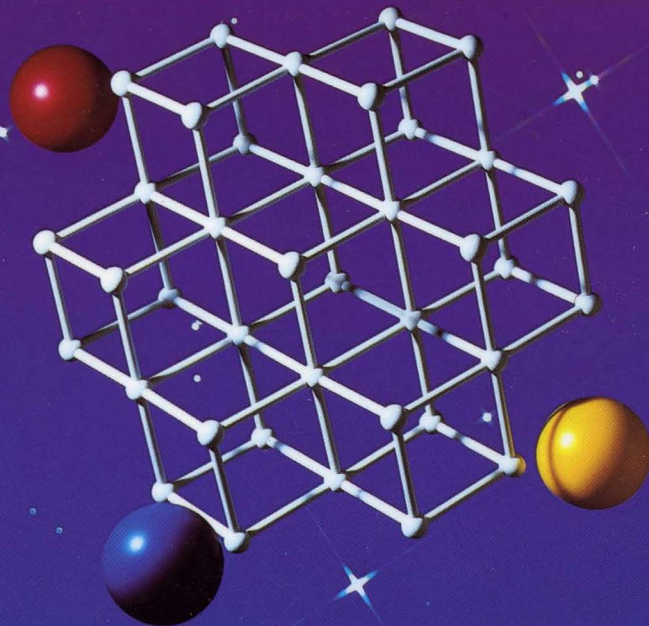


LOGITECH™  
MODULA-2  
VERSION 3.0



TURBO PASCAL  
TO LOGITECH™  
MODULA-2  
TRANSLATOR

**LOGITECH**

**SOFTWARE ENGINEERING LIBRARY**

**PASCAL TO MODULA-2 TRANSLATOR  
USER'S MANUAL**

First Edition      May 1986

Copyright (C) 1984, 1985, 1986, 1987 LOGITECH, Inc.

All Rights Reserved. No part of this document may be copied or reproduced in any form or by any means without the prior written consent of LOGITECH, Inc.

*LOGITECH, MODULA-2/86, and MODULA-2/VX86 are trademarks of LOGITECH, Inc.*

*Microsoft is a registered trademark of Microsoft Corporation. MS-DOS is a trademark of Microsoft Corporation.*

*Intel is a registered trademark of Intel Corporation.*

*IBM is a registered trademark of International Business Machines Corporation.*

*Turbo Pascal is a registered trademark of Borland International, Inc.*

LOGITECH, Inc. makes no warranties with respect to this documentation and disclaims any implied warranties of merchantability and fitness for a particular purpose. The information in this document is subject to change without notice. LOGITECH, Inc. assumes no responsibility for any errors that may appear in this document.

From time to time changes may occur in the filenames and in the files actually included on the distribution disks. LOGITECH, Inc. makes no warranties that such files or facilities as mentioned in this documentation exist on the distribution disks or as part of the materials distributed.

LU-GU110-1

Initial issue: May 1986

Reprinted: September 1987

This edition applies to Release 1.00 or later of the software.

**LOGITECH'S POLICIES AND SERVICES**

Congratulations on the purchase of your LOGITECH Pascal To Modula-2 Translator. Please refer to the following information for details about LOGITECH's policies and services.

We feel that effective communication with our customers is the key to quality service. Therefore we have designed a bulletin board, the LOGITECH MODULA-2 Information Service, so you can contact us directly and conveniently. You can contact the LOGITECH MIS simply by dialing

**(415) 795-0408**

using a 1200 or 300 baud modem. You log in by typing `modula2`. The menu with available options is self explanatory and allows you to:

- order a MODULA-2 product
- report a bug
- access MODULA-2 source files for downloading
- read about recent LOGITECH developments and other interesting information

If you are an Independent Software Vendor, we encourage you to join the impressive list of developers who have used LOGITECH MODULA-2 products to design their own applications. For all LOGITECH MODULA-2 users, including ISVs, we have formed a LOGITECH MODULA-2 User Group. LOGIMUG publishes a newsletter and provides a forum through which LOGITECH MODULA-2 users can exchange ideas and information.

LOGITECH is committed to customer support. Whether you are an individual or part of a large organization, we offer a support plan designed to meet your needs.

The Modula-2 User Association is another important source of information about the Modula-2 language, as well as a forum for Modula-2 users to exchange ideas and to share pertinent technical tips. LOGITECH is an active corporate member of this association. We encourage you to contact MODUS at:

**MODUS  
P.O BOX 51778  
PALO ALTO, CALIFORNIA 94303  
(415) 322-0547**

For those of you who would like to keep up with current MODULA-2/86 developments and communicate electronically with us, LOGITECH sponsors a conference on the BYTE Information Exchange (BIX). BIX is an electronic conferencing system that allows you to communicate with other MODULA-2/86 users about technical problems, language issues, third-party software developments, and any other topics that interest you. In addition to the LOGITECH conference, BIX offers conferences on a vast range of other subjects including computers, operating systems, applications, chips, AI, and up-to-date information from industry leaders on new technology. To join BIX contact:

**BYTE INFORMATION EXCHANGE  
ONE PHOENIX MILL LANE  
PETERBOROUGH, NH 03458  
(603) 924-9281**

We look forward to hearing from you on BIX!

The following syntactic conventions are used in this manual:

- Input a user must type on the keyboard looks like this:

A><u>m2 pas2mod <CR>

- Output the program displays on the screen looks like this:

Pascal in>

- Program source code looks like this:

```
VAR
  a   : INTEGER;
  str : ARRAY [0..10] OF CHAR;
BEGIN
  ...
```

- Special keys such as 'escape' and 'carriage return' are abbreviated and enclosed in brackets, for example:

<ESC>, <CR>.


- Control characters, characters entered while the key marked 'ctrl' is depressed, are preceded by 'Ctrl' and enclosed in brackets, for example:

<Ctrl-C>, <Ctrl-Break>.

---

## Errata

**LOGITECH Modula-2, Version 3.0** supercedes all previous versions of *Modula-2* from LOGITECH, Inc., including *LOGITECH Modula-2/86*.

Since *LOGITECH Modula-2* now provides .OBJ files at compilation time, all references of the file extension .LNK should be replaced with .OBJ. And since .EXE and .OVL files are produced directly through a standard DOS linker, or through the *LOGITECH Linker*, all references to .LOD should be replaced with either .EXE or .OVL. At the same time it is no longer necessary to load the **M2** prefix to run the translator; you simply type **PAS2MOD** .

Also, other files mentioned in conjunction with the **M2** prefix, can now be executed per instructions in the *LOGITECH Modula-2 User's Manual, Version 3.00*.

References to the *MOD Editor*, chiefly in Section 1.4, need to be replaced by *POINT Editor* commands.

<u>Page 24</u>	<b>MOD first.pas</b>	<b>PT first.pas</b>
<u>Page 26</u>	[Alt F9]   r   pas2mod	must be run in a <i>DOS</i> shell from <i>POINT</i>
<u>Page 26</u>	[F3]   First	use <b>open</b> in <i>POINT</i> on FIRST.MOD
<u>Page 29</u>	1) Syntax check [M2]	Use M2ASSIST
	2) Compile [F5]	Use M2ASSIST
	3) Link [F6]	Use M2ASSIST
	4) Run First.MOD [Alt F9   r   first]	Use M2ASSIST

Refer to *LOGITECH Modula-2 Version 3.00* documentation for current *LOGITECH Modula-2* commands.

Large Sets are now allowed in *Version 3.00*

All references to module System should be re-referenced to module RTSMain, which is described in the *LOGITECH Modula-2 User's Manual*.

The NIL pointer has the value **0:FFFF**.

---

## TABLE OF CONTENTS

INTRODUCTION .....	1
1 THE TRANSLATOR.....	6
1.1 How to Translate a Pascal Program .....	6
1.2 Using the Translator .....	7
1.2.1 Installation.....	7
1.2.2 Running The Translator .....	9
1.2.3 Translation Options .....	10
1.3 Translation Rules.....	13
1.3.1 General Translation Rules.....	13
1.3.2 Standard Identifiers an Libraries.....	16
1.3.3 When the Manual Adaptation is Required.....	17
1.3.4 Flags and Error Handling .....	19
1.3.5 Translator's Capacities.....	23
1.4 Now Let's Try to Translate.....	24
2 LANGUAGE FEATURES THAT REQUIRE MANUAL ADAPTATION.....	30
2.1 Label and Goto Statements.....	30
2.2 Constants and Initialized Variables.....	30
2.2.1 Simple Untyped Constants .....	30
2.2.2 Variable Constants .....	32
2.3 Sets.....	42
2.3.1 Large Sets .....	42
2.3.2 Set of Char.....	45
2.3.3 Use of Variables in Set Construction.....	47
2.4 Strings .....	55
2.4.1 Differences Between Pascal and Modula-2.....	55
2.4.2 How Strings are Translated From Pascal to Modula-2 .....	58
2.4.3 String Operator '+' .....	61
2.4.4 String Expression in Modula-2.....	63
2.4.5 Control Characters Used as String Constants .....	65
2.4.6 Turbo Pascal and Modula-2 Standard String Functions.....	66
2.4.7 Functions Returning a String .....	67
2.4.8 Open Arrays.....	67
2.5 User Defined Functions Returning a String.....	69
2.6 Copy, Concat ParamStr Functions and the Flag ?UNDEF .....	72



2.7 Other Data Types.....	76
2.7.1 Integers, Cardinals and Subranges.....	77
2.7.2 Reals.....	80
2.7.3 Bytes.....	82
2.8 WITH and CASE.....	83
2.8.1 WITH.....	83
2.8.2 CASE.....	86
2.9 Absolute Statements and Untyped Variables.....	86
2.10 Inline Machine Code.....	89
2.11 Turbo Pascal Predefined Variables.....	90
2.11.1 User I/O Drivers.....	91
2.11.2 ErrorPtr.....	93
2.11.3 Mem, MemW, Port, PortW.....	100
2.11.4 HeapTop, Mark and Release.....	101
2.12 I/O Operations.....	101
2.13 Bit Manipulation Operators 'AND', 'OR', 'NOT', 'XOR', 'SHR', 'SHL'.....	105
2.14 Functions and Procedures.....	106
2.14.1 Result of Functions.....	106
2.14.2 Exit.....	107
2.14.3 Halt.....	107
2.14.4 Forward Declarations.....	107
2.15 Overlay, Chain and Execute.....	107
2.15.1 Overlay.....	107
2.15.2 Chain and Execute.....	108
<b>3 ADVANCED SOFTWARE ENGINEERING USING MODULA-2.....</b>	<b>109</b>
3.1 Creating Library Modules.....	109
3.1.1 From a Turbo Pascal Program to a Single MODULA-2/86 Module.....	109
3.1.2 How to Create a Modula-2 Library Module.....	118
3.2 Data Abstraction Using Opaque Types.....	131
3.2.1 How to Hide Internal Representations of Data Types.....	131
3.2.2 Space Allocation for Opaque Types.....	134
3.3 Summary.....	135
<b>4 CALLING TURBO PASCAL'S EXTERNAL PROCEDURES FROM MODULA-2.....</b>	<b>137</b>
4.1 The Turbo Pascal Approach.....	137
4.2 The Modula-2 Approach.....	141
4.3 Summary.....	147

APPENDIX A..... 153  
APPENDIX B..... 176  
APPENDIX C..... 179  
APPENDIX D..... 183  
  
INDEX..... 194



## INTRODUCTION

Congratulations on your purchase of the LOGITECH Pascal to Modula-2 Translator!

The valuable time you have invested programming in Turbo Pascal will not go to waste. Awaiting you is an exciting world of modern software engineering brought by the Modula-2 language.

Turbo Pascal is an excellent product that made an important contribution to the world of microcomputer programming. It broke the myth that BASIC was the language of the masses. In addition to the advantages of structured programming of Pascal over BASIC, Turbo Pascal added speed and language extensions which made a versatile Pascal implementation available to numerous programmers. The world of structured programming enjoyed popular recognition in an unprecedented fashion.

### Why Cross Over to LOGITECH MODULA-2/86 ?

There are several reasons why you should make the valuable transition from Turbo Pascal to MODULA-2/86:

- The most significant reason to translate your Turbo Pascal programs to LOGITECH MODULA-2/86 is that Modula-2 implements **modern software engineering techniques** that are not found in Pascal. These include separately compiled modules split into Definition and Implementation parts, data abstraction, open array parameters, concurrency, opaque types, procedures passed as parameters, and a standard set of low level facilities.
- LOGITECH MODULA-2/86 allows the programmer to develop **programs larger than 64K**, a limitation of Turbo Pascal version 3.0 .
- In Pascal, your whole program must be recompiled after every change you make. In Modula-2, **separate compilation** enables you to recompile only what you have changed.

- In Modula-2 you can build your own reusable **library modules** or use the library created by someone else. You do not have to reinvent the same algorithm or rewrite the same procedures defined in the module library. This saves development time and avoids bugs!
- Because Modula-2 is standard, your code is **portable** to any machine.
- LOGITECH MODULA-2/86 offers a complete line of **sophisticated tools** to enhance development. Among these are two powerful debugging tools -- the run-time debugger and the post-mortem debugger. The first allows you to monitor the execution of a program in steps, like in slow motion, to find and correct software errors. The second enables you to efficiently and quickly investigate the cause of a run-time crash. No such tools exist for Turbo Pascal.

