6      : name  : ARRAY [0..7] OF CHAR   |
          ELSE
                     link    : POINTER TO sneaky
          END ;
          d, e : BOOLEAN ;
          CASE c : color OF
```

```
     red, blue :        |
     green : g : date   |
   END ;
   last : BOOLEAN ;
END
```

Procedure Types

Procedure types are new to Turbo Pascal programmers. They can be used to define the interface of a procedure that is passed to some other procedure as a parameter. This is useful for allowing one data access routine to perform many different functions on the data.

For example, you may pass a tree-traversal procedure a procedure parameter that prints the node, uses the node for a calculation, or performs some other function on the nodes of the tree. The point is that each of these operations have the same interface, which can be defined globally and exported.

The type PROC is a predefined procedure type with no parameters. Variables of this type may receive assignments from procedures declared as

```
PROCEDURE Foo;
BEGIN
  (* Statements *)
END Foo;
```

Procedure variables are defined with the reserved word **PROCEDURE**, followed by a formal type list. In contrast to normal parameter lists, the names of the parameters are not given. Types are separated by commas and may be preceded by a **VAR** to indicate variable parameters. Function procedures declare a result type (see the section, "Function Procedures").

Variables of a procedure type may assume as their values procedures whose formal parameter list is compatible with the formal type list of the procedure type. However, procedures local to another procedure and standard procedures may not be assigned to procedure variables. Note that arithmetic and file-handling procedures (for example, *Sin* and *Open*) are not standard procedures in Turbo Modula-2. They are library procedures and can be assigned to procedure variables of the correct type. For example:

```
MODULE ProcedureVars;
FROM Texts IMPORT TEXT;
TYPE .
 RealFunc = PROCEDURE (REAL): REAL;
 TextDriver = PROCEDURE(VAR Text, CHAR);
VAR
 MyExp: RealFunc;
 MyWriteChar: TextDriver;
 MyClearScreen: PROC;    (* Predefined parameterless procedure *)

PROCEDURE MyExponentiation(r:REAL): REAL;
BEGIN
 (* Statements *)
END MyExponentiation.

PROCEDURE MyWriteCharacter(VAR t: TEXT; ch: CHAR);
BEGIN
 (* Statements *)
END MyWriteCharacter;

PROCEDURE MyClearTheScreen;
BEGIN
 (* Statements *)
END MyClearTheScreen;

PROCEDURE ExecuteP(p:PROC);
BEGIN  p;
END ExecuteP;

BEGIN   (* Assign procedures to the procedure variables *)
 MyExp := MyExponentiation;
 MyWriteChar := MyWriteCharacter;
 MyClearScreen := MyClearTheScreen;
 (* Execute procedure variables *)
 MyClearScreen;
 WRITE(MyExp(29.0));
 MyWriteChar(output "A");
 ExecuteP(MyClearScreen);
END ProcedureVars.
```

Variable Declarations

Turbo Modula-2's variable declarations are identical to those of Pascal: A list of variable identifiers are given (separated by commas), along with a colon and the variable's type. For example:

```
i, j, k    : INTEGER;
printer    : textwriter;
```

Like Turbo Pascal, Turbo Modula-2 offers a facility to specify the address of ͎ riable. This must be considered a low-level facility and must be used with care. The address is specified in brackets after the variable identifier. Unlike Turbo Pascal, Turbo Modula-2 restricts absolute variables from assuming dynamic values. Thus when a Turbo Pascal program uses a local variable "absoluted" with a parameter to the procedure, the Turbo Modula-2 equivalent would simply be pointer assignments that may require type coercion. Variables in Turbo Modula-2 are made absolute as shown here.

```
maskregister [OFFCDH]: BITSET;
```

Procedure Declarations

Procedures in both languages are declared in much the same way. Unlike Turbo Pascal but available in other Pascals, Turbo Modula-2's procedures and functions may be passed as parameters. The parameter must be declared as an already defined procedure type (shown in the previous section, "Variable Declarations").

Open Array Parameters

(͎n array parameters allow arrays declared of different length to be passed to the same procedure. In Turbo Pascal, this can be done only with string parameters, and only when the *V* compiler option is turned on. In Turbo Modula-2, there is a provision for open arrays of any type and there is a mechanism to dynamically obtain the upper bound of open array parameters.

Within the procedure, the lowest array element always has an index of zero. The index of the highest array element can be obtained as *HIGH(a)*, *a* being specified as **ARRAY OF** <some type>. For example:

```
PROCEDURE writevector(v : ARRAY OF REAL);
VAR
 i : CARDINAL;
BEGIN
 FOR i := 0 TO HIGH(v) DO WRITE(v[i]) END
END writevector;
```

Untyped Parameters

Turbo Pascal allows you to skip type-checking with untyped parameters by simply leaving off the type specification. This is useful for writing generic procedures. Of course, Turbo Modula-2 also allows this with an explicit declaration that uses the *SYSTEM* type WORD. Turbo Modula-2 goes one step further by allowing you to dynamically determine the size of the object passed with the standard procedures *SIZE* and *HIGH*. This is shown in the following example:

```
MODULE Untyped;

PROCEDURE foo(object: ARRAY OF WORD);
BEGIN
 WRITELN(SIZE(object), HIGH(object));
END foo;

VAR
 c: CARDINAL;
 r: REAL;
BEGIN
 foo(c);
 foo(r);
END Untyped.
```

The first call to the procedure *foo* results in the output of 2 and 0. Thus, the procedure knows the object passed is 2 bytes and the highest index into the array of words is 0. During the second call, this procedure outputs a 4 and a 1, indicating the object is 4 bytes and occupies positions 0 and 1 of the word array.

Function Procedures

In Turbo Modula-2, FUNCTION is no longer a reserved word; function declarations differ from procedure declarations only by the indication of a result type.

The following is an example of a function declaration that finds the length of a string:

```
PROCEDURE len(s: ARRAY OF CHAR): CARDINAL;
(* return length of string s *)
VAR
 i : CARDINAL;
BEGIN  i := 0;
 WHILE (i <= HIGH(s)) & (s[i] # OC) DO i := i+1 END ;
 RETURN i
 D len;
```

Expressions

Turbo Modula-2 expressions are very similar to those of Turbo Pascal. The usual operators, +, -, *, /, DIV, and MOD, are available for operands of type IN-TEGER, CARDINAL, and REAL, (DIV and MOD apply to INTEGER and CARDINAL, / applies to REAL). Of course, those operators applicable to IN-TEGER also work on LONGINT, and the same is true for REAL and LONGREAL.

There are, however, no implicit conversions. The following example would therefore be legal in Turbo Pascal but illegal in Turbo Modula-2, because REAL and CARDINAL are not compatible:

```
x := 1.0 + 1;
```

The logical operators AND (also written &), OR, and NOT (also written ˜), are available. If the first operand of an AND evaluates to FALSE, the second is t evaluated. Similarly, the second operand of an OR is not evaluated if the first one is TRUE. This rule sometimes shortens programs by eliminating a Boolean flag and a GOTO statement. For example:

```
WHILE (I # 0) AND (s[I] > 0) DO
 (* Something *)
END ;
```

would have to be translated to the following Pascal statement if *s* is not defined for *I* equal to 0:

```
while I < > 0 do begin
 if s[I]<=0 then goto endwhile;
 (* Something *)
end ;
endwhile:
```

Set Operators

The major benefit of Turbo Modula-2 sets is the ability to treat bits in a word as elements in a set. This makes bit manipulation very easy and defines it directly in the language instead of as an extension as in Turbo Pascal.

The type BITSET is a special predefined set type. It is declared as

```
TYPE
 BITSET = SET OF [0..wordlength-1];
```

where wordlength is the word length of the computer, which is 16 for Turbo Modula-2.

Since Turbo Modula-2 sets only take one word, the operations defined for sets can be viewed as equivalent to Turbo Pascal bitwise operations. Thus Turbo Modula-2 set operators can be viewed as abstract set operations or low-level bitwise operations as seen in Turbo Pascal equivalents.

Operation	Modula Symbol	Pascal Symbol
Union	+	OR
Difference	-	XOR and then AND
Intersection	*	AND
Symmetric Difference	/	XOR

In terms of set operations available in Turbo Pascal, only / is new. It is called symmetric set difference and is an exclusive OR. The resulting set contains all elements that are in either the first set operand or the second, but not in both. The following is an example using sets:

```
MODULE ExampleSet;
TYPE
 colors = (Red,White,Blue,Orange,Purple,Black,Yellow,Green,Cyan);
 flagColors = SET OF colors;
```

```
CONST
 frenchFlag = flagColors{Red,White};
VAR
 currentColors: flagColors;
BEGIN
 currentColors := flagColors{Red};
 currentColors := currentColors + flagColors{White};
 IF (currentColors=frenchFlag) THEN
  WRITELN('Viva la France !');
 END ;
END ExampleSet;
```

Familiar to Turbo Pascal programmers is Turbo Modula-2's alternate use for the preceding logical operators (**AND, OR, XOR**); in Turbo Pascal, these are also bitwise operators. As previously shown, Turbo Modula-2 has set operators that double as bitwise operators. The only bitwise operators defined in Turbo Pascal and not in Turbo Modula-2 are the shl and shr operators. However this is not a problem because Turbo Modula-2 translates multiplication and division by 2 into machine-language shifts in the appropriate direction. Thus, we have the following equivalents for the integer variable *I*:

Modula	Pascal
I * 2	I **shl** 1
I * 2 * 2	I **shl** 2
I * 256	I **shl** 8
I **DIV** 16	I **shr** 4

Turbo Modula-2 and Turbo Pascal use the same relational operators. The only 'ifference is that Turbo Modula-2 provides an additional inequality operator, the pound sign ($\#$), which has the same effect as $<>$. Both symbols are allowed in Turbo Modula-2.

Operands in expressions in Turbo Modula-2 are very similar to those found in Pascal. The familiar operations of indexing,, field selection, dereferencing, and function invocation are available. One difference is seen in expressions that include parameterless function calls. Turbo Modula-2 requires the empty parameter list to be specified in function calls. This helps distinguish function identifiers from variable and constant identifiers.

The following are examples of expressions in Turbo Modula-2:

`c + b*3`	Integer or cardinal expression
`list IN options`	Boolean expression
`sum + a[i,k]*a[k,j]`	Numeric expression using arrays
`(ch >= "A") & (ch <= "Z")`	Boolean expression with relations
`s * {0..3}`	Set expression
`Exp(Random( ))`	Real expression with nested function calls

Statements

Like Pascal, the most elementary statement in Turbo Modula-2 is the assignment statement. It is stricter in Turbo Modula-2 than in Turbo Pascal in that no implicit type conversions are made. This means you may not assign an integer expression to a real variable, as is allowed in Turbo Pascal. Of course, Turbo Modula-2 provides a mechanism to do this with explicit type-transfer functions; thus we have the following equalities (where x is a real and i is an integer):

Modula   **Pascal**

```
x := FLOAT(i);                      x := i;
i := TRUNC(x * FLOAT(i));           i := trunc(x * i);
```

Turbo Modula-2's structured statements have a more modern syntax that does away with Turbo Pascal's compound statement (**begin ... end**). Where Pascal requires the compound statement, Turbo Modula-2 allows a statement sequence to be terminated by an **END** statement.

Procedure Calls
The procedure call statement remains essentially unaltered in Turbo Modula- 2. If a procedure has no parameters, empty parentheses are allowed but not required, as in function procedures.

Looping Statements
Turbo Modula-2's **WHILE and REPEAT statements are essentially the same as** Turbo Pascal's.

Turbo Modula-2 has an **EXIT** statement, but it is used to terminate the **LOOP** statement. The **LOOP** statement is the same as a Turbo Pascal's **repeat until**

FALSE or **while** TRUE **do begin end**, except that instead of using a GOTO to exit the endless loop, Turbo Modula-2 uses the explicit **EXIT** statement. **EXIT** causes control to pass to the statement directly after the **END** that matches the **LOOP** statement.

The **FOR** statement has been slightly changed. A step value may now be given using a **BY** clause (default is +1 if the **BY** part is left out). The step value must be a constant expression. The Turbo Pascal **downto** symbol is no longer used. You now simply specify a step value of -1. For example:

```
FOR i := 0 TO size DO
  flags[i] := TRUE          (* Notice the explicit END *)
END ;
FOR i := n-2 TO 2 BY -1 DO   (* Pascal's DOWNTO *)
  s[i] := (b[i] -
a2[i]*s[i+1])/a1[i];        (* Any number of statements *)
  j := j + i                 (* between DO and END *)
END ;
```

CASE Statements
Turbo Modula-2's **CASE** statement has a different syntax than Pascal's. The vertical bar | is used to separate cases, thus eliminating the need for **begin end** in the statement part of the case statement. Like Turbo Pascal, Turbo Modula-2 allows ranges of values for **CASE** labels (like "A".."Z") and an optional ELSE part. For example:

```
CASE ch OF
| "A".."Z","a".."z"          : chartype := letter
| "0".."9"                   : chartype := digit
| " ",'"',"#","&".."/",
  ":".."A","[".."^","{".."~"  : chartype := special
|
ELSE    WRITELN('illegal

                character'); chartype := illegal
END ;
```

Note that the first and last vertical bars in this example are optional.

WITH Statements

The **WITH** statement has remained essentially unchanged. While in Turbo Pascal a list of record variables is allowed after **WITH**, Turbo Modula-2 makes you nest **WITH** statements to achieve the same effect. For example:

```
WITH p ^ DO      (* in Pascal, only one WITH would have been used *)
  WITH valu ^ DO
    typ := reel; rval := nxrval;
  END ;
  link := head
END ;
```

RETURN Statements

There are two reserved words that take the place of Turbo Pascal's **goto** statement: the **EXIT** statement (already discussed) and the **RETURN** statement. The **RETURN** statement serves to terminate procedures the same way as the **exit** statement in Turbo Pascal. In addition, it is used to return function results instead of an assignment to the Pascal function identifier.

Standard Procedures in Turbo Modula-2

Several standard functions and procedures found in Turbo Pascal are no longer directly available in Turbo Modula-2: Some standard procedures have been added and some have been left to be implemented in library modules. Only the most important and commonly used standard procedures are still within the language.

Mathematical functions like *Sqrt*, *Sin*, *Exp*, and so on, are no longer standard functions. They must now be imported from the module *MathLib* (see Chapter 11). File-handling procedures must also be imported from the appropriate library modules (*Files*, *Texts*, or *InOut*).

The Turbo Pascal functions *pred* and *succ* can be replaced by the procedures *INC* and *DEC*. These procedures accept one argument of any scalar type and one count argument that determines the size of the increment or decrement.

Thus, Turbo Pascal statements that look like the following:

```
i := i + 1;
color := pred(Green);
```

will look like this in Turbo Modula-2:

```
INC(i);
DEC(color);
```

NEW and *DISPOSE* still exist and are translated by the compiler into calls to procedures *ALLOCATE* and *DEALLOCATE*, which must be imported from the module *STORAGE* but may also be redefined within the user program. In contrast to Pascal, a *FLOAT* function exists in Turbo Modula-2. You may be surprised to find that there is no *ROUND* or *EVEN* function; instead, use *Entier(Value + 0.5)* for negative numbers and *TRUNC(Value + 0.5)* for positive numbers.

Tables A-1 through A-3 show which identifiers remain defined within the language. If you don't find what you're looking for here, try Chapter 12, "Turbo Modula-2 Reference Directory." Table A-1. Standard Modula-2 Functions

ABS(x)

Returns absolute value of x; result type is the same as argument type.

CAP(c)   Returns argument and result type are of type **CHAR, the** capital letter corresponding to c.

CHR(x)   Returns the character with ordinal number x.

FLOAT(x)   Converts value x of type *CARDINAL* to type **REAL.**

HIGH(a)   Returns high index bound of array a.

MAX(T)   Returns the largest element of the argument type.
                T is CARDINAL, INTEGER, BOOLEAN, CHAR, REAL,
                LONGREAL, LONGINT, any enumeration or scalar type.

MIN(T)   Returns the smallest element of type T.

ODD(x)   Returns TRUE is x is odd; otherwise it returns FALSE.

ORD(x)   Returns the ordinal value of x, where x is of type
                BOOLEAN, CHAR, INTEGER, CARDINAL, or every
                enumeration type.

SIZE(T)	Returns the storage requirements of type T in bytes.
SIZE(x)	Returns the storage requirements of variable x in bytes.
TRUNC(x)	Returns *CARDINAL* result; x of type *REAL* truncated to integral part.
VAL(T,x)	Returns the value of type T, which has ordinal number x. T is BOOLEAN, CHAR, INTEGER, CARDINAL or every enumeration type.

Table A-2. Additional Functions Offered by Turbo Modula-2

LONG(x)	Converts its argument to LONGINT
INT(x)	Converts its argument to INTEGER
CARD(x)	Converts its argument to CARDINAL
FLOAT(x)	Converts its argument to REAL
DOUBLE(x)	Converts its argument to LONGREAL

Note: These functions only work on numeric types.

Table A-3. Standard Procedures in Turbo Module-2

DEC(x)	$x := x - 1$
DEC(x,n)	$x := x - n$
EXCL(s,i)	$s := s - \{i\}$
HALT	Terminates **program execution**
INC(x)	$x := x + 1$
INC(x,n)	$x := x + n$
INCL(s,i)	$s := s + \{i\}$

Note that type identifiers may be used like function identifiers denoting so-called type-transfer functions. (This is considered a low-level facility and is discussed in Chapter 8.)

APPENDIX B
Installation Procedures

The installation program INSTM2 has two functions. Its primary function is to allow you to configure the Turbo Modula-2 system to your hardware and to configure the Turbo Modula-2 editor to your own taste. Thus, you may tell the system what type of terminal you have and which disk drives you wish it to search for ceded files. You may also change the editing commands to more familiar ones.

Note: The installation must be performed if you want Turbo Modula-2 to work the way it is described in this manual.

The second function of INSTM2 is to allow programs compiled on your setup to be re-installed on different terminals. This allows you to distribute .COM files (executable files) to your customers without knowing what type of terminal they have. All you need to do is include INSTM2 files along with your executable program.

In this chapter we will describe how to use INSTM2 to install Turbo Modula-2, and then discuss programs in general.

Installing M2

By this point we assume you have made a backup copy of the Turbo Modula-2 distribution disk, as well as a working disk. (If not, these procedures are described in Chapter 1, »Getting Started.«)

Place your working disk in the logged drive. **It should contain all the installation** files and the M2.COM file, listed as follows:

```
INSTM2.COM
INSTM2.OVR
INSTM2.DTA
M2.COM
```

Type INSTM2 to start the installation program. After the prompt, press ⟨ ⏎ ⟩ and the following menu will appear:

```
           Modula-M2 system installation menu.
           Choose installation item from the following:


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

                   Enter S, C, M, or Q:
```

There are four possible choices at this menu: Screen installation, Command installation, Miscellaneous, and Quit. To select a choice, enter one of the highlighted letters shown in square brackets. Any legal selection except _Q_ will bring up a new menu: Quit will return you to the operating system (unless it is being run from the M2 shell).

Screen installation allows proper functioning of Turbo Modula-2's WordStar-like editor and the various screen manipulation functions in the module _Terminal_.

Command installation is performed if you wish to alter the WordStar-style editing commands. We recommend that you become familiar with the capabilities of the editor before you change any of the editing keys. Generally, you only need to go through the Command installation if your keyboard is missing the necessary keys or if you want your Turbo Modula-2 editor to be compatible with a non-WordStar-like editor. Miscellaneous installation is used in two instances. First, if your keyboard does not have certain display characters used in standard Modula-2, this command will allow you to substitute an alternate character or sequence of characters for those missing from your keyboard. Second, this command allows you to tell Turbo Modula-2 where it should look for files when it is compiling and linking.

Screen Installation

When you press ⟨ S ⟩ to perform Screen installation, the following menu will appear:

```
 1) ADDS 20/25/30            17) Otrona Attache
 2) ADDS 40/60               18) Qume
 3) ADDS Viewpoint-1A        19) RC-855 (ITT)
 4) ADM-3A                   20) Soroc 120/Apple CP/M
 5) DEC Rainbow, 8 bit       21) Soroc new models
 6) Ampex D80                22) SSM-UB3
 7) ANSI                     23) Tandberg TDV 2215
 8) Morrow MDT-20            24) Teleray series 10
 9) Hazeltine 1500           25) Teletex 3000
10) Hazeltine Esprit         26) Televideo 912/920/92
11) Kaypro with hilite       27) Texas Instruments
12) Kaypro, no hilite        28) Visual 200
13) Lear-Siegler ADM-20      29) Wyse WY-100/200/300
14) Lear-Siegler ADM-31      30) Zenith
15) Liberty                  31) None of the above
16) Osborne 1                32) Delete a definition
```

Which terminal? (Enter no. or ^Q to exit):

Select the terminal type you will be using while running Turbo Modula-2 by entering the appropriate number for your terminal at the prompt. Several things are dependent on the terminal you are using. Of course, the editor needs to know about the terminal, but your programs must also know the capabilities of the terminal they are running on. This information is available from the standard module *Terminal*.

Before installation is actually performed, you are asked the question:

Do you want to modify this definition before installation? (Y/N)?

This allows you to modify one or more of the values being installed (described in the next section). If you do not want to modify the terminal definition, type *N* to complete the installation and be returned to the Main Installation Menu.

Manual Installation

If your terminal is *not* on the menu, however, you must define the required values yourself (refer to your terminal manual for these values). An alternative method exists if you also own a copy of Turbo Pascal and have installed it for your configuration.

Using the Turbo Pascal TINST.DTA File

Many people with unusual terminals have Turbo Pascal because (like Turb
Modula-2) it runs on almost anything with a Z80 in it. These people have alread
gone through a painstaking installation procedure that required searching obscu
manuals for cryptic codes. If you are one of these people, relax; you can use yo
existing TINST.DTA file simply by renaming it to INSTM2.DTA. (Note that tl
.DTA file only contains data for screen commands not keyboard commands.)

Use the renamed file as if it were the one on the distribution disk. When tl
terminal menu comes up you will see your terminal listed with any others in tl
TINST.DTA file.

Entering Terminal Codes

Enter the number corresponding to »None of the above« and answer the questio
one by one as they appear on the screen.

The following section describes each installable command in detail. If your te
minal does not support a command, press ⟦ ↵ ⟧ at the prompt. If *Delete lir
Insert line,* or *Erase to end of line* are not installed, they can be emulated in t
editor. However, these emulation functions are not available in the module *Te
minal*; the module *Terminal* only provides a method to check if the functions a
available so that your programs can emulate them if necessary. Commands m
be entered by pressing the appropriate keys or by entering the decimal or he
adecimal ASCII value of the command. If a command requires the two characte
['ESCAPE'] and ['='], you may

☐ First press the [Esc] key, then the [=], and the entry will be echoed with a
   propriate labels.

☐ Or you can enter the decimal or hexadecimal values separated by spac
   (Hexadecimal values must be preceded by a dollar sign ($); for example,
   61, $1B 61, or $1B $3B, which are all equivalent.) These two methods can
   be mixed since once you have entered a specific character (except a dol
   sign), the rest of the command must be defined in that mode, and vice ver
   If you need to delete an entry, you may enter a hyphen as the first charac
   on the line; the text *Nothing* is echoed. This is only effective on comma
   that expect more than a single number of a Yes/No answer.

Terminal Properties

The following is an explanation of each question asked by the terminal installation program.

Terminal type:

Enter the name of the terminal you are about to install. When you complete INSTM2, the values will be stored and the terminal name will appear on the initial list of terminals. If you later need to re-install Turbo Modula-2 to this terminal, you can do so by choosing the terminal from the list.

Send an initialization string to the terminal?

If you want to initialize your terminal when Turbo Modula-2 starts (for example, to download commands to programmable function keys), press ⎡ Y ⎤ to answer this question; otherwise, press ⎡ ⏎ ⎤ .

Send a reset string to the terminal?

Define a string to be sent to the terminal when Turbo Modula-2 terminates. The description of the preceding initialization command also applies here.

CURSOR LEAD-IN command:

Cursor Lead-in is a special sequence of characters that tells your terminal that the following characters comprise 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 address in the format: **first** ROW, **then** COLUMN. If this is the case on your terminal, press ⎡ N ⎤ . If your terminal requires COLUMN first, then ROW, press ⎡ 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 this is true for yc
terminal, press ⌐Y⌐ . If your terminal expects the cursor address as ASC
digits, press ⌐N⌐ . You are then asked the supplemental 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, bc**
foreground and background, if applicable.

Does CLEAR SCREEN also HOME cursor?
This is normally the case; if it is not so on your **terminal, press** ⌐N⌐ **and (**
fine the cursor *HOME* command.

DELETE LINE command:
Enter the command that dele**tes the entire line at the cursor position.**

INSERT LINE command:
Enter the command that inser**ts a line at the cursor position.**

ERASE TO END OF LINE command:
Enter the command that erases the line at the cursor position, starting from
cursor position through to the end of the line.

START OF 'LOW VIDEO' command:
If your terminal supports different video intensities, then define the command t
initiates *dim video*.

START OF 'NORMAL VIDEO' command:
Define the command that sets the screen to show characters in 'normal vid
Some terminals have one command that toggles the video mode; thus this c
may be the same as the last one.

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, press [N] to be returned to the ter-
minal 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 installa-
tion will not be performed.

   When you press ⟨ Y ⟩ in response to this question, installation is completed,
the installation data is written to M2.COM, and you are returned to the main
menu. Installation data is also saved in the installation data file and the new ter-
minal will appear on the terminal selection list when you run INSTM2 in the
future.

Installation of Editing Commands

The built-in editor performs a number of commands. Each function may be ac-
tivated by either of two commands: a primary command and a secondary com-
mand. Primary commands are undefined, but can be easily defined to fit your
taste or your keyboard. The secondary commands are installed by Borland and
comply with the standard set by WordStar. From the installation main menu, press
⟨ C ⟩ for Command installation. The first command will appear like the follow-
ing:

```
CURSOR MOVEMENTS:

1:   Character left     Nothing  ->
```

This means that no primary commands have been installed to move the cursor one character left. If you want to install a primary command (in addition to the secondary WordStar-like Control-S, which is not shown here), you may enter the desired command after the -> prompt in either of two ways:

1. Simply press the key you want to use. It can be a function key; for example, a left-arrow key or any other key or sequence of keys that you choose (the maximum is 2).

   The installation program responds with a mnemonic of 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 given previously, your screen will look like the following:

   CURSOR MOVEMENTS:

   1:   Character left   Nothing -> <ESC> a

2. Instead of pressing the 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 and separated by spaces. Hexadecimal values are prefixed by a dollar sign ($1B).

   This may be useful to install commands 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.

In both cases, terminate your input by pressing ⏎ . Note 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 (-) sign 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, all of which may be installed to your specifications. If you make an error in the installation, such as defining the same command for two different purposes, a self-explanatory error message is issued. You must correct the error before terminating the installation. If a primary com-

mand conflicts with one of the WordStar- compatible secondary commands, it will render the secondary command inaccessible.

Table B-1 lists the secondary commands, and allows you to enter any primary commands you have installed.

Note: Items 2 and 28 let you define alternative commands to *Character Left* and *Delete Left Character* commands. Normally, the Back Spacec key is the alternative to ⌐J⌐ ⌐S⌐ , while there is no defined alternative to ⌐Del⌐ . You may install primary commands to suit your keyboard; for example, to use the Back ⌐ ⌐e as an alternative to ⌐Del⌐ if the Back Space key is more conveniently located. Of course, the two alternative commands must be unambiguous like all other commands.

Table B-1 Turbo **Modula-2 Secondary Commands**

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
'2:	To left on line	Ctrl	Q	S
₁3:	To right on line	Ctrl	Q	D
14:	To top of page	Ctrl	Q	E
15:	To bottom of page	Ctrl	Q	X
16:	To top of file	Ctrl	Q	R
17:	To end of file	Ctrl	Q	C
18:	To beginning of **block**	Ctrl	Q	B
19:	To end of block	Ctrl	Q	K
20:	To last cursor position	Ctrl	Q	P

APPENDIX C
Summary of Compiler Directives

This appendix contains a list of available compiler directives and a brief description of each. For a detailed explanation of compiler directives, refer to Chapter 10, »System Operations.«

There are two ways to set the current compiler options. The first is with the global switches in the Options submenu in the Turbo Modula-2 shell. These global switches have an effect over an entire program text. However, the second method, embedded switches, overrides the first method. Embedded switches appear directly in the program text within comments. In the following list of compiler options, the switch settings on the left are embedded and the ones on the right are global (as they appear in the options menu).

List Source Option

Embedded Global

| $L+ | List(ON) | Generates a listing |
| $L- | List(OFF) | Generates none |

Overflow Check Option (INTEGER and CARDINAL)

Embedded Global

| $O+ | Overflow(ON) | Checks for overflows |
| $O- | Overflow(OFF) | Does not check for overflow |

Test Range Option

Embedded Global

| $T+ | Test(ON) | Checks array indices and subrange variables |
| $T- | Test(OFF) | Does not check them |

The **Upper**=**Lower Case** Sensitivity Option

Embedded Global

| $U+ | Upper=lower(ON) | Does not distinguish between uppercase and lowercase |
| $U- | Upper=lower(OFF) | Distinguishes between type case |

Extension Option

Embedded Global

| $X+ | eXtensions(ON) | Allows Turbo Modula-2 extensions |
| $X- | eXtensions(OFF) | Flags extension as nonstandard |

Native Code Option

Embedded Global

| N/A | Native(ON) | Generates native code for the Z80 |
| N/A | Native(OFF) | Generates M-code |

Note: Embedded native code switches have no effect.

APPENDIX D
Error Diagnosis

This section lists most messages that Turbo Modula-2 issues for execution errors. Usually, an error message is caused by a faulty program; however, some messages can also result from illegal input to an otherwise correctly executing program.

Format of a Runtime Error Message

While a program is executing, the Turbo Modula-2 system checks for a number of runtime error conditions. The various error sources are listed in the next two sections.

An error message can be issued by the runtime system, as well as by some support module or application program. The process is the same in every case: The executing program is stopped and a message of the following form is given:

```
(Name of Error) in module (Name of Module)
(possibly a further explaining message)
Press "C" for calling chain >
```

For example, in module *BadStuff* the value of an expression computes to 11 where the maximum allowed is 10. The following message is then given:

```
BoundsError in module BADSTUFF
0 to 10 is legal range, but 11 was evaluated
Press "C" for calling chain >
```

In another example, an attempt was made to read data from file IN.DAT when the end of that file had already been reached. This error is not detected by the interpreter, but rather by the module *Files*.

```
EndError in module FILES
While processing file B:IN.DAT
Press "C" for calling chain >
```

In Turbo Modula-2, an error constitutes an *exception*. Exceptions can be issued by the Turbo Modula-2 interpreter and by any Modula-2 program unit. They can be trapped in an application program with an exception handler. The raising and handling of an exception is explained further in Chapter 9, »Turbo Modula-2 Extensions.«

Errors Detected by the Interpreter

Errors detected by the interpreter are often caused by faulty program logic, and with some of these errors there is no reasonable way to continue a program. Therefore, only part of the following errors can be caught explicitly (by name) in an exception handler. These exceptions are the ones exported by the pseudomodule *SYSTEM*: OVERFLOW, REALOVERFLOW, and OUTOFMEMORY. The remainder can only be caught with the **ELSE** clause of the **EXCEPTION** statement: BoundsError, DivisionByZero, StringTooLong, FunctionReturnsNoResult, EndOfCoroutine, CaseSelectError, PointerError, and IllegalInstruction.

BoundsError. The value of an arithmetic expression lies outside of its admissable bounds. This can happen when

☐ A value is assigned to a variable of some subrange type, some user- defined scalar type, or one of the types CHAR, BOOLEAN, CARDINAL, or INTEGER. The value lies outside of the bounds defined by the type used.

☐ A function procedure returns a value exceeding the bounds defined by the type of the function.

☐ An array index lies outside of the admissable bounds.

A check for this error can be suppressed by the compiler option (* $T- *). This leads to somewhat shorter object modules and slightly faster execution times. As a rule, however, the efficiency gains do not make up for the resulting safety loss; therefore, bounds-checking is turned on by default.

Additional messages: *(number)* to *(number)* is legal range, but *(number)* evaluated. Indicates the maximum or minimum value allowed, as well as the computed value. This can occur if a negative INTEGER value is assigned to a CARDINAL, or if a CARDINAL above 32767 (the maximum INTEGER value) is

assigned to an INTEGER. The system does not know whether the left-hand side of the assignment is of type INTEGER or CARDINAL; therefore, both possibilities are displayed. For example:

```
65535 is assigned to an INTEGER
```

or

```
-1 is assigned to a CARDINAL
```

Note that the same internal bit value denotes 65535 when interpreted as a CAR-ṆNAL and -1 when regarded as an INTEGER.

DivisionByZero. An attempt has been made to divide by 0.

Additional message: None

Overflow. The result of some computation involving CARDINALs, INTEGERs, or LONGINTs that became too large or too small to be represented in the computer's memory. If the truncated value does not fit into the range of the destination type, an OVERFLOW can also result from a truncation function such as *INT,* *CARD, LONG,* or *TRUNC.* The ranges of the three types in question are CARDINAL, 0 to 65535; INTEGER, -32768 to 32767; and LONGINT, -2147483648 to 2147483647.

For operations involving INTEGER and CARDINAL arithmetic and CARDINAL multiplication, overflow checking can be suppressed by turning the compiler option $O off.

Additional message: None

RealOverflow. A real value has been computed that exceeds the admissable range for REALs (-10 ^ 38 to 10 ^ 38).

Additional message: None

StringTooLong. A string expression has been assigned to an array with elements of type CHAR. The string is too long to be held in its full length in the array. The string is not truncated to fit; instead, StringTooLarge is raised.

Additional message: None

FunctionReturnsNoResult. A function procedure reaches its end before executing a RETURN statement.

Additional message: None

EndOfCoroutine. A coroutine reaches its end before being left by a *TRANSFER* or *IOTRANSFER* statement.

Additional message: None

OUTOFMEMORY. The program has run out of main memory.

Additional message: *Stack* = (*number*), *Heap* = (*number*). Storage is occupied from two sides: Stack and Heap mark the boundaries of the upper and lower occupied part of memory, respectively. The out-of-memory condition arises when Stack and Heap meet.

CaseSelectError. A CASE statement without an ELSE clause has executed where no case alternatives apply.

Additional message: None.

PointerError. Dereferencing of some pointer with the value *NIL* has been attempted. For example, given the declaration VAR *cp*: POINTER TO RECORD *x,y*: REAL END; and the program fragment *cp:=NIL*; IF *cp* ^ .*x* > *0.0* THEN ..., the exception PointerError is raised upon execution of the last statement.

Additional message: None

IllegalInstruction. The interpreter finds a code byte that does not represent a legal command. This is a rare but severe error. It can occur if some parts of code have been overwritten by the executing program; for example, if the user erroneously addresses computations.

Additional message: (1) *Absolute* PC = (*number*), and (2) prints the value of the instruction counter.

Errors Detected by Support Modules

This section concentrates on exceptions that result from presenting illegal input or exceeding the computer's limits. All exceptions signaling a programming error are discussed in their respective modules description in Chapter 11, »The Standard Library.«

The modules in the Turbo Modula-2 Library recognize some error conditions. An error issued by a library module can be caught in an exception handler, provided the exception is imported from the library module.

Exceptions Issued by Module Files

UseError. This usually occurs if writing to a write-protected disk has been attempted. This exception is most often caused by a disk swap. You can solve the situation by resetting the disk before trying again.

Additional message: *Drive* (*Drive-Code*) is read-only.

DiskFull. Writing to disk is impossible, presumably because the disk or disk directory is full.

Additional message: While processing file, (*Name*): indicates the name of the file involved in the write operation.

EndError. An attempt has been made to read data after the end of a file has been reached. This usually constitutes a programming error.

Additional message: While processing file, (*Name*): indicates the name of the file involved in the read operation.

DeviceError. The disk cannot be read correctly. This is equivalent to the dreaded BDOS ERROR/BAD-SECTOR message of CP/M.

Exceptions Issued by Module Loader

LoadError. An error has occurred during the loading of a program. The cause of the error is given in an additional message.

Additional messages: (1) File not found : (*File Name*); (2) Read Error : (*File Name*); (3) Out of memory; (4) Version Conflict : (*File Name*).

The first three messages are self-explanatory. The fourth, version conflict, results when the version of an object module does not coincide with the version required by an importing module. (For a more detailed discussion, refer to the section about *Loader* in Chapter 11.)

The Calling Chain

If a program exhibits a runtime error, the Turbo Modula-2 system offers a powerful diagnosis. Turbo Modula-2 helps to localize the error and gives some clues about the state of the program at the time of the error's occurrence. This information is contained in the calling chain, which displays all active procedures at the point where they were called. The chain is displayed on the screen if you press [*C*] after the following prompt:

```
Press "C" for calling chain >
```

This appears next to the actual error message. If you do not want a calling chain, simply press ⌑ or any letter except C. This will return you to the main menu.

The calling chain consists of one or several lines of the following form:

```
(Name of Module) (Name of Procedure) (Offset Number) (Program
Counter)
```

The first line states the module, the procedure, the offset number, and the program counter where the error occurred. Note that only the first six characters of the module and procedure are displayed. The left column specifies the module enclosing the procedure in which the error occurred. The second column shows the procedure where the error occurred. The offset number corresponds to the numbers produced in the compiler listing. It specifies the offset (in bytes) of the error point from the beginning of the enclosing procedure or module, whose name is listed in the second column. The program counter is used in the Options menu to find where the error occurred in the source file.

The second line of the calling chain specifies the point where the procedure in the first line was called. The same specification holds for the lines that follow.

The chain finishes when the main program is reached. At that point, the name of the main program is in both the module and the procedure column. The calling chain also terminates if a procedure representing a coroutine is reached. Since coroutines are not called by any other part of the program, they are considered the same as main programs.

If you trace the calling chain from bottom to top, you will experience the same sequence of situations that your program went through before the point of the run-time error.

A calling chain, caused by some hypothetical program, is shown in the following example:

```
BoundsError in module BADSTUFF
0 to 10 is legal range, but 11 was evaluated
Press "C" for calling chain > C

Module      Procedure   offset      PC
BADSTUFF    SNEAKY      17          101
BADSTUFF    SUB         113         123
TEST        Q           55          154
TEST        TEST        20          164
>
```

We can follow the events by starting at the bottom with the main program *Test*. *Test* executes until it reaches the offset of 20 in the main program, then it calls the procedure *Q*. The call to procedure *Q* occurs when the program counter is equal to 164.

Looking at the next line we see that procedure *Q* exists in the module *Test*. This procedure then executes until it reaches an offset of 55 from its beginning. At this point the program counter is 154, which is pointing at a call to the procedure *Sub*.

The next line shows that we have called the procedure *Sub* in module *BadStuff*. The procedure *Sub* executes until it calls *Sneaky*. When *Sneaky* is called, the program counter is 123.

The top line of the calling chain shows that the error occurred in the procedure *Sneaky* in module *BadStuff*. The offset of the error in *Sneaky* is 17. The program counter where the error occurred is 101. We know from the error message that there was some type of bounds error; so we can look at that line for assignment or indexing problems.

Finding RunTime Errors

To find where this error occurred in the source code without having to guess with offsets into the procedures or look at a listing, you can use the program counter and the Find RunTime Error option found in the Options menu.

At the main menu, press [O] to enter the Options menu. Then press [F] for Find RunTime Error. You should know the program counter and the name of the module in which the error occurred. First you are prompted with the name of the main module that was running. If this is not the module where the error occurred, then backspace over it and enter the correct name. If it is the correct module name, then just press [↵] .

Next you are prompted to enter the PC. At this point you should enter the number you obtained from the calling chain in the program counter column. When the compiler has compiled to this point, you will be thrown into the editor with the cursor at the corresponding runtime error position. This utility is helpful for finding errors quickly and reducing program development time.

Compiler Error Messages

Error messages sent by the compiler can be synthesized or fixed. The »fixed« messages are read from a file named ERRMSGS.TXT, which may or may not be online during compilation. If it is not, only the error number, not the corresponding message, can be displayed. Table D-1 provides all of the messages contained in ERRMSGS.TXT.

Some messages will not be found in this list. **For example:**