### The Power of the LOGITECH Translator

The LOGITECH Translator functions as a bridge between Turbo Pascal and LOGITECH MODULA-2/86 programs. It is designed to aid the programmer in converting a Pascal program to an equivalent Modula-2 program module. The Modula-2 program is pretty printed correctly indented and aligned. The translator makes one pass over the Pascal program being translated and produces the equivalent Modula-2 code.

The Translator assumes that the Pascal program has been compiled and is error free from the viewpoint of the Pascal compiler. If you input an illegal Pascal program, the Translator may produce unpredictable results.

The Translator comes as an executable Modula-2 program, 'pas2mod' and a set of **library modules** that implement all the functions and procedures provided by Turbo Pascal 3.0. The Translator is capable of directly translating Turbo Pascal syntax into Modula-2 syntax, and Turbo Pascal functions and routine calls into a new set of calls to LOGITECH MODULA-2/86 procedures. All the Turbo Pascal 3.0 (PC-DOS/MS-DOS Version) functions are supported, with few exceptions. The following is a list of supported routines:

- Files (text, file of records, untyped files, predefined files).
- Device and screen I/O.
- Arithmetic functions.
- Scalar functions.
- Transfer functions.
- String manipulation routines.

- Heap control routines.
- Screen management routines.
- Basic, advanced and Turtle graphics routines.
- Random routines.
- Delay routines.
- Floating routines.
- Memory Operation routines.
- DOS and Intr routines.
- Reals operations and routines.
- Parameter handling routines.
- Execute routine.
- Number conversion routines.
- Miscellaneous Turbo Pascal routines.

LOGITECH MODULA-2/86 supports Real Arithmetic. Reals libraries are provided in emulation mode, 8087/80287 coprocessor mode, and mixed emulation/coprocessor mode. The LOGITECH MODULA-2/86 compiler has options to generate calls both to the emulator and to the 8087/80287 math coprocessor.

The Translator does not support the Turbo-BCD version and generates no equivalent for the 'Form' function. In MODULA-2/86, the Decimal module performs similar functions as the BCD version. If you want to translate a Turbo-BCD version, study the Decimal module first and modify your program accordingly.

The LOGITECH Translator attempts to completely translate your Pascal programs. However, some Pascal programs after translation require manual adaptation to be correct Modula-2 programs. Please refer to the Chapter Two in this manual for more information on how to modify your program.

### Structure of this Manual

This manual is designed for users with varying levels of technical expertise. Everyone should read the Chapter One before using the Translator. After experimenting with the Translator a little, all users should read Chapter Two. Chapters Three and Four are for those users who want to take advantage of more sophisticated features of the Modula-2 language. The appendices should be used for reference.

- **Chapter One The Translator**  
Gives you a complete description on how to translate a Pascal program and how to use the Translator.
- **Chapter Two Language Features that Require Manual Adatation**  
Describes the post translation techniques you can use to complete your translation.
- **Chapter Three Advanced Software Engineering Using Modula-2**  
Explains how to produce library modules and how to use advanced data structures such as 'Opaque Types'.
- **Chapter Four Calling Turbo Pascal's External Procedures from Modula-2**  
Explains how to modify calls to Turbo Pascal External Procedures.
- **Appendix A Mapping of Turbo Pascal Procedures to Modula-2**  
Lists how Turbo Pascal variables, functions and procedures are mapped into their equivalent in MODULA-2/86.
- **Appendix B Mapping of Turbo Pascal Compiler Directives to Modula-2**  
Lists how Turbo Pascal compiler directives are translated.
- **Appendix C Compatibility Between Turbo Pascal and Modula-2 Data Files**  
Describes the differences in file formats between Turbo Pascal and MODULA-2/86.
- **Appendix D Index Library of Modules**  
Lists all Library Modules and Procedures.

Throughout the manual we use 'Modula-2' when we refer a to a generic Modula-2 language feature and 'MODULA-2/86' when we refer to specific features of LOGITECH's implementation of the language, such as, system dependent data types and functions and library modules.

**Where to Get More Information about Modula-2**

This manual is not intended to teach Modula-2 to Turbo Pascal programmers. The LOGITECH MODULA-2/86 User's Guide includes a Modula-2 tutorial for Pascal programmers as well as a complete bibliography of Modula-2 reference books. In this manual we will concentrate on crossing the bridge from Pascal to MODULA-2/86 using the LOGITECH Translator. Two useful books to be used during the translation are:

Programming in Modula-2, Niklaus Wirth, Springer Verlag, 1982.

Modula-2 for Pascal Programmers, Richard Gleaves, Springer Verlag, 1984.



## 1 THE TRANSLATOR

### 1.1 How to Translate a Pascal Program

You need the following tools to use the Translator:

- PC DOS 2.X, 3.X or MS DOS 2.X, 3.X. (The Translator will run on IBM PC/XT/AT and compatibles)
- A full screen editor such as MOD, the LOGITECH MODULA-2/86 Editor.
- The LOGITECH MODULA-2/86 Base Language System.

The scenario for the use of the Pascal to Modula-2 Translator is as follows:

- 1 Compile the Pascal program 'X.pas' until all errors have been removed.
- 2 Translate 'X.pas' into 'X.mod' by calling 'pas2mod'.  
The translator will attempt to complete a 100% automatic translation, however in some particular cases the user is required to modify the generated Modula-2 code in order to conform with the Modula-2 syntax and LOGITECH libraries.
- 3 Modify all unresolved items in the 'X.mod' file.  
The translator will mark with the flag '?number' all the places with problems to be solved manually by the user. With your editor look for all '?' in 'X.mod'. If there are such problems the translator will produce at the end of 'X.mod' a summary and explanation of all '?number' generated (see section on Translation Rules).
- 4 Compile 'X.mod' and fix all the possible errors detected by the compiler.  
Some errors may not be indicated by the Translator (they are discussed in the next chapters). The Modula-2 compiler will flag any errors which the Translator has not been able to detect.

**Note:** If you run the compiler without any options the code generated contains additional code for Stack tests, Range and Overflow tests, Index and NIL pointer tests. This could make your Modula-2 program run differently from the Turbo Pascal version. To remove this test code see the LOGITECH MODULA-2/86 User's Guide 'Compiler Options'.

- 5 Test the Modula-2 program until it is debugged.

Depending on the type of program, you may experience some run-time errors. We strongly recommend to use both the LOGITECH MODULA-2/86 Run Time Symbolic Debugger (RTD) and the LOGITECH MODULA-2/86 Post Mortem Symbolic Debugger (PMD) for fast trouble-shooting. With these two debugging tools your development time will be reduced up to 50% !!!

- 6 Consider rewriting the resultant 'X.mod' program to use a more sophisticated Modula-2 feature.

Before you start to actually translate your Pascal programs we recommend that you spend some time studying the definition modules (.DEF) of the library distributed with the Translator. In this way you can find out all the powerful procedures available to you.

## 1.2 Using the Translator

### 1.2.1 Installation

The Translator runs on the IBM PC/XT/AT, or any compatible machine. Storage requirements are 256K memory double-sided disk drive, or hard disk. The MS-DOS or PC-DOS version 2.x operating system or higher is required.

The installation of the Translator follows the general guidelines used to install a LOGITECH MODULA-2/86 program (see LOGITECH MODULA-2/86 User's Guide). Remember to make backup copies of all distribution diskettes before starting the installation. The executable 'PAS2MOD.LOD' should go in the subdirectory where all .LOD files are kept (usually \m2lod), while the library modules should go where all the Base Language System library modules are kept, grouped by extension (usually \m2lib\def, \m2lib\sym, \m2lib\lnk, \m2lib\ref). We have included all the sources of sample programs used in this manual. You should load these files in your working directory. Read the file READ.ME for last minute information about the Translator.

NOTE: If you want to generate the .EXE version from PAS2MOD.LOD (it is faster to load), use the LOD2EXE Utility from the Base Language System.

**If you have the 8087/80287 Math Coprocessor**

In the library distribution diskette you will find the following set of .LNK and .REF files with names starting with 'C87':

Module	Real Emulation	Real 8087/80287
Random	RANDOM.yyy	C87RAMDO.yyy
TRealIO	TREALIO.yyy	C87TRIO.yyy
TBinaryIO	TBINARYI.yyy	C87TBINA.yyy
TRealNumberConversion	TREALNUM.yyy	C87TRNUM.yyy
FloatingUtilities	FLOATING.yyy	C87FLOAT.yyy

These files C87xxx.LNK are the versions of the file xxx.LNK compiled with the option to generate code for the Math Coprocessor 8087/80287.

If the machine where you will run your application has a math coprocessor, you can link your application with this version of the .LNK module for enhanced performance of your application at run-time.

In this case, we suggest that you rename the versions xxx.LNK that use the emulator to E87xxx.LNK, and that you rename the version C87xxx.LNK using the Coprocessor to xxx.LNK.

From DOS type:

```
A>REN RANDOM.LNK E87RAMDO.LNK
A>REM RANDOM.REF E87RAMDO.REF
A>REN C87RAMDO.LNK RANDOM.LNK
A>REN C87RAMDO.REF RANDOM.REF
```

For more details see the chapter on Real Arithmetic in the LOGITECH MODULA-2/86 User's Guide.

### 1.2.2 Running the Translator

The LOGITECH Translator is very easy to operate.

To invoke the Translator from DOS type 'm2 pas2mod'. The Translator displays a version number and copyright message, followed by a prompt.

```
LOGITECH MODULA-2/86 Pascal to Modula-2 Translator,
```

```
...
```

```
Type Escape to exit.
```

```
Pascal in >
```

Now, you can enter the filename of the Turbo Pascal program you want to translate, ending with <CR>. The Translator accepts any DOS filename with optional drive id and directory paths. If you do not complete the filename with an extension, the Translator will use the default .PAS for Pascal filenames and .MOD for Modula-2 filenames.

After the Pascal filename you will be prompted for the name of the output Modula-2 file. You can assign any valid DOS file name. If you enter <CR>, the Modula-2 program will take the same name as the Pascal program with the extension .MOD. If the name you type refers to an already existing file, the Translator asks whether you want to overwrite the previous file:

```
Pascal in > <name of Turbo Pascal source program><CR>
```

```
Modula-2 out > <name of output MODULA-2/86 source file><CR> or <CR> only
```

```
overwrite ? (y/n) > Y to overwrite  
                  N to re-enter the filename.
```

Once the translation is complete, the main prompt reappears. At this point you can translate another program. To terminate and exit, type <ESC> twice.

If you prefer, you can call the Translator giving all the parameters on the command line. For example to translate mypas1.pas and mypas2.pas into mymod1.mod and mymod2.mod type:

```
C> m2 pas2mod mypas1 mymod1 mypas2 mymod2 ... <CR>
```

remember to put 'Y' if you want to overwrite existing files:

```
C> m2 pas2mod mypas1 mymod1 y mypas2 mymod2 y ... <CR>
```

The Translator will prompt a '\*' on the screen for each ten lines of Modula-2 code produced. If, for some reason, you need to stop the Translator while it is running, type Ctrl-Break. If you have generated the .EXE version of the Translator, you can invoke it from DOS by typing:

```
C> pas2mod<CR>
```

To invoke the Translator from the MOD Editor, type <Alt F9>, then type r for run, and finally type pas2mod<CR> to run it.

### 1.2.3 Translation Options

The Translator allows you to set some translation options. Rather than respond to the prompt for a Turbo Pascal source program, you can press <ESC>. A menu appears with three entries. Make your selection by typing the character that is capitalized.

```
Pascal in > <ESC>
```

```
Options, Directories or Exit >
```

## ■ Options

If you choose options by typing either an upper or lower case 'O' a submenu appears with five options. Make your selection by typing the character that is capitalized.

Options, Directories or Exit > O

include Paths, Indent, mark Undefined, mark Strings,  
show Options or Exit > ;

### ■ include Paths

Assigns directory path names used by the Translator to search the include files. You can enter one or more path names terminated by '\ ' and separated by ';' (default current path). If you have your include file in the directories '\ work ' and '\ old \ 01 ' enter:

directories for include files>\work;\old\01\ <CR>

The Translator will search for an include file, first in the current directory, and after, in '\work' and finally in '\old\01'.

### ■ Indent

Assigns the indentation factor used by the Translator to produce a well aligned Modula-2 program (default = 2).

### ■ mark Undefined

Enables or disables the Translator to mark all undefined identifiers (default YES). This is useful when you want to know all the identifiers used, but not declared, so that you can fix them before running the compiler.

### ■ mark String

Enables or disables the Translator to mark, each time you use a string (default NO). This is useful when you want to know all the places where strings are used, because you need to modify the algorithms (see section on String).

### ■ show Options

Shows the current option values.

**■ Exit**

Exits to the upper level prompt.