```
":=" expected, but "=" found
```

is not in the list because it is synthesized from parts of the faulty program together with predefined pieces of text. Other messages are contained only partial

ly by ERRMSGS.TXT; the rest are supplied from the faulty program. Items supplied by the compiler appear in parentheses.

Table D-1 Messages in ERRMSGS.TXT

Error in Identifier

  0 *(Identifier)* is undeclared.
  1 *(Identifier)* is declared twice.
  2 *(Identifier)* is not field of this record.
  3 *(Identifier)* is not exported by this module.
  4 *(Identifier)* already exists outside of module.
  5 Unresolved **FORWARD** reference: (Identifier).
  6 Unresolved export: (Identifier).
  7 *(Identifier)* is not readable.
  8 *(Identifier)* is not printable.
  9 *(Identifier)* expected.
10 *(File name)* not found.
11 CODE must be imported from *SYSTEM*.
12 Two different versions of (Module name).SYM imported.
13 No standard procedure allowed here.
14 Procedure must be declared at outer-most level.
15 *(Identifier)* is exported twice.

Error in Syntax

20 Illegal key word at start of statement.
21 Illegal start of statement.
22 Identifier, literal, or »(« expected.
23 (Not used).
24 Loop counter may not be external or parameter.
25 Expression must have constant value.
26 No enclosing **LOOP** for **EXIT**.
27 No **RETURN** from module allowed.
28 String literal spans over several lines.
29 Past end of file.
30 Badly formed number.
31 Illegal symbol.
32 End of file expected.
33 String literal expected.

34 CODE must follow procedure heading; it is not allowed here.

General Errors

41 Set elements may only range from 0 to 15.
42 (Not used).
43 Modules with unqualified export list may not be **exported**.
44 Module does not export in QUALIFIED mode.
45 Length of actual string does not match.
46 Second value must be greater.
47 Case label occurs twice.
50 String assignments not allowed in standard Modula-2.
51 String comparisons not allowed in standard Modula-2.
52 Not allowed in standard Modula-2.
53 Procedure ALLOCATE not found.
54 Procedure DEALLOCATE not found.
55 Illegal definition of procedure ALLOCATE.
56 Illegal definition of procedure DEALLOCATE.

Error in Type

60 Function procedure required.
61 No function procedure allowed here.
62 Actual parameter is byte-packed, but **Formal VAR parameter assigns a** word.
63 No such variant exists.
64 Element types of actual and formal arrays differ.
65 Types of actual and formal parameters differ.
66 Sizes of type and argument differ.
67 Declarations of procedure variable and procedure differ.
68 Two different declarations of same procedure.

Error in Constant

70 Integer required, but large cardinal value computed.
71 Cardinal required, but negative value computed.
72 Value lies outside of subrange bounds.
73 Number too large.
74 Real number too large.
75 Overflow in constant **expression.**

Error/Compiler Limit Exceeded

80 Case label must not be greater than 32767.
81 Too many local variables.
82 Too many procedures, strings, and exceptions.
84 Too many imported modules.
85 Insufficient space for import/export: name-table overflow.
86 Insufficient space for import/export: type-table overflow.
87 Insufficient space for import/export: identifier-table overflow.
88 Insufficient space for import/export.
90 Expression too complex or too many parameters.
91 Boolean expression too long.
92 Too many nested function calls.

Error Implementation Restriction

95 Byte-sized array elements cannot be substituted for VAR parameters.
96 Byte-sized array elements cannot be used as arguments of *INC* or *DEC*.

Warning

50 String assignments not allowed in standard Modula-2.
51 String comparisons not allowed in standard Modula-2.
52 & not allowed in standard Modula-2.

APPENDIX E
BNF Syntax for Turbo Modula-2

The syntax of the Turbo Modula-2 language is presented here using the formalism known as Backus-Naur Form (BNF). The following symbols are meta symbols belonging to the BNF formalism; they are not symbols of the language.

< term >	Names of language constructs are surrounded by "<" and ">".
{ X }*	Represents zero or more repetitions of X.
[X]	Means X is optional.
X	Means X is mandatory.
X | Y	Indicates that X and Y are alternatives and that either X or Y must be used.
"X" or 'X'	Means X is written exactly as shown.
(X)	Means X must be chosen.

All other symbols are part of the language (reserved words of Turbo Modula-2 are in **boldface** type). For easy reference, the syntactic constructs are listed alphabetically.

< ActualParameters > ::= "(" [< ExpList >] ")"

< AddOperator > ::= "+" | "-" | "OR"

< ArrayType > ::= "ARRAY" < SimpleType > {"," < SimpleType >}* "OF"
< type >

< assignment > ::= < designator > ":=" < expression >

< block > ::= { < declaration > }* ["BEGIN" < Statement**Sequence** >]
        [< ExceptionHandler >] "END"

< **case** > ::= [< CaseLabelList > ":" < StatementSequence >]

```
<CaseLabelList> ::= <CaseLabels> {"," <CaseLabels>}*

<CaseLabels> ::= <ConstExpression> [ ".." <ConstExpression> ]

<CaseStatement> :: = "CASE" <expression> "OF" <case>
                     { "|" [ <case> ] }
                     [ "ELSE" <StatementSequence> ]
                       "END"

.<character> ::= <letter> | <digit> | " " | "!" | "'" | "#" |
                 "$" | "%" | "&" | "'" | "(" | ")" | "*" | "+" |
                 "," | "-" | "." | "/" | ":" | ";" | "<" | "=" |
                 ">" | "?" | "@" | "[" | "\" | "]" | "^" | "_" |
                 """ | "{" | "|" | "}" | "~"

<CompilationUnit> ::= <DefinitionModule> | ["IMPLEMENTATION"]
                      <ProgramModule>

<ConstantDeclaration> ::= <ident> "=" <ConstExpression>

<ConstExpression> ::= <expression>

<declaration> ::= "CONST" { <ConstantDeclaration>   ";" }* |
                  "TYPE"  { <TypeDeclaration>        ";" }* |
                  "VAR "  { <VariableDeclaration>    ";" }* |
                  <ExceptionDeclaration>             ";" |
                  <ProcedureDeclaration>             ";" |
                  <ModuleDeclaration>                ";"

<definition> ::= "CONST" { <ConstantDeclaration>    ";" }* |
                 "TYPE"  { <ident> ["="<type>]       ";" }* |
                 "VAR "  { <VariableDeclaration>     ";" }* |
                 <ExceptionDeclaraton>               ";" |
                 <ProcedureHeading>                  ";"

<DefinitionModule> ::= "DEFINITION MODULE" <ident> ";"
                       { <import> }* { <definition> }* "END"
                       <ident> "."
```

```
<designator> ::= qualident { "." <ident>|
"["ExpList"]"| " ∧ "}*

<digit> ::= <octalDigit> | "8" | "9"

<element> ::= <expression> [ ".." <expression> ]

<enumeration> ::= "(" <IdentList> ")"

<exception> ::= [ <IdentList> ":" <StatementSequence> ]

<export> ::= "EXPORT" [ "QUALIFIED" ] <IdentList> ";"

<expression> ::= <SimpleExpression> { <relation>
<SimpleExpression> }*

<ExceptionDeclaration>::="EXCEPTION"<ident> {","  <ident>}*

<ExceptionHandler> ::="EXCEPTION" <exception> { "|"
                       <exception> }*
                      [ "ELSE" <StatementSequence> ]

<ExpList> ::= <expression> { "," <expression> }*

<factor> ::=<number> | <string> | <set> | <designator>
[<ActualParamters>] |
            "(" <expresion> ")" | "NOT" <factor>

<FieldList> ::=[ <IdentList> ":" <type> |
                "CASE" [<ident>] ":" <qualident>
                "OF" <variant> {"|" <variant>}*
                [ "ELSE" <FieldListSequence> ] "END" ]

<FieldListSequence> ::= <FieldList> {";" <FieldList>}*

<ForStatement> ::="FOR" <ident> ":=" <expression> "TO"
                  <expression>
                  [ "BY" <ConstExpression> ]
                  "DO" <StatementSequence> "END"
```

```
<FormalParamenters> ::= "(" [ <FPSection>
                              { ";" <FPSection> }*]")"
                              [ ":" <qualident> ]


<FormalType> ::= [ "ARRAY OF" <qualident> ]


<FormalTypeList> ::="(" [ ["VAR"] <FormalType>
                        {"," [VAR"] <FormalType>}* ] ")" [ ":"
                        <qualident> ]