**■ Directories**

If you choose directories by typing either an upper or lower case 'D', a directory prompt is displayed..

```
Options, Directory or Exit > D  
Dir >
```

The Translator accepts any DOS filename with optional drive names, directory paths and wildcards. If you press the <CR> key, you exit to the main prompt. Otherwise, if you type \*.mod you will get the list of all the Modula-2 source files.

```
Dir > *.mod
```

A listing appears showing all the Modula-2 source files in the selected directory.

**■ Exit**

To exit, you can choose either an upper or lower case 'E' or press <ESC>. Exits the Translator back to DOS.

```
Options, Directory or Exit > E
```

```
C>
```

### 1.3 Translation Rules

The general approach to the translation is to accurately translate the Pascal program into an equivalent Modula-2 program module, to flag the questionable or untranslatable constructs, and to help reduce the editing task of converting the flagged forms into the Modula-2 program. The Translator makes a simple one pass sweep over the source input and therefore sometimes does not know the most appropriate translation until after it has parsed a construct. This can generate additional empty lines or place your comments before the original place where they were posted.

#### 1.3.1 General Translation Rules

The Pascal to Modula-2 Translator follows these rules.

- The Pascal program name becomes the Modula-2 module name. If no program name is given in the Pascal program, the Pascal filename is used as module name. To ease your development using LOGITECH MODULA-2/86 tools we suggest that you rename the Modula-2 file using the first 8 characters of the module name. In this way, all tools (linker, debuggers, make, ...) will be able to automatically retrieve your files starting from the module name. (for example, module Memory Operation is in files MEMORYOP.XXX)
- Include files are included and translated. A comment is posted in the Modula-2 program before and after each inclusion. The file 'graph.p' is not actually included, instead the relative IMPORT statement is generated so you can use the graphic routines provided by LOGITECH. You can specify one or more subdirectories where the Translator will search for include files (See the section on Translator Options).
- The Translator preserves comments even if nested, but they may not always appear in the precise position in which they occurred in the Pascal text.
- Compiler directives are translated into LOGITECH MODULA-2/86 compiler directives, and when no corresponding directive is available, into procedure calls (see Appendix B).
- When you use types, variables or procedures from another module, the Translator will automatically generate the required IMPORT list. Because of the automatic nature of the translation, ALL the exported identifiers of that module are imported, regardless of how few are used. This could result in a very long import list. This will not influence the performance of your program, although it will make the translated version less readable. In fact, it does not allow you to know the minimal list of identifiers that need to be imported for use in your module.



- All reserved words and standard identifiers are capitalized. All references to identifiers agree in spelling with their declaration. This is necessary because Modula-2 is sensitive to the use of upper and lower case letters while Pascal is not. Special provisions are made for the use of underscores in Pascal identifiers because many Pascal compilers support them. The underscore is removed and the character following it is forced to be uppercase. This is the Modula-2 style of spelling identifiers.
- Loops, conditional statements and structured statements are correctly generated. FOR statements that use DOWNTO are transformed to BY-1. WITH statements containing a 'with' list are transformed into nested WITH. CASE statements are always generated with the ELSE clause, even if not used in Turbo Pascal. The ELSIF statement is generated when appropriate.
- If a real number is found without a decimal point, then the decimal point is inserted unless it is a constant definition. The exponent indicator is always emitted as an upper case 'E'.
- The Translator uses primes (') to enclose strings unless the prime character is found within the string, then the quote (") character is used to enclose the string and the extra prime is deleted. If both quote and prime are found within the string, the pair of primes are changed to the grave accent (`). The Modula-2 language does not allow both (') and (") to occur in a string, and therefore this compromise is adopted.
- Set brackets are changed to braces and the type name is prefixed to the set expression. If the elements of the set expression are both constant elements and expressions, the translation will not be acceptable to the Modula-2 compiler and it will complain.
- Function definition in Modula-2 is slightly different from Pascal. For each Pascal function, the Translator generates a local Result variable of the same type as the function with the name equal to the function name, plus the word 'Result'. The Translator takes care to generate correct 'RETURN funcNameResult' statements when needed, in particular the final return at the end of the function, to ensure the proper return of values as required by Modula-2.

- LOGITECH MODULA-2/86 does not allow return structured data types from a function. Turbo Pascal allows functions to return Strings, for example the Turbo Pascal functions Copy, Concat and ParamStr return strings. For this reason the implementation of these three functions have been changed in Modula-2 to three procedures with an additional VAR parameter that holds the string to be returned (see the section on Copy, Concat and ParamStr). If your programs define and use other functions that return strings, you will have to change these functions in procedures with the additional parameters, and modify all the function calls into procedure calls.
- The Turbo Pascal procedure Execute is implemented by the Modula-2 procedure Execute (module TExec). The Modula-2 procedure is more powerful than the Turbo Pascal version because it allows you to execute any executable DOS program (.COM or .EXE). For more information on the overlay/subprogram system of Modula-2 and on calling procedures, refer to the LOGITECH MODULA-2/86 User's Guide, and in particular to the module Program and to the chapter on Memory Organization.
- The Translator surrounds the subrange specification 1 .. 10 with brackets [1 .. 10], as required by Modula-2.
- If not already present, empty parentheses are added to parameterless function calls. These are also appended to the heading of function declarations if necessary.
- Forward declarations are correctly handled, moving the headings to the site of the body of the function or procedure.
- The SUCC and PRED built-in functions of Pascal are translated into Modula-2 in a variety of ways.  $X := \text{SUCC}(X)$ , where X is a simple variable naming a scalar, is translated as  $\text{INC}(X)$ . However,  $X := \text{SUCC}(Y)$  is translated as  $X := \text{VAL}(\langle \text{typeName} \rangle, \text{ORD}(Y)+1)$ . However, if Y is a nonsimple name designator, the type of the expression to the left of the SUCC(Y) is used. If SUCC(X) appears first in an expression, its type is not known and  $\text{VAL}(\langle \text{typeName} \rangle, \text{ORD}(Y)+1)$  is used. This may not always be correct but the Modula-2 compiler will be of help.
- Expressions of the form  $\text{SetName} := \text{SetName} + \text{SetElement}$  and  $\text{SetName} := \text{SetName} - \text{SetElement}$ , are translated to INCL and EXCL, respectively.
- INC, DEC are also produced for appropriate assignment statements, otherwise, the ORD forms are produced. For example:  $a:=a+1;$  is translated into  $\text{INC}(a,1);$

- The constants 'pi' and 'maxint' are generated only if used.
- The Translator handles all Turbo Pascal file types (text, file of records, untyped files, predefined files, devices, ...). All files are mapped into the type File (module TKernelIO) while global variables are provided to support predefined files and user written I/O drivers. All Turbo Pascal I/O operations are translated into one or more Modula-2 procedure calls (see Appendix A for the different mapping).
- Reals are supported both in emulation or generating 8087 code (see chapter in the MODULA-2/86 User's Guide on Real Arithmetic).
- I/O and Run Time error messages are supported using the same Turbo Pascal codes. Remember that with LOGITECH MODULA-2/86 you can use two powerful debuggers to trouble-shoot your program. The Run-Time Debugger (RTD) to follow the flow of your program at run-time in a symbolic source level way, and the Post-Mortem Debugger (PMD) to find out in a symbolic source level manner when and why your program crashed.

### 1.3.2 Standard Identifiers and Libraries

Standard identifiers are names known to the compiler that are not reserved words. Because they are known to the compiler they do not have to be declared. They have, in effect, been predefined. The Pascal language and the Modula-2 language have their own sets of standard identifiers. Some of these are duplicated in both languages and some only occur in one or the other language.

The standard identifiers of the Pascal language are read by the Translator. The Translator will find that the standard identifiers have not been declared and the problem of how to translate them arises. There are several possible solutions as outlined below:

- They could be flagged with '?1' as undeclared (a).
- They could be recognized and translated into Modula-2 standard identifiers where a correspondence exists (b).
- They could be mapped into services provided by Modula-2 library modules (c).

All three alternatives are used. Alternatives (a) and (b) are straightforward while alternative (c) is implemented by calling the equivalent Modula-2 procedures supplied with the Translator. To smooth the transition from Pascal to Modula-2, LOGITECH has developed a new set of library modules that implement all the Turbo Pascal functions as defined in the Turbo Pascal 3.0 (PC-DOS / MS-DOS version) manual.

You can find a complete description of the modules and procedures in Appendix D and by looking at the definition modules (.DEF) included in the Translator diskette. Furthermore, in Appendix A of this manual, there is a complete description of the mapping schematic used to translate Turbo procedures into Modula-2 procedures.

Later on, when you have acquired more familiarity with the LOGITECH MODULA-2/86 system, you can switch to modules from the Base Language System library which implement functions similar to the Turbo Pascal functions. For example, for simple I/O you can use module InOut instead of TFiles, TKernelIO, or module Directories instead of TDiskDirectory. Thus, you take advantage of some of the features of these modules and of others available such as RS232, Decimals, Debug, Devices, Dos 3.0 and 3.1, Mouse, NumberConversions, Processes, System and SYSTEM.

### 1.3.3 When the Manual Adaptation is Required

The following is a list of Turbo Pascal features that require your particular attention. If you use one of these features in your Pascal program, edit the Modula-2 program after the translation. With the help of the mapping described in Appendix A, study how the translation has been performed on these cases and apply all the needed modifications as explained in Chapter 2. Depending on the different cases, you will be required to manually translate some of the Pascal code, to add parameters to some routines, or to change some algorithms because of the differences between Pascal and Modula-2. The Translator tries to minimize your manual intervention after the translation, but in these cases it cannot automatically translate and requires your cooperation. Some of these cases are detected by the Translator itself, some by the compiler, and finally some will generate a Run-Time error if not properly fixed. Some of the Pascal features to be adapted are as follows:

- Strings operations, handling and usage
- Initialized variables
- Absolute statements
- Large Sets (> 16 elements)
- Set of Chars
- Goto and Label statements
- Shr, Shl, Xor (bitwise operators)
- And, Or, Not (bitwise operators)
- Execute, Chain, Overlay and OvrPath functions
- Concat, Copy and ParamStr functions
- Reference to variables in Inline code
- Function returning structured data types string

In the following cases the Modula-2 implementation uses routines instead of variables:

- User written I/O drivers
- ErrorPtr and Error handling routines
- Mem, MemW, Port, PortW arrays

### Partially Supported Turbo Pascal Features

All Turbo Pascal functions, procedures and system variables are supported in accordance with the rules described in the following chapters and in the appendices, except:

- **Overlays:**  
There is no need for overlays with LOGITECH MODULA-2/86 because the limit for the size of the code and data is one Megabyte. However, overlays are available with LOGITECH MODULA-2/86, but differently than with Turbo Pascal. The Turbo Pascal statement 'OVERLAY' is transformed in comment by the Translator. See the LOGITECH MODULA-2/86 User's Guide for more details.
- **External Subprogram:**  
External subprograms are supported in a different way than in Turbo Pascal. The Translator marks the external reference. Please refer to the chapter on external calls for a complete example of how to change your program to interface external subprograms. The external subprogram 'graphic', used by 'graph.p', is implemented by different Modula-2 library modules and the translation of the graphic routines is automatically done by the Translator without any additional work required from the user.
- **Turbo-BCD:**  
The Translator does not support the Turbo-BCD version and consequently the 'Form' function has no equivalent generated by the Translator. In LOGITECH MODULA-2/86, the module Decimal performs functions similar to the Turbo-BCD. If you want to translate a Turbo-BCD version, please study the Decimal module first and modify your program accordingly.

#### 1.3.4 Flags and Error Handling

The features of Pascal that do not have an equivalent in Modula-2 or that require manual adaptation from the user appear in the Modula-2 translated code preceded by a problem flag string '?number'. These will have to be modified manually according to the suggestions given in Chapter 2.

The following is a list of possible flags:

- ?0 = system problems
- ?1 = unknown identifier, undefined symbol or null operator
- ?2 = invalid Modula-2 constant string
- ?3 = warning - using string make sure to follow the Modula-2 rules
- ?4 = Pascal statement not supported by Modula-2
- ?5 = function or procedure parameter
- ?6 = absolute variable declaration
- ?7 = external procedure or subprogram declaration
- ?8 = special in-line code statement not supported
- ?9 = to be changed in a function call
- ?10 = element expression in set can only be a constant expression
- ?11 = to be changed in a procedure call

When the Translator encounters a statement that does not correspond to the syntax of a correct Pascal statement, it will issue an **\*\*\* ERROR \*\*\*** message followed by an explanation on what was found and what was expected. It will try to continue the translation and eventually it will skip some statements without translating them until it reaches a point where it can resynchronize itself with a correct Pascal statement. In this case, please modify your source and make sure that it compiles correctly with Turbo Pascal 3.0. Then, retranslate the program.

The total number of '?' flags emitted is found at the end of the 'X.mod' file.

The definitions of flag messages are as follows:

- ?0 = system problems  
If ?0 refers to a constant string too long, please split that string in two or more.

If you have reached the limits of the internal tables of the Translator as specified by the message posted with ?0, please modify your source code to avoid the problem. See the section on Translator's Capacities.

- ?1 = unknown identifier, undefined symbol or null operator  
 The identifier following '?1' is not known, that is, it has not been declared so far in the Pascal program. If you find ?1UNDEF it means that the Translator needs, at that point, a nonexistent identifier not needed in the Pascal program (see section on Copy, Concat and ParamStr). You will have to modify the Modula-2 program, declare a variable with the new identifier, according with the statement where the '?1UNDEF' occurs, and substitute ?1UNDEF with the new identifier.
  
- ?2 = invalid Modula-2 constant string  
 Pascal strings are very different from Modula-2 strings (see the section on Strings). This message occurs when you use a special feature of Turbo Pascal that has no automatic equivalent in Modula-2, that is, you are using a special control character representation to be embedded in string like:
 

```

        'This is another line of text'^M^J
        #27'Hello'
        #13#10^U^G
      
```
  
- ?3 = warning - using string make sure to follow the Modula-2 rules  
 This flag will be posted when you select the Translator option 'mark String = YES'. In all places where you use a string, the '?3' will precede the string identifier. This allows you to easily detect all the places where you are using strings to modify the algorithm. Pascal strings are very different from Modula-2 strings (see the section on Strings).
  
- ?4 = Pascal statement not supported by Modula-2  
 These statements (Label definition, Goto, Label use) cannot be translated into Modula-2 because such statements do not exist. You must modify your program to use other Modula-2 statements like LOOP, WHILE, REPEAT.
  
- ?5 = function or procedure parameter  
 This error will not occur with Turbo Pascal because it does not support procedures or functions passed as a parameter. Modula-2 allows you to pass procedures as parameter, and using this features allows you to implement sophisticated software engineering techniques.
  
- ?6 = absolute variable declaration  
 Modula-2 does support absolute variable declaration in a format slightly different from Turbo Pascal (see the section on Absolute Variable).



- ?7 = external procedure or subprogram declaration  
This Turbo Pascal statement cannot be automatically translated into Modula-2. This feature is supported in Modula-2 in a different way and requires you to do some work to change your program. The Translator will only mark the external references. Please refer to the others for a complete example on how to change your program to interface external subprograms. The external subprogram 'graphic' used by 'graph.p' is implemented by different Modula-2 library modules and the translation of the graphic routines is automatically done by the Translator without any additional work required from the user.
- ?8 = special in-line code statement not supported  
Reference to variables using a symbolic name is not allowed in Modula-2 CODE statements (see the section on Inline code).
- ?9 = to be changed in a function call  
Some of the Turbo Pascal operators (SHL, SHR, XOR, ...) are not available in Modula-2. To solve the problem, you can use a set of functions that implements the same features. These functions are in module MemoryOperation with names And, Or, Xor, Shl, Shr. (see the section on Expressions using And, Or, ...)
- ?10 = element expression in set can only be a constant expression  
The sets handling in Modula-2 is different than in Pascal (see the section on Sets).
- ?11 = to be changed in a procedure call  
LOGITECH MODULA-2/86 does not allow you to return structured data types from a function. Turbo Pascal allows functions to return Strings, for example, the Turbo Pascal functions Copy, Concat and ParamStr return strings. For this reason, the implementation of these three functions have been changed in Modula-2 in three procedures with an additional VAR parameter that holds the string to be returned (see the section on Copy, Concat and ParamStr). If your programs define and use other functions that return strings, you will have to change these functions in procedures with the additional parameter, and modify all the function calls into procedure calls.

### 1.3.5 Translator's Capacities

The principal capacities of the Translator are as follows:

- The maximum number of characters in all unique identifiers in a Pascal source program is 10,000. Identifiers are constant enumerated values, types, variables, fields, formal parameters, functions and procedures. All of the characters are retained.
- The maximum number of unique identifiers in a Pascal source program is 1,000. You can have 1000 identifiers with an average length of 10 characters before exceeding the Translator's capabilities, or up to 500 identifiers with an average length of 20 characters.
- The maximum number of nested levels is 30.

For example:

```
program ABC;
...
  procedure A;      (* nested level 2 *)
    ...
      procedure B;  (* nested level 3 *)
        ...
          procedure C; (* nested level 4 *)
            ...
              begin ... end (* C *);
            begin ... end (* B *);
          begin ... end (* A *);
        ...
      begin ... end;      (* program body - nested level 1 *)
```

- The overall capacity for data storage is based on the memory size of your PC available when you run the Translator.

If you exceed these capacities the Pascal program being translated must be divided into smaller files before the translation. The Modula-2 programs that result should be rejoined. This cannot be done by simply cutting the Pascal program indiscriminately because the global identifier references must have their declarations visible to the Translator, otherwise, you will have many '?' warnings. Also, each Pascal program translated must be a legal Turbo Pascal program or the Translator will give unpredictable results. If you use include files, one alternative is to rename some of the include files, so that the Translator will not find them, and translate separately the main file and the include file with the Translation option 'mark Undefined = NO' to avoid a lot of '?' warnings.

#### 1.4 Now Let's Try to Translate

Now that you have a general idea about the capabilities of the Translator, let's try to translate a simple program. In your diskette you will find a Turbo Pascal program, called FIRST.PAS. For your convenience we suggest you to work from the MOD editor, but you can apply the same commands from DOS. (In square brackets find the MOD commands separated by '|')

Edit FIRST.PAS:

[mod first.pas]

```
program first;

(* Program to demonstrate simple constant translation *)

const
  Version = 1;
  Sub_version = 0;
  Product = 'Turbo Translator';
  Bell = #$07; (* Bell hex code *)
```

```
procedure Siren;
var
  Frequency: integer;
begin
  for Frequency := 500 to 2000 do
  begin
    Delay(1);
    Sound(Frequency);
  end;
  for Frequency := 2000 downto 500 do
  begin
    Delay(1);
    Sound(Frequency);
  end;
  NoSound;
end;

begin
  writeln; writeln;
  write('Welcome to ',Product,' version ');
  writeln(Version,'.',Sub_version);
  writeln(Bell, bell, BELL, bELL);
  writeln('The Programmers'' time saver');
  SIREN;
END.
```

Translate this program by calling pas2mod:

[ALT F9 | r | pas2mod<CR>]

LOGITECH MODULA-2/86 Pascal to Modula-2 Translator,...

Type <Esc> to exit.

Pascal in >first<CR>

Modula-2 out ><CR>

Pascal in ><Esc><Esc>

FIRST.PAS is translated into FIRST.MOD without translation errors. Now edit FIRST.MOD:

[F3 | first]

```
MODULE first;
```

```
  FROM Sounds IMPORT Sound, NoSound;
```

```
  FROM Delay IMPORT Delay;
```

```
  FROM TTextIO
```

```
    IMPORT ReadInt, ReadCard, ReadChar, ReadString, ReadLn, ReadBuffer,
    WriteInt, WriteCard, WriteChar, WriteString, WriteBool, WriteLn,
    Eoln, SeekEof, SeekEoln;
```

```
  FROM TKernelIO
```

```
    IMPORT File, FileType, OptionMode, StatusProc, ReadProc, WriteProc,
    stdinout, input, output, con, trm, kbd, lst, aux, usr,
    conStPtr, conInPtr, auxInPtr, usrInPtr, conOutPtr, lstOutPtr,
    auxOutPtr, usrOutPtr, IOresult, KeyPressed, IOBuffer, IOCheck,
    DeviceCheck, CtrlC, InputFileBuffer, OutputFileBuffer;
```

```
(* Program to demonstrate simple constant translation *)
```

```
CONST
  Version = 1;
  SubVersion = 0;
  Product = 'Turbo Translator';
  Bell = 7C; (* Bell hex code *)

PROCEDURE Siren;

  VAR

    Frequency: INTEGER;
BEGIN
  FOR Frequency := 500 TO 2000 DO

    Delay(1);
    Sound(Frequency);
  END;
  FOR Frequency := 2000 TO 500 BY -1 DO

    Delay(1);
    Sound(Frequency);
  END;
  NoSound;
END Siren;

BEGIN
  WriteLn(stdinput);
  WriteLn(stdinput);
  WriteString(stdinput, 'Welcome to ', 0);
  WriteString(stdinput, Product, 0);
  WriteString(stdinput, ' version ', 0);
  WriteInt(stdinput, Version, 0);
  WriteChar(stdinput, '.', 0);
  WriteInt(stdinput, SubVersion, 0);
  WriteLn(stdinput);
  WriteString(stdinput, Bell, 0);
  WriteString(stdinput, Bell, 0);
  WriteString(stdinput, Bell, 0);
```

```
WriteString(stdinout, Bell, 0);
WriteLn(stdinout);
WriteString(stdinout, "The Programmers' time saver", 0);
WriteLn(stdinout);
Siren;
END first.
```

Note the following differences in the Modula-2 listing that reflect some new language features:

- Modula-2 programs start with the `MODULE` reserved word. The module name is inserted after the last `END` statement.
- Modula-2 does not support the underscore character as part of an identifier name. The Pascal `'Sub_version'` constant is translated into `'SubVersion'`.
- Modula-2 is case sensitive while Pascal is not (for example, in Modula-2 `Bell` is different from `BELL`). The Translator takes care to generate the correct name when a different capitalization is used. In our example, the constant `Bell` referred to in the program body in different modes (`Bell`, `bell`, `BELL`, `bELL`) is always translated as the declaration (`Bell`).
- The numeric hexadecimal constants assigned to `'Bell'` are converted into an octal notation. In Modula-2 octal numbers representing characters are terminated with the letter `'C'` if they are used as `CHAR`, or by the letter `'B'` if they are used as `Scalar` (`INTEGER` or `CARDINAL`).
- Modula-2 uses more procedures for output than Pascal, since it outputs each item by a typed I/O procedure.
- The string constant `"The Programmer's time saver"` is now enclosed in double quotes, since it contains a single quote.
- Modula-2 requires that you explicitly declare the function, procedure, types or variables you are using from other modules. The Translator takes care to automatically import the identifiers it generates. Because of the automatic nature of the translation, it actually imports ALL the exported identifiers of the module, regardless of how few are used.
- Note how the `FOR STATEMENT` is changed from Pascal to Modula-2. All Pascal statements and Turbo Pascal functions are automatically translated.

Now you can:

- 1 syntax check [F2],
- 2 compile [F5],
- 3 link [F6] and
- 4 run FIRST.MOD [ALT F9 | r | first]

and it will work like FIRST.PAS.



## 2 LANGUAGE FEATURES THAT REQUIRE MANUAL ADAPTATION

Due to some differences between the Pascal and Modula-2 languages in general, and Turbo Pascal 3.0 and LOGITECH MODULA-2/86 in particular, the Translator will not be able to automatically translate some statements or will translate them into Modula-2 statements that are not correct or optimal.

First, read the sections of this chapter that refer to Turbo Pascal features used on your program. Run the Translator on your program, but before compiling, we suggest that you pay attention to the following cases, study the translated program and eventually change some algorithms to make them more oriented to the Modula-2 style of programming. For the non trivial statements that are not fully translated, the user must intervene with manual editing, modifying the translated code using the suggestions given in this chapter.

### 2.1 Label and Goto Statements

While Pascal discourages GOTO statements, they are still offered by the language. Modula-2 does not support GOTOS or labels. Thus, Turbo Pascal programs that use GOTOS or labels must be rewritten using Pascal FOR, WHILE and REPEAT statements, and the new versions must then be translated. Or, first they can be translated into Modula-2 and later modified using Modula-2 FOR, WHILE, REPEAT and LOOP statements.

### 2.2 Constants and Initialized Variables

There are three categories of constants for the Translator:

- Simple untyped constants.
- Typed constants with scalar types.
- Typed constants with array, record and set types.

Turbo Pascal has two predefined integer constants -- 'Pi' and 'MaxInt'. They are inserted in a constant declaration by the Translator whenever they occur in the Turbo Pascal source program.

#### 2.2.1 Simple Untyped Constants

Simple untyped constants are generally translated in a straightforward manner, except for string constants with control characters and hex constants.

Consider the following Turbo Pascal program:

```
(* Pascal *)
const Version = 1;
      Sub_version = 0;
      Product = 'Turbo Translator';
      Bell = #$07; (* Bell hex code *)
begin
  ...
  writeln('The Programmers' time saver');
```

It translates into the following Modula-2 program:

```
(* Modula-2 *)
CONST
  Version = 1;
  SubVersion = 0;
  Product = 'Turbo Translator';
  Bell = 7C; (* Bell hex code *)
BEGIN
  ...
  WriteString(stdout, "The Programmers' time saver", 0);
```

Modula-2 does not support the underscore character as part of an identifier name. The Pascal 'Sub\_version' constant is translated into 'SubVersion'.

The numeric hexadecimal constants assigned to 'Bell' are converted into octal notation. In Modula-2, octal numbers representing CHARs are terminated with the letter 'C'.

The string constant "The Programmers' time saver" is now enclosed in double quotes, since it contains a single quote.