<FPSection> ::= [ "VAR" ] <IdentList> ":" <FormalType>


<hexDigit> ::= <digit> | "A" | "B" | "C" | "D" | "E" | "F"


<ident> ::= letter {letter | <digit>}*


<IdentList> ::= <ident> {"," <ident>}*


<IfStatement> ::="IF" <expression> "THEN"
                    <StatementSequence>
                    {"ELSIF" <expression> "THEN"
                    <StatementSequence> }*
                    ["ELSE <StatementSequence> ]
                    "END"


<import> ::= [ "FROM" <ident> ] "IMPORT" <IdentList> ";"


<InlineCode> ::= "CODE" "(" <string> ")" "END"


<integer> ::=<digit> {<digit>}* |
             {<octalDigit>}* ( "B" | "C" ) |
             <digit> {<hexDigit>}*


<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"
```

<LoopStatement> ::= "LOOP" <StatementSequence> "END"

<ModuleDeclaration> ::="MODULE" <ident> [<priority>] ";"
            { <import> } [<export>] <block>
            <ident>

<MulOperator> ::= "*" | "/" | "DIV" | "MOD" | "AND"

<number> ::= <integer> | <real>

<octalDigit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7"

<PointerType> ::= "POINTER TO" <type>

<priority> ::= "[" <ConstExpression> "]"

<ProcedureCall> ::= <designator> [<ActualParameters>]

<ProcedureDeclaration> ::= <ProcedureHeading> ";"
            (<block> | <InlineCode>)
            <ident>

<ProcedureHeading> ::= "PROCEDURE" <ident>
[<FormalParameters>]

<ProcedureType> ::= "PROCEDURE" [FormalTypeList]

<ProgramModule> ::="MODULE" <ident> [<priority>] ";"
            { <import> }* <block> <ident> "."

<qualident> ::= <ident> {"." <ident>}*

<RaiseStatement> ::= "RAISE" [<ident> ["," <expression>]]

<real> ::= <digit> {<digit>}* "." {<digit>}* [ScaleFactor]

<RecordType> ::= "RECORD" <FieldListSequence> "END"

<relation> ::= "=" | ">" | "<" | ">=" | "<=" | "#" | "<>"
| "IN"

<RepeatStatement> ::= "REPEAT <StatementSequence> "UNTIL"
<expression>

<ScaleFactor> ::= ("E" | "D") ["+" | "-"] <digit>
{<digit>}*

<set> ::= [<qualident>] "{" [<element> { ","
<element> }*] "}"

<SetType> ::= "SET OF" <SimpleType>

<SimpleExpression> ::= ["+" | "-"] <term> { <AddOperator>
<term> }*

<SimpleType> ::= <qualident> | <enumeration> |
<SubrangeType>

<statement> ::=[assignment | ProcedureCall | IfStatement |
                 CaseStatement | WhileStatement | RepeatStatement |
                 LoopStatement | ForStatement | WithStatement |
                 RaiseStatement | "EXIT" | "RETURN" [expression]]

<StatementSequence> ::= <statement> { ";" <statement> }*

<string> ::= "'" {<character>}* "'" | '"' {<character>}* '"'

<SubrangeType> ::=[<qualident>] "[" <ConstExpression> ".."
                   <ConstExpression> "]"

<term> ::= <factor> { <MulOperator> <factor> }*

<type> ::=<SimpleType> | <ArrayType> | <RecordType> |
          <SetType> | <PointerType> | <ProcedureType>

<TypeDeclaration> ::= <ident> "=" <type>

```
<VariableDeclaration>::= <ident>["[" <ConstExpression> "]"]
                        { "," <ident> [ "[" <ConstExpression>
                        "]" ] }*
                        ":" <type>

<variant> ::= [ <CaseLabelList> ":" <FieldListSequence> ]

<WhileStatement> ::="WHILE" <expression> "DO"
                    <StatementSequence>
                    "END"

<WithStatement> ::="WITH" <designator> "DO"
                    <StatementSequence>
                    "END"
```

Index

Reflex

Reflex ist mehr als nur die außergewöhn-
liche Kombination aus Datenbank und
Tabellenkalkulation. Reflex ist Daten-
analyse. Der direkte Weg zu den ver-
steckten Wahrheiten hinter reiner
Information.

In der Verbindung aus grafischen
Eigenschaften, übersichtlicher Fenster-
(hnik, hoher Programmgeschwindigkeit
.. .ɹ einem vollständigen Satz von Funk-
tionen schafft Reflex eine Transparenz,
die Ihnen bisher kein anderes Datenbank-
programm geboten hat.

Reflex macht schnell

Da Reflex's Daten im Hauptspeicher
gehalten werden, sind die meisten Opera-
tionen wie sortieren, filtern etc. sehr
schnell. Durch die Above-Board-Unter-
stützung kann Reflex aber auch mit größe-
ren Datenmengen umgehen, ohne Sie auf
die Folter zu spannen.

Reflex kennt fünf verschiedene
Ansichten: Die Listenansicht, die Formu-
laransicht, die Kreuztabelle, die Grafik-
und die Berichtsansicht. Während Sie in
der Listenansicht alle Daten auf einen
Blick sehen und erfassen, läßt sich der
vollständige Datensatz im Formular dar-
stellen. Was Sie gerade in der einen
(ısicht bearbeiten, sehen Sie sofort auch
als Grafik in der Grafikansicht - als
Balken- Kreis- oder Liniendiagramm.

Reflex kann mehr

Mit Hilfe der Kreuztabelle bekommen Sie
ein Instrument, von dem viele Spread-
sheet-Eigner träumen: Denn hier können
Sie verschiedene Datensätze und Daten-
felder miteinander in Beziehung setzen,
und zwar so lange, bis Sie nur das sehen,
worauf es Ihnen ankommt.

Reflex ist durch seine grafische Benut-
zeroberläche leicht zu beherrschen.
Pull-down-Menüs und Dialogboxen
führen Sie sicher durch die Vielfalt von
Funktionen. Ihren ersten Bericht oder Ihre
erste Grafik erzeugen Sie schon nach den
ersten 15 Minuten.

Mit unzähligen Funktionen aus den
Bereichen Betriebswirtschaft, Finanzen
Statistik und Mathematik lösen Sie die
meisten kalkulatorischen Probleme. Diese
Kombination finden Sie bei kaum einer
anderen Datenbank.

Technische Daten

■ 250 Felder pro Datensatz
■ 254 Zeichen pro Datenfeld
■ 64.000 Datensätze pro Datenbank
■ Feldformate: numerisch, Ganzzahl,
Datum oder Text
■ suchen und filtern in den verschieden-
sten boolschen Verknüpfungen.
■ automatisch berechnete Felder
■ Logik- und Datumsfunktionen,
■ VARY-Funktion generiert Datensätze
nach bestimmten Verlaufsvorgaben
■ Fremdatenübernahme aus Lotus 1-2-3,
dBase II, DIF (Multiplan) und ASCII

Systemvoraussetzungen

IBM PC bzw. kompatible,
oder Siemens PCD
mindestens 384 KByte Hauptspeicher,
Farbgrafikadapter, Herculeskarte oder
EGA-Karte

HEIMSOETH & BORLAND