### 2.2.2 Variable Constants

Turbo Pascal permits explicitly typed constants. They are actually initialized variables since they can be assigned initial values. Thus, the Translator converts them into actual variables and places their declaration in the VAR section, while the initialization part goes at the beginning of the body (procedure or main). In the case of scalar types, the Translator is able to translate the initialization code correctly. Again string constants containing control characters and hexadecimal numbers must be edited by the user. For array, record and set variable constants the initialization parts are copied by the Translator at the beginning of the body but need editing from the user to conform to the Modula-2 syntax.

The following Turbo Pascal program demonstrates the limitations that require the user to edit the output Modula-2 program. It contains:

- a string constant with control characters, represented by the constant 'Greetings'.
- a variable constant declared as an array, represented by 'Programming\_Skill'. The array range is enumerated.

This program is similar to 'FIRST.PAS'. It is more conversant with the user, asking you about your programming skill and commenting on it.

in file SECOND.PAS:

```
PROGRAM second;
(* Program to demonstrate variable *)
(* constants translation.          *)

TYPE Skill = (Novice, Moderate, Expert);

CONST (* String constant with control characters *)
  Greeting = 'Hello there'^G^G;
  My_Name : STRING[20] = 'Turbo Translator';
  Version : INTEGER = 1;
  Sub_Version : INTEGER = 0;
  Programming_Skill : ARRAY[Skill] OF STRING[30] =
    ('New to Modula-2', 'Familiar', 'Real Pro');
```

```

VAR Answer, I : INTEGER;
    Your_Name : STRING[30];
    Skill_Index : Skill;

BEGIN
    WRITELN(Greeting, ' I am the ', My_Name); WRITELN;
    WRITELN('This is version ', Version, '.', Sub_Version);
    WRITELN; WRITE('What is your name? ');
    READLN(Your_Name); WRITELN;
    WRITELN('Hello ', Your_Name, ', are you ');
    WRITELN;
    I := 1;
    FOR Skill_Index := Novice TO Expert DO BEGIN
        WRITELN(I, ' ', Programming_Skill[Skill_Index]);
        I := I + 1;
    END;
    REPEAT
        WRITELN;
        WRITE('Select by number ');
        READLN(Answer);
    UNTIL (Answer > 0) AND (Answer < 4);
    WRITELN; WRITELN;
    WRITE('Very good ', Your_Name, '.');
    IF Answer < 3 THEN WRITELN(' I hope you become a real pro!')
        ELSE WRITELN(' It is so nice to meet a pro!');
    WRITELN; WRITELN;
END.

```

The Translator produces the listing shown below, appended by a list of warning messages. The Translator inserts question marks followed by a number that corresponds to the intended warning. Reading the Modula-2 program we find a '?2' after the declaration of the string constant 'Greetings'. Looking at the warning table we find,

```
?2 = invalid Modula-2 constant string
```

indicating that 'Greetings' has control characters and/or hexadecimal constants involved.

NOTE: A single control character or hex number is translated correctly, as was demonstrated in the first program.

Before we discuss the solution to the above problem, let us look at a second problem which is caused by the initialized array-typed constant. The Translator inserts comments reminding the user of the presence of constant arrays. As shown, it is copied 'AS IS' into the output Modula-2 program. The user must break down the one Turbo Pascal assignment statement into a series of assignments, one for each initialized array member.

```
in file SECOND.MOD:
```

```
MODULE second;
```

```
FROM TTextIO
```

```
  IMPORT ReadInt, ReadCard, ReadChar, ReadString, ReadLn, ReadBuffer,  
         WriteInt, WriteCard, WriteChar, WriteString, WriteBool, WriteLn,  
         Eoln, SeekEof, SeekEoln;
```

```
FROM TKernelIO
```

```
  IMPORT File, FileType, OptionMode, StatusProc, ReadProc, WriteProc,  
         stdinout, input, output, con, trm, kbd, lst, aux, usr,  
         conStPtr, conInPtr, auxInPtr, usrInPtr, conOutPtr, lstOutPtr,  
         auxOutPtr, usrOutPtr, errorPtr, IOresult, KeyPressed, IOBuffer,  
         IOCheck, DeviceCheck, CtrlC, InputFileBuffer, OutputFileBuffer;
```

```
(* Program to demonstrate variable *)
```

```
(* constants translation.          *)
```

```
TYPE
```

```
  Skill = (Novice, Moderate, Expert);
```

```
(* String constant with control characters *)
```

```
CONST
```

```
  Greeting = 'Hello there'^G^G ?2;
```

```
VAR
```

```
  MyName: ARRAY [0..20-1] OF CHAR;
```

```
  Version: INTEGER;
```

```
  SubVersion: INTEGER;
```

```
  ProgrammingSkill: ARRAY Skill OF ARRAY [0..30-1] OF CHAR;
```

```
VAR
  Answer, I: INTEGER;
  YourName: ARRAY [0..30-1] OF CHAR;
  SkillIndex: Skill;

BEGIN
(* !!! These variable were defined in Turbo Pascal as 'typed constants'  *)
(* and were initialized in the program declaration part                *)
(* !!! In Modula-2 'typed constants' become variables and are initialized *)
(* in the module or procedure body part                                *)
  ProgrammingSkill := ('New to Modula-2','Familiar','Real Pro');
  SubVersion := 0;
  Version := 1;
  MyName := 'Turbo Translator';
  WriteString(stdiout, Greeting, 0);
  WriteString(stdiout, ' I am the ', 0);
  WriteString(stdiout, MyName, 0);
  WriteLn(stdiout);
  WriteLn(stdiout);
  WriteString(stdiout, 'This is version ', 0);
  WriteInt(stdiout, Version, 0);
  WriteChar(stdiout, '.', 0);
  WriteInt(stdiout, SubVersion, 0);
  WriteLn(stdiout);
  WriteLn(stdiout);
  WriteString(stdiout, 'What is your name? ', 0);
  ReadBuffer(on);
  ReadString(stdiout, YourName);
  ReadLn(stdiout);
  ReadBuffer(off);
  WriteLn(stdiout);
  WriteString(stdiout, 'Hello ', 0);
  WriteString(stdiout, YourName, 0);
  WriteString(stdiout, ', are you ', 0);
  WriteLn(stdiout);
  WriteLn(stdiout);
  I := 1;
  FOR SkillIndex := Novice TO Expert DO
```

```
WriteInt(stdout, I, 0);
WriteChar(stdout, ')', 0);
WriteString(stdout, ProgrammingSkill[SkillIndex], 0);
WriteLn(stdout);
INC(I, 1);
END;
REPEAT

WriteLn(stdout);
WriteString(stdout, 'Select by number ', 0);
ReadBuffer(on);
ReadInt(stdout, Answer);
ReadLn(stdout);
ReadBuffer(off);
UNTIL (Answer > 0) AND (Answer < 4);
WriteLn(stdout);
WriteLn(stdout);
WriteString(stdout, 'Very good ', 0);
WriteString(stdout, YourName, 0);
WriteChar(stdout, '.', 0);
IF Answer < 3 THEN
    WriteString(stdout, ' I hope you become a real pro!', 0);
    WriteLn(stdout)
ELSE
    WriteString(stdout, ' It is so nice to meet a pro!', 0);
    WriteLn(stdout)
END;
WriteLn(stdout);
WriteLn(stdout);
END second.

*****
```

NUMBER OF ? = 1

Please refer to the translator manual for detailed explanation and solutions

- ?0 = system problems
- ?1 = unknown identifier, undefined symbol or null operator
- ?2 = invalid Modula-2 constant string
- ?3 = warning - using string make sure to follow the Modula-2 rules
- ?4 = Pascal statement not supported by Modula-2
- ?5 = function or procedure parameter
- ?6 = absolute variable declaration
- ?7 = external procedure or subprogram declaration
- ?8 = special in-line code statement not supported
- ?9 = to be changed in a function call
- ?10 = element expression in set can only be a constant expression
- ?11 = to be changed in a procedure call

The remedy for the first problem is to remove the control character from the string constant and create new constants. In our example, we have two Control-G characters. Thus, declaring a single 'Bell' constant is the first step. Next, we insert two 'WriteChar(stdinout, Bell, 0)' statements after the one that outputs 'Greetings'. As an alternative solution one could avoid declaring constants for 'Greetings' and 'Bell', and instead declare a variable 'GreetingsBell' to hold the string constant 'Greetings' and two extra characters for 'Bell'.

Later in the procedure body, assign the variable the constant string and the two bell extra characters, and finally write the variable.

```

VAR
    GreetingsBell : ARRAY [0..15] OF CHAR; (* bigger than we need *)
    ...
BEGIN
    GreetingsBell := "Greetings";
    GreetingsBell[ 9] := 7C;                (* 1st bell *)
    GreetingsBell[10] := 7C;                (* 2nd bell *)
    GreetingsBell[11] := 0C;                (* string terminator *)
    ...
    WriteString(stdinout, GreetingBell, 0);
    ...
END;
```



The original array constant assignment is broken down into three assignments. This removes the error causing condition. Other changes are marked by the comment (\* MODIFICATION \*). The edited Modula-2 program is shown below:

```

in file SECOND.M01:

MODULE second;

  FROM TTextIO
    IMPORT ReadInt, ReadCard, ReadChar, ReadString, ReadLn, ReadBuffer,
           WriteInt, WriteCard, WriteChar, WriteString, WriteBool, WriteLn,
           Eoln, SeekEof, SeekEoln;
  FROM TKernelIO
    IMPORT File, FileType, OptionMode, StatusProc, ReadProc, WriteProc,
           stdout, input, output, con, trm, kbd, lst, aux, usr,
           conStPtr, conInPtr, auxInPtr, usrInPtr, conOutPtr, lstOutPtr,
           auxOutPtr, usrOutPtr, errorPtr, IOresult, KeyPressed, IOBuffer,
           IOCheck, DeviceCheck, CtrlC, InputFileBuffer, OutputFileBuffer;

(* Program to demonstrate variable *)
(* constants translation.           *)

TYPE
  Skill = (Novice, Moderate, Expert);

(* String constant with control characters *)
CONST
  Greeting = 'Hello there';

(* This constant has been added *)      (* !!! MODIFICATION !!! *)
  Bell = 07C;

VAR
  MyName: ARRAY [0..20-1] OF CHAR;
  Version: INTEGER;
  SubVersion: INTEGER;
  ProgrammingSkill: ARRAY Skill OF ARRAY [0..30-1] OF CHAR;

VAR
```

```
Answer, I: INTEGER;
YourName: ARRAY [0..30-1] OF CHAR;
SkillIndex: Skill;
```

```
BEGIN
```

```
(* The three statements below were edited by the user *)
ProgrammingSkill[Novice] := 'New to Modula-2'; (* !!! MODIFICATION !!! *)
ProgrammingSkill[Moderate] := 'Familiar'; (* !!! MODIFICATION !!! *)
ProgrammingSkill[Expert] := 'Real Pro'; (* !!! MODIFICATION !!! *)

SubVersion := 0;
Version := 1;
MyName := 'Turbo Translator';
WriteString(stdout, Greeting, 0);

(* The two statements below are inserted by the user *)
WriteChar(stdout, Bell, 0); (* !!! MODIFICATION !!! *)
WriteChar(stdout, Bell, 0); (* !!! MODIFICATION !!! *)

WriteString(stdout, ' I am the ', 0);
WriteString(stdout, MyName, 0);
WriteLn(stdout);
WriteLn(stdout);
WriteString(stdout, 'This is version ', 0);
WriteInt(stdout, Version, 0);
WriteChar(stdout, '.', 0);
WriteInt(stdout, SubVersion, 0);
WriteLn(stdout);
WriteLn(stdout);
WriteString(stdout, 'What is your name? ', 0);
ReadBuffer(on);
ReadString(stdout, YourName);
ReadLn(stdout);
ReadBuffer(off);
WriteLn(stdout);
WriteString(stdout, 'Hello ', 0);
WriteString(stdout, YourName, 0);
WriteString(stdout, ', are you ', 0);
```

```
WriteLn(stdinput);
WriteLn(stdinput);
I := 1;
FOR SkillIndex := Novice TO Expert DO

    WriteInt(stdinput, I, 0);
    WriteChar(stdinput, ')', 0);
    WriteString(stdinput, ProgrammingSkill[SkillIndex], 0);
    WriteLn(stdinput);
    INC(I, 1);
END;
REPEAT

    WriteLn(stdinput);
    WriteString(stdinput, 'Select by number ', 0);
    ReadBuffer(on);
    ReadInt(stdinput, Answer);
    ReadLn(stdinput);
    ReadBuffer(off);
UNTIL (Answer > 0) AND (Answer < 4);
WriteLn(stdinput);
WriteLn(stdinput);
WriteString(stdinput, 'Very good ', 0);
WriteString(stdinput, YourName, 0);
WriteChar(stdinput, '.', 0);
IF Answer < 3 THEN
    WriteString(stdinput, ' I hope you become a real pro!', 0);
    WriteLn(stdinput)
ELSE
    WriteString(stdinput, ' It is so nice to meet a pro!', 0);
    WriteLn(stdinput)
END;
WriteLn(stdinput);
WriteLn(stdinput);
END second.
```

Let us consider another example. The next Turbo Pascal sample defines 'Home\_Info', a record type with three integer fields. The constant declaration section defines 'My\_Home' as an initialized record constant.

```

TYPE
    Home_Info = RECORD
        Bedrooms,
        Bath_Rooms,
        Doors : INTEGER;
    END;

CONST My_Home : Home_Info =
    (Bedrooms : 3; Bath_Rooms : 2; Doors : 1);

```

The Translator outputs the next Modula-2 listing. It contains comments that remind the user to resolve the Turbo Pascal structured constant assignment problem. The 'MyHome' identifier is now declared in the VAR section.

```

TYPE
    HomeInfo = RECORD
        Bedrooms, BathRooms, Doors: INTEGER;
    END;

VAR
    MyHome: HomeInfo;

...
BEGIN
(* !!! These variable were defined in Turbo Pascal as 'typed constants'  *)
(*    and were initialized in the program declaration part                *)
(* !!! In Modula-2 'typed constants' become variables and are initialized *)
(*    in the module or procedure body part                                *)

    MyHome := (Bedrooms:3;BathRooms:2;Doors:1);

```

To resolve the problem, we rewrite the original Turbo Pascal assignment into three field assignments enclosed in a WITH block. The correct Modula-2 program is:

```
...
BEGIN
  (* Next three line have been edited from the original output *)
  WITH MyHome DO
    Bedrooms := 3;
    BathRooms := 2;
    Doors := 1
  END;
```

## 2.3 Sets

Modula-2 supports the Set data type and operations the same as Turbo Pascal.

### 2.3.1 Large Sets

The LOGITECH MODULA-2/86 Rel 2.xx has a (temporary) implementation restriction on Sets. A Set is limited to only 16 elements, therefore it is impossible to use it to implement large sets, such as set of characters.

To overcome this limitation, LOGITECH supplies the library module LongSet that implements all the set operations as procedures or functions and allows the user to have large sets in his/her program. Using this module the user can define a set of any size as an ARRAY [0..x] OF BITSET and operate on it in a fashion similar to the Modula-2 set operators. The library exports the type 'SetOfChar' representing the full character set (all 256 members).

The relation among Set Operators in Turbo Pascal, Modula-2 and the LongSet Module is:

Set Operator	Turbo Pascal	Modula-2	LongSet MODULE
include elem. in set	set + element	INCL(s,e)	Include
exclude elem. in set	set - element	EXCL(s,e)	Exclude
union	+	+	SetUnion
difference	-	-	SetDifference
intersection	*	*	SetIntersection
symmetric set differ.	not available	/	SymmetricSetDiff.
test on elem.membership	IN	IN	InSet
test on set equality	=	=	EqualSet
test on set inequality	<>	#	NOT EqualSet
test on set inclusion	s1 <= s2	<=	SetIncluded(s1, s2)
test on set inclusion	s1 => s2	=>	SetIncluded(s2, s1)

The following examples suggest how to use the module LongSet; we suggest that first you study the definition module (file LONGSET.DEF)

```
(* Pascal *)
...
type Color = ( blue, .. , yellow, .. , green, .. , red );
( element #   0         6         26         31 )

var
  colorSet = Set of Color;
begin
  colorSet := [];
  ...
  colorSet := [ blue, yellow, green, red ];
  ...
```

```
(* two different translations can be used *)
```

```
(* Modula-2 version A *)
```

```
...
```

```
TYPE color = ( blue, .. , yellow, .. , green, .. , red );
```

```
VAR
```

```
    colorSet : ARRAY [ 0..1 ] OF BITSET;
```

```
BEGIN
```

```
    MakeEmptySet ( colorSet );
```

```
...
```

```
    colorSet[ 0 ] := colorSet [ 0 ] + { 0, 6 };
```

```
    colorSet[ 1 ] := colorSet [ 1 ] + { ( 26 - 16 ), ( 31 - 16 ) };
```

```
...
```

```
(* Modula-2 version B *)
```

```
...
```

```
TYPE color = ( blue, .. , yellow, .. , green, .. , red );
```

```
VAR
```

```
    colorSet,
```

```
    tempColorSet : ARRAY [ 0..1 ] OF BITSET; (* temporary variable *)
```

```
BEGIN
```

```
    MakeEmptySet ( colorSet );
```

```
...
```

```
    Include( colorSet, ORD( blue ) );
```

```
    Include( colorSet, ORD( yellow ) );
```

```
    Include( colorSet, ORD( green ) );
```

```
    Include( colorSet, ORD( red ) );
```

```
...
```

### 2.3.2 Set of Char

You use large sets in your Pascal code when you use statements such as:

```
(* Pascal *)  
...  
begin  
...  
repeat  
  ...  
  read(kbd, ch);  
until ch in ['Y', 'y', 'N', 'n', #27];  
...
```

In this case you are checking if the value of what you are reading from the keyboard is one of the expected values to terminate the repeat. The set ['Y', 'y', 'N', 'n', #27] is, in fact, a set of characters with only these five elements. Being a large set (it could contain up to 256 elements) it cannot be automatically translated into LOGITECH MODULA-2/86. The Translator will copy the statement as is, without flagging it, while the compiler will complain that the set is too large.

A possible solution to this problem is to use the LongSet module as described before. You create a temporary variable of type 'SetOfChar', make the variable an empty set, include in the set 'Y', 'y', 'N', 'n' and 33C, and finally, you should modify your code to test if 'ch' is in this set.

```
(* Modula-2 manual translation *)  
...  
IMPORT LongSet;  
...  
VAR mySet: LongSet.SetOfChar;  
...
```



```
BEGIN
MakeEmptySet ( mySet );
Include( mySet, WORD(ORD( 'Y' )) );
Include( mySet, WORD(ORD( 'y' )) );
Include( mySet, WORD(ORD( 'N' )) );
Include( mySet, WORD(ORD( 'n' )) );
Include( mySet, WORD(33C) );
...
REPEAT
    ...
    ReadChar(kbd, ch, 0);
UNTIL LongSet.InSet(mySet, WORD(ch) );
...
```

However, in this case, a correct and more readable approach would be:

```
(* Modula-2 manual translation *)
...
BEGIN
...
REPEAT
    ...
    ReadChar(kbd, ch, 0);
UNTIL (ch = 'Y') OR
      (ch = 'y') OR
      (ch = 'N') OR
      (ch = 'n') OR
      (ch = 33C);
...
```

Another example of the use of a set of characters, and an alternative translation is:

```
(* Pascal *)
...
if ch IN ['a', 'l'..'p', 'z', '0'..'9'] then
  ...
else
  ...

(* Modula-2 manual translation *)
...
CASE ch OF
  'a', 'l'..'p', 'z', '0'..'9':
    ...
ELSE
  ...
END;
```

### 2.3.3 Use of Variables in Set Construction

In Turbo Pascal you can use a variable in a set constructor. In Modula-2 this is not allowed and instead you have to use the predefined function INCL.

```
(* Pascal *)
...
type
  Days = (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
var
  myDay : Days;
  workDay : Set of Days;
begin
  workDay := [];
  myDay := Sat;
  workDay := [ Mon .. Wed, myDay ];
  ...
```

```

(* Modula-2 translation *)
...
TYPE
    Days = (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
    SetDays = SET OF Days; (* in Modula-2 you must define this type *)
                                (* to use the set constructor operator  *)

VAR
    myDay : Days;
    workDay : SetDays;

BEGIN
    workDay := SetDays ();          (* set constructor *)
    myDay := Sat;
    workDay := SetDays ( Mon .. Wed ); (* set constructor *)
    INCL(workDay, myDay);
    ...

```

Consider the next Turbo Pascal program which scans a line entered from the keyboard and counts the number of lower case characters, upper case characters, digits and other characters typed.

in file SETS.PAS

```

program sets;
(* Program to test translation of sets *)

type CharSet = SET OF CHAR;

var Answer : CHAR;
    Line : STRING[80];
    Digits, UpperCase, LowerCase : CharSet;
    I, Digit_Count, Lower_Count, Upper_Count, Others : INTEGER;

begin
    (* Initialize character sets *)
    Digits := ['0'..'9'];
    LowerCase := ['a'..'z'];
    UpperCase := ['A'..'Z'];
    repeat
        (* Initialize counters *)

```

```
Digit_Count := 0;
Upper_Count := 0;
Lower_Count := 0;
Others := 0;
writeln('Type a line up to 80 characters ');
readln(Line); writeln;
for I := 1 TO Length(Line) do
  if Line[I] in Digits
  then
    Digit_Count := Digit_Count + 1
  else if Line[I] in LowerCase then
    Lower_Count := Lower_Count + 1
  else if Line[I] in UpperCase then
    Upper_Count := Upper_Count + 1
  else Others := Others + 1;

writeln('You typed ',Length(Line),' characters with ');
writeln(Digit_Count:2,' digits');
writeln(Upper_Count:2,' upper case letters');
writeln(Lower_Count:2,' lower case letters');
writeln(Others:2,' other characters'); writeln;
WRITE('Want to type another line? (Y/N) ');
readln(Answer); writeln;
until Answer <> 'Y';
end.
```

The Translator produces an intermediate Modula-2 listing, as shown below:

in file SETS.MOD

```
MODULE sets;

  FROM Strings
    IMPORT Assign, Insert, Delete, Pos, Copy, Concat, Length, CompareStr;
  FROM TTextIO
    IMPORT ReadInt, ReadCard, ReadChar, ReadString, ReadLn, ReadBuffer,
           WriteInt, WriteCard, WriteChar, WriteString, WriteBool, WriteLn,
           Eoln, SeekEof, SeekEoln;
  FROM TKernelIO
    IMPORT File, FileType, OptionMode, StatusProc, ReadProc, WriteProc,
           stdinout, input, output, con, trm, kbd, lst, aux, usr,
           conStPtr, conInPtr, auxInPtr, usrInPtr, conOutPtr, lstOutPtr,
           auxOutPtr, usrOutPtr, errorPtr, IOresult, KeyPressed, IOBuffer,
           IOCheck, DeviceCheck, CtrlC, InputFileBuffer, OutputFileBuffer;

  (* Program to test translation of sets *)

  TYPE
    CharSet = SET OF CHAR;

  VAR
    Answer: CHAR;
    Line: ARRAY [0..80-1] OF CHAR;
    Digits, UpperCase, LowerCase: CharSet;
    I, DigitCount, LowerCount, UpperCount, Others: INTEGER;

  BEGIN
    (* Initialize character sets *)
    Digits := CharSet('0'..'9');
    LowerCase := CharSet('a'..'z');
    UpperCase := CharSet('A'..'Z');
    REPEAT

      (* Initialize counters *)
      DigitCount := 0;
```

```
UpperCount := 0;
LowerCount := 0;
Others := 0;
WriteString(stdout, 'Type a line up to 80 characters ', 0);
WriteLn(stdout);
ReadBuffer(on);
ReadString(stdout, Line);
ReadLn(stdout);
ReadBuffer(off);
WriteLn(stdout);
FOR I := 1 TO Length(Line) DO
  IF
    Line[I] IN Digits THEN

      INC(DigitCount, 1)
    ELSIF Line[I] IN LowerCase THEN

      INC(LowerCount, 1)
    ELSIF Line[I] IN UpperCase THEN

      INC(UpperCount, 1)
    ELSE
      INC(Others, 1)
    END
  END;
WriteString(stdout, 'You typed ', 0);
WriteCard(stdout, Length(Line), 0);
WriteString(stdout, ' characters with ', 0);
WriteLn(stdout);
WriteInt(stdout, DigitCount, 2);
WriteString(stdout, ' digits', 0);
WriteLn(stdout);
WriteInt(stdout, UpperCount, 2);
WriteString(stdout, ' upper case letters', 0);
WriteLn(stdout);
WriteInt(stdout, LowerCount, 2);
WriteString(stdout, ' lower case letters', 0);
WriteLn(stdout);
WriteInt(stdout, Others, 2);
```

```
WriteString(stdout, ' other characters', 0);
WriteLn(stdout);
WriteLn(stdout);
WriteString(stdout, 'Want to type another line? (Y/N) ', 0);
ReadBuffer(on);
ReadChar(stdout, Answer);
ReadLn(stdout);
ReadBuffer(off);
WriteLn(stdout);
UNTIL Answer <> 'Y';
END sets.
```

The user needs to perform a number of changes to produce a version that is accepted by the compiler. The Translator assumes that all sets declared in the program are small sets. This is not the case in our example, hence we must import from the library module 'LongSet'. The editing steps are as follows:

- 1 Insert an import list for the 'LongSet'. The list should contain 'BuildSet', 'Inset' and 'SetOfChar'.
- 2 Remove the character set declaration inherited from the parent Turbo Pascal program.
- 3 Replace the type 'CharSet' with 'SetOfChar'.
- 4 Replace the type INTEGER with CARDINAL.
- 5 The three sets are initialized using the imported procedure 'BuildSet()'. Even though the 'Digits' set has ten elements, we will treat it the same way as the other two. The 'ORD()' function is needed to convert a single-byte character into a CARDINAL occupying one WORD.
- 6 The FOR loop must have its lower and upper limits shifted by one.
- 7 Use the boolean function 'Inset()' in the IF statement and enclose the 'Line[I]' character in the 'ORD()' function.

The above modifications produce the correct and working Modula-2 program, shown below.

```

in file SETS.M01

MODULE sets;

  FROM Strings
    IMPORT Assign, Insert, Delete, Pos, Copy, Concat, Length, CompareStr;
  FROM TTextIO
    IMPORT ReadInt, ReadCard, ReadChar, ReadString, ReadLn, ReadBuffer,
    WriteInt, WriteCard, WriteChar, WriteString, WriteBool, WriteLn,
    Eoln, SeekEof, SeekEoln;
  FROM TKernelIO
    IMPORT File, FileType, OptionMode, StatusProc, ReadProc, WriteProc,
    stdinout, input, output, con, trm, kbd, lst, aux, usr,
    conStPtr, conInPtr, auxInPtr, usrInPtr, conOutPtr, lstOutPtr,
    auxOutPtr, usrOutPtr, errorPtr, IOresult, KeyPressed, IOBuffer,
    IOcheck, DeviceCheck, CtrlC, InputFileBuffer, OutputFileBuffer;
  FROM LongSet
    IMPORT BuildSet, InSet, SetOfChar;

(* Program to test translation of sets *)

VAR
  Answer: CHAR;
  Line: ARRAY [0..80-1] OF CHAR;
  Digits, UpperCase, LowerCase: SetOfChar;           (* MODIFIED *)
  I, DigitCount, LowerCount, UpperCount, Others: CARDINAL; (* MODIFIED *)

BEGIN
(* Initialize character sets *)
  BuildSet(Digits,ORD('0'),ORD('9'));                (* MODIFIED *)
  BuildSet(UpperCase,ORD('A'),ORD('Z'));             (* MODIFIED *)
  BuildSet(LowerCase,ORD('a'),ORD('z'));             (* MODIFIED *)

  REPEAT

```



```

(* Initialize counters *)
DigitCount := 0;
UpperCount := 0;
LowerCount := 0;
Others := 0;
WriteString(stdout, 'Type a line up to 80 characters ', 0);
WriteLn(stdout);
ReadBuffer(on);
ReadString(stdout, Line);
ReadLn(stdout);
ReadBuffer(off);
WriteLn(stdout);
FOR I := 0 TO Length(Line)-1 DO                                (* MODIFIED *)
  IF
    InSet(Digits,ORD(Line[I])) THEN                            (* MODIFIED *)

      INC(DigitCount, 1)

    ELSIF InSet(LowerCase,ORD(Line[I])) THEN                    (* MODIFIED *)

      INC(LowerCount, 1)

    ELSIF InSet(UpperCase,ORD(Line[I])) THEN                    (* MODIFIED *)

      INC(UpperCount, 1)
    ELSE
      INC(Others, 1)
    END
  END;
WriteString(stdout, 'You typed ', 0);
WriteCard(stdout, Length(Line), 0);
WriteString(stdout, ' characters with ', 0);
WriteLn(stdout);
WriteInt(stdout, DigitCount, 2);
WriteString(stdout, ' digits', 0);
WriteLn(stdout);
WriteInt(stdout, UpperCount, 2);
WriteString(stdout, ' upper case letters', 0);
WriteLn(stdout);
WriteInt(stdout, LowerCount, 2);
WriteString(stdout, ' lower case letters', 0);

```

```
WriteLn(stdiout);
WriteInt(stdiout, Others, 2);
WriteString(stdiout, ' other characters', 0);
WriteLn(stdiout);
WriteLn(stdiout);
WriteString(stdiout, 'Want to type another line? (Y/N) ', 0);
ReadBuffer(on);
ReadChar(stdiout, Answer);
ReadLn(stdiout);
ReadBuffer(off);
WriteLn(stdiout);
UNTIL Answer <> 'Y';
END sets.
```

## 2.4 Strings

Pascal and Modula-2 implement strings in a very different way. This implies a fundamentally different mechanism for keeping track of strings that will be outlined in the following sections. If you make extensive use of the Turbo Pascal data type 'string[x]', read these notes carefully and apply to the translated Modula-2 code all the needed modifications before compiling, linking and running your program.

To help you pinpoint all the places in your program where you use strings, you can run the Translator with the option 'mark String' at YES. The generated Modula-2 code will contain the flag ?3 in ALL places where you use or refer to a string, so that you can have an easy way to detect these places and to modify your code if needed.

### 2.4.1 Differences Between Pascal and Modula-2

In Pascal you can declare strings with 'n' characters as STRING[n]. The element at position 0 of the string contains the length of the string. The first valid character in the string starts from index 1 up to index n. The maximum size of a Turbo Pascal string is 255 characters.

It is possible to know the length of your string by accessing the index 0 and modifying it, so you can write statements such as:

```

...
len := ord(mystring[0]);          (* get the length *)
...
for i := 1 to ord(mystring[0]) do  (* loops on all valid element *)
...
mystring[0] := chr(40);          (* set length to 40 chars *)
...

```

In Modula-2 THERE IS NO predefined identifier 'string', thus strings are implemented as an array of characters. In an ARRAY [x..y] OF CHAR, elements are stored starting from the lowest index of the array (x) up to the highest (y). The maximum size of a string or array of characters is not limited to 255 like in Turbo Pascal, but can be up to 65535 characters.

If the number of valid characters fills the whole array, the string is not terminated. Otherwise, strings shorter than the array are terminated by storing the binary value 0 after the last valid element (like in the C language). In other words, starting from the lowest index, an ARRAY OF CHAR is terminated when you encounter an element with the binary value zero or reach the highest index. This implies that binary zero is NOT a valid character.

```

VAR
  mystring: ARRAY [0..3] OF CHAR;

BEGIN
  mystring[0] := 'A';  (* loading mystring with 'ABCD' *)
  mystring[1] := 'B';
  mystring[2] := 'C';
  mystring[3] := 'D';  (* string fully loaded no need to terminate *)

  mystring := 'ABCD';  (* another way to load mystring with 'ABCD' *)

  mystring[0] := 'A';  (* loading mystring with 'AB' only *)
  mystring[1] := 'B';
  mystring[2] := 0C;   (* string NOT fully loaded, terminated by 0C *)

  mystring := 'AB';    (* another way to load mystring with 'AB' *)

```

```
(* here the termination is done by the code *)
(* generated by the compiler *)
```

## NOTE:

- In Modula-2 0C is different from 0 (zero) or 0H. Modula-2 uses strong type checking so the correct way to represent a CHAR with binary value 0 is to use the its octal representation followed by the letter C (C for CHAR). For example, to load a CHAR with the letter 'B' you can use both "B" or 102C, but not 42H because this is NOT a CHAR constant.

```
...
mystring[0] := 'A';
mystring[1] := 102C; (* correct B *)
mystring[2] := 43H; (* wrong *)
mystring[3] := 33C; (* correct ESCAPE *)
...
```

- An empty string is represented both by 0C or '':

```
...
Concat ( '', mystring, ... );
mystring := 0C; (* empty string *)
...
```

- Single character strings can be considered both as an ARRAY OF CHAR or as a single CHAR.

'A' is an ARRAY [0..0] OF CHAR as well as a CHAR.

'' is an empty string, NOT a CHAR.

0C is a CHAR and also represents an empty string.

## 2.4.2 How Strings are Translated From Pascal to Modula-2

Turbo Pascal strings are translated into an array of characters, with the lower index always zero (0).

```
(* Pascal *)
...
myString : string[10];
otherString : string[maxStr];
...

(* Modula-2 *)
...
myString : ARRAY [0..10-1] OF CHAR;
otherString : ARRAY [0..maxStr-1] OF CHAR;
...
```

Starting the character string with an index of 0 (instead of 1) is necessary so the compiler can perform automatic string assignments, such as:

```
...
VAR
  Title: ARRAY [0..63] OF CHAR;
...
Title := 'Gone with the wind';
...
```

In this case, Title[0] stores 'G', Title[1] stores 'o' and so on. The last character 'd' is stored in Title[17] and the compiler will properly terminate the string by storing 0C in Title[18]. For string procedures to operate correctly, this is mandatory.

Since in Modula-2 a valid string starts with an index 0, and not 1 as in Turbo Pascal, some re-editing of your program is required to insure correct behavior at run time. For example, this is necessary if you use FOR statements or loops starting from index 1, or if you use string functions like Pos or Delete.

**Remember:** In your Turbo Pascal program the valid part of the String starts at index 1, in Modula-2 it starts at Index 0.

Generally, you should pay attention to statements which directly address the element at index 0 of a string. For example, storing the length of a string in a variable:

```
x := myString[0];
```

or loading the actual length of a string:

```
myString[0] := 10;
```

These statements **MUST** be modified.

Also if you address the elements of a string using a loop statement (FOR, WHILE, REPEAT), remember to start from element 0 and terminate either when you reach the element with value 0C or at the physical end of the array, element Max-1.

Here are some examples of string manipulation:

```
(* Pascal *)
...
type MaxString: string[100];
...
function LoadStr(Len: Integer; var Str: MaxString);
(* if in Str there is an '*' loads Str with Len characters '*' *)
var i: integer;
    found: boolean;
begin
    found := FALSE;
    for i := 1 to ord(Str[0]) do
        if Str[i] = '*' then found := TRUE
    if found then
        begin
            for i := 1 to Len do
                begin
                    Str[i] := '*';
                end;
            Str[0] := Chr(Len);
        end
    end;
...
end;
```

```

(* Modula-2 manually modified *)
...
TYPE MaxString: ARRAY [0..99] OF CHAR;
...
PROCEDURE LoadStr(Len: INTEGER; VAR Str: MaxString);
(* if in Str there is an '*' loads Str with Len characters '*' *)
VAR i: INTEGER;
    found: BOOLEAN;
BEGIN
    found := FALSE;
    i := 0;
    WHILE (i <= HIGH(Str)) AND          (* modified to detect *)
           (Str[i] # 0C) DO             (* end of string      *)
        IF Str[i] = '*' THEN found := TRUE; END;
        INC(i);
    END;
    IF found THEN
        FOR i := 0 TO Len-1 DO           (* modified 1->0, Len->Len-1 *)
            Str[i] := '*';
        END;
        Str[Len] := 0C;                  (* modified *)
    END;
END LoadStr;

(* Modula-2 manually modified, alternative solution using module Strings *)
...
FROM Strings IMPORT Length;
...
TYPE MaxString: ARRAY [0..99] OF CHAR;
...
PROCEDURE LoadStr(Len: INTEGER; VAR Str: MaxString);
(* if in Str there is an '*' loads Str with Len characters '*' *)
VAR i: INTEGER;
    found: BOOLEAN;
    actualLen: CARDINAL;                 (* used to hold the actual length *)

```

```

BEGIN
  found := FALSE;
  actualLen := Length(Str);          (* modified to use module Strings *)
  FOR i := 0 TO actualLen DO
    IF Str[i] = '*' THEN found := TRUE; END;
  END;
  IF found THEN
    FOR i := 0 TO Len-1 DO           (* modified 1->0, Len->Len-1 *)
      Str[i] := '*';
    END;
    Str[Len] := 0C;                 (* modified *)
  END;
END LoadStr;

```

### 2.4.3 String Operator '+' in Modula-2

Modula-2 does not allow you to use the operator '+' to concatenate two or more strings or characters. The correct way to perform such an operation is to use the procedure 'Concat' from module Strings (see the section on Concat). These modifications should be done manually, for example:

```

(* Pascal *)
...
var str1, str2, str3: string[100];
...
begin
  str1 := 'Release';
  str2 := 'DOS';
  str3 := str1 + ' 1.0 ' + str2 + ^M;
  ...
  (* the result is the string 'Release 1.0 DOS' with <CR> at the end *)
  ...

(* Modula-2 manual translation *)
...
FROM Strings IMPORT Concat;
...
VAR str1, str2, str3: ARRAY [0..99] OF CHAR;
...

```



```

BEGIN
  str1 := 'Release';
  str2 := 'DOS';
  str3 := '';                (* added to make an empty string *)
  Concat(str1, ' 1.0 ', str3); (* loads str1+' 1.0 ' into str3, *)
  Concat(str3, str2, str3);   (* here adds str2 to str3 and *)
  Concat(str3, 15C, str3);    (* here adds the character ^M to str3 *)
  ...
  (* the result is the string 'Release 1.0 DOS' with <CR> at the end *)
  ...

```

Note that the Modula-2 procedure Concat from module Strings is VERY DIFFERENT from the Turbo Pascal function Concat. Turbo Pascal Concat is a function that takes a variable number of parameters and returns the concatenated string while Modula-2 Concat takes two source strings s1 and s2 (first and second parameter), concatenates them and puts the result in string s3 (third parameter). Modula-2 Concat does not allow a variable number of parameters and returns the result, as the third parameter of the procedure. For these reasons each '+' operator in the expression should be changed into a call to Concat. Also if there is no string variable available to hold the result, a local temporary variable should be used.

```

(* Pascal *)
...
begin
  write( 'Hello world' + ^G);
  ...
  (* the string 'Hello world' goes on the screen plus the bell rings *)
  ...

(* Modula-2 *)
...
FROM Strings IMPORT Concat;
...
VAR tempString : ARRAY [0..13] OF CHAR;  (* temporary string *)
...

```

```

BEGIN
  Concat('Hello world', 07C, str3);
  WriteString(str3);
  ...
  (* the string 'Hello world' goes on the screen plus the bell rings *)
  ...

```

#### 2.4.4 String Expression in Modula-2

Modula-2 does not allow you to use the relational operators '=', '<', '>', '<=', '>=', '<>' between structured data types (arrays and records). For strings, the right way to perform such an operation is to use the procedure 'CompareStr' from module Strings. This function compares s1 and s2 (first and second parameters) and returns -1 if s1 is less than s2, 0 if s1 equals s2, +1 if s1 is greater than s2. These modifications should be done manually, for example:

```

(* Pascal *)
...
var str1, str2, str3: string[100];
...
begin
  if str1 = str2 then ...
  ...
  if str2 >= str3 then ...
  ...
  if str1 < str3 then ...
  ...

(* Modula-2 *)
...
FROM Strings IMPORT CompareStr;
...
VAR str1, str2, str3: ARRAY [0..99] OF CHAR;
...

```

```
BEGIN
  IF CompareStr(str1, str2) = 0 THEN ...
  ...
  IF CompareStr(str2, str3) >= 0 THEN ...
  ...
  IF CompareStr(str1, str3) = -1 THEN ...
  ...
```

Further examples on string expressions are in the files `EXPR.PAS` and `EXPR.MOD`.

- The string assignment operations simply have been copied. Some will cause the compiler to generate error messages and user intervention is needed (see section on Strings).

```
Str4 := Str1; (* not valid because of different type *)
```

should be changed to:

```
Assign(Str1, Str4);
```

- The string expressions simply have been copied. This will cause the compiler to generate error messages and user intervention is needed (see section on Strings).

```
Str3 := 'Hello '+'There';
Str3 := Ch1+Str1;
```

should be changed to:

```
Concat('Hello', 'There', Str3);
Concat(Ch1, Str1, Str3);
```

- The boolean expression using the string is also copied, similar to the above string expressions. This will cause the compiler to generate error messages and user intervention is needed (see section on Strings).

```
Flag := Str2 = 'Sally';
```

should be changed to:

```
Flag := CompareStr(Str2,'Sally') = 0;
```

### 2.4.5 Control Characters Used as String Constants

Modula-2 does not allow you to use special control character representation to be embedded in strings. In these cases, the strings should be broken down into components and then you use the Concat function or the assign statement.

```
(* Pascal *)
...
var str1: strings[100];
...
begin
  str1 := 'This is another line of text'^M^J;
  ...
  write('^M'Hello world'^G);
  ...

(* Modula-2 *)
...
VAR str1: ARRAY [0..99] OF CHAR;
    tempString : ARRAY [0..12] OF CHAR;    (* temporary string *)
...
BEGIN
  str1 := 'This is another line of text';
  str[28] := 15C;    (* ^M as a character in octal representation *)
  str[29] := 12C;    (* ^J as a character in octal representation *)
  str[30] := 0C;    (* string terminator *)
  ...
```

```

tempString[0] := 15C;  (* ^M as a character in octal representation *)
tempString[1] := 0C;  (* string terminator *)
Concat(tempString, 'Hello world', tempString);
tempString[12] := 07C; (* ^G as a character in octal representation *)

(* there is no need to terminate this array with 0C because the string *)
(* fills the whole physical space of the array *)

write(tempString);
...

```

#### 2.4.6 Turbo Pascal and Modula-2 Standard String Functions

The module String of LOGITECH MODULA-2/86 provides a complete set of string handling procedures. Most of these procedures are similar to their counterpart in Turbo Pascal, but some of them are slightly different (see Appendix A for more details and the definition module String).

- Length

Same as Modula-2, the result is always positive and could be stored in a CARDINAL variable.

- Delete, Insert

Same as Modula-2, but be careful to change the starting position because in Modula-2 the string starts from index 0. Generally, you should assume that the Modula-2 Pos is equal to TurboPascal Pos-1. For example, if Str has the value 'ABCDEFG', to remove 'BCDE':

```

in Turbo      Delete(Str, 2, 4);
in Modula-2   Delete(Str, 1, 4);

```

while if Str has the value 'ABCDEFG' and you want the value 'ABXXCDEFG':

```

in Turbo      Insert('XX', Str, 3);
in Modula-2   Insert('XX', Str, 2);

```

Remember that in Modula-2, the first element of a string is at position 0.

- Pos

Same as Modula-2, but if the element is not found, Turbo returns 0 while Modula-2 returns a CARDINAL greater than HIGH(string).

- Str  
Depending on the parameter, Str is translated into IntToStr or RealToStr.
- Val  
Depending on the parameter, Val is translated into StrToInt or StrToReal.
- Copy, Concat  
These Turbo Pascal functions are translated into the equivalent Modula-2 procedures, but require particular attention from the user. Moreover, with the Copy function you may have the same problems as with Delete and Insert because you must specify a position. Refer to the sections on Copy, Concat and ParamStr in this chapter.

#### 2.4.7 Functions Returning a String

With Turbo Pascal, you can define a function that returns a string. LOGITECH MODULA-2/86 Version 2.x does not allow you to return structured data types from a function, thus you cannot define a function returning an ARRAY OF CHAR. Some examples of Turbo Pascal functions which return a string are Copy, Concat and ParamStr. If you use them, refer to the sections on Copy, Concat and ParamStr in this chapter.

If your program declares and uses other functions that return structured data types, you must change these functions into procedures with an additional parameter, and modify all the function calls into procedure calls. For more details refer to the section on Functions Returning Strings.

#### 2.4.8 Open Arrays

One of the limitations of Pascal is the strong type checking on strings passed as VAR parameters, in other words, the length of the actual and formal parameter must match. The Turbo Pascal compiler directive `{$V-}` relaxes this type checking, allowing to pass actual parameters with lengths that do not match the formal parameter length.

Modula-2 allows you more powerful programming techniques using its 'open arrays' and the predefined function HIGH(x). For more information, refer to any introductory book on Modula-2. Using both features, you can easily build procedures that operate on generic ARRAY OF <type>, like the following example of a procedure that counts the number of spaces in a generic string:

```

(* Modula-2 *)
...
PROCEDURE CountSpaces(str: ARRAY OF CHAR): CARDINAL;
VAR
    i, count: CARDINAL;
BEGIN
    count := 0;
    i := 0;
    WHILE (i <= HIGH(str) AND (str[i] # 0C) DO
        IF (str[i] = ' ') THEN INC(count); END;
        INC(i);
    END;
    RETURN count;
END CountSpaces;
...
VAR
    myStr: ARRAY [10..90] OF CHAR;
...
BEGIN
    ...
    Assign('This is a test string', myStr);
    numberOfSpaces := CountSpaces(myStr);
    numberOfSpaces := CountSpaces('This is a second test string!');
    ...

```

In this example, `str` is an 'open array'. The formal parameters 'str' will be bound to an actual parameter that is a one-dimensional array of the specified type. The index type of the array is not known. In the body of the procedure `CountSpaces` there is no need to know the index type of the actual parameter as long as there is a way to access each element of the array. Modula-2 provides a way by adopting the rule that a formal parameter of an open array type is assumed to have an index type that is a subrange of the cardinal type, beginning at zero (0), while the maximum index in the range is given by the predefined function `HIGH(x)`, where `x` is the open array parameter.

The test `(i <= HIGH(str))` is used to detect the physical end of the array, in fact `HIGH(str)` returns the high index for `str` with the low index assumed to be zero. The test `(str[i] # 0C)` detects the logical end of the string. The `0C` marker indicates that the rest of the array until the physical end does not contain valid data.

A complete example on how to transform a program from Turbo Pascal into a Modula-2 program using open arrays, is on files STAR.PAS (Pascal Version), STAR.MOD (Modula-2 translation) and STAR.MO1 (Modula-2 modified to include open arrays).

## 2.5 User Defined Functions Returning a String

With Turbo Pascal you can define a function that returns a string. LOGITECH MODULA-2/86 Version 2.x does not allow you to return a structured data type from a function, thus you cannot define a function returning an ARRAY OF CHAR. If your program defines other functions that return structured data types, you will have to change these functions into procedures with an additional parameter, and modify all the function calls into procedure calls. For example:

```
(* Pascal *)
...
type  MaxString = string [255];
var   str50: MaxString;
...
function BuildStr (Len: Integer): MaxString;
(* returns a string with Len '!' *)
var I: integer;
begin
  BuildStr[0] := Chr(Len);
  for I := 1 to Len do
    BuildStr[I] := '!';
end;
...
str50 := BuildStr(10);
...
Write(BuildStr(10));
...
```



```

(* Modula-2 after translation *)
...
TYPE
  MaxString = ARRAY [0..255-1] OF CHAR;
VAR
  str50: MaxString;

(* returns a string with Len '!' *)

PROCEDURE BuildStr(Len: INTEGER): MaxString ?11 ; <--- translator flag

  VAR
    I: INTEGER;
  VAR BuildStrResult: MaxString;
BEGIN
  BuildStrResult[0] := CHR(Len);
  FOR I := 1 TO Len DO
    BuildStrResult[I] := '!'
  END;
  RETURN BuildStrResult
END BuildStr;
...
str50 := BuildStr(10);
...
WriteString(stdout, BuildStr(10), 0);
...

*****
NUMBER OF ? = 1

```

Please refer to the Translator manual for detailed explanations and solutions

?0 = system problems

...

?11 = to be changed in a procedure call

The Translator will mark the procedure heading with '?11' because the function is returning a string. The steps to modify this example to generate a correct Modula-2 program involve both the change from a function to a procedure, including the change from function calls to procedure calls, and changes due to the string handling in the procedure BuildStr itself:

From Function to Procedure (a):

- Modify the procedure heading adding an additional VAR parameter to hold the return value and removing the function return type.
- Remove the local variable <functionName>Result automatically generated by the Translator for each function. There is no need for this variable given that an additional parameter has been added.
- Remove the RETURN statement at the end of the function. There is no need for it in procedures.
- Modify the call to BuildStr. If the function is assigned to a variable, just move the variable as the return parameter function. If the function is the string parameter of another procedure X you need a temporary variable to hold the result of the function, and then you can use this temporary variable as a parameter for the procedure X.

String handling (b):

- As explained in the String section of this chapter, the way to handle strings in Modula-2 is very different from Turbo Pascal.
- In Modula-2, your string of length Len is stored in an ARRAY OF CHAR with the first element at index 0 up to the last element at index Len-1.
- The same string is terminated by the character 0C at index Len (if the length of the string is shorter than the size of the array).

```

(* Modula-2 after translation and manual modification*)
...
TYPE
  MaxString = ARRAY [0..255-1] OF CHAR;
VAR
  str50: MaxString;

VAR
  tempStr : MaxString;          (* added because of Write call later (a) *)

(* returns a string with Len '!' *)

PROCEDURE BuildStr(   Len: INTEGER;
                    VAR BuildStrResult: MaxString);          (* modified (a) *)
  VAR
    I: INTEGER;
  (* VAR BuildStrResult: MaxString; removed because change in heading (a) *)
  BEGIN
  (* BuildStrResult[0] := CHR(Len); removed because of string handling (b) *)
    FOR I := 0 TO Len-1 DO      (* modified because of string handling (b) *)
      BuildStrResult[I] := '!'
    END;
    BuildStrResult[Len] := 0C;  (* added because of string handling (b) *)
  (* RETURN BuildStrResult removed because of change in procedure (a) *)
  END BuildStr;
...
BuildStr(10, str50);          (* modified because of change in procedure (a) *)
...
BuildStr(10, tempStr);        (* modified using a temporary variable *)
WriteString(stdinout, tempStr, 0); (* because of change in procedure (a) *)
...

```

## 2.6 Copy, Concat, ParamStr Functions and the Flag ?1UNDEF

The Modula-2 definition of Copy, Concat (module Strings) and ParamStr (module TParameter) is different from the Turbo Pascal version. LOGITECH MODULA-2/86 Version 2.x does not allow you to return structured data types from a function, thus you cannot define a function returning a string that is an ARRAY OF CHAR.

For this reason, the definition of these three functions has been changed in Modula-2 into three procedures with an additional VAR parameter that holds the string to be returned. When Copy, Concat and ParamStr are used as the rightmost part of an assignment, the translation will be correct, while if they are used as parameters of other functions, some manual modification is needed.

Moreover, the Turbo Pascal function Concat allows a variable number of parameters, a feature not available in Modula-2. A Turbo Pascal program using a Concat with more than two parameters is translated with a series of Modula-2 Concat which produce the same result. With the function Copy, you must specify a position and this can create the same problems as described for the Delete and Insert function in the section on Strings.

Using Copy, Concat and ParamStr in assignments does not create problems. The Translator will generate the correct code. It will use the leftmost part of the assignment as the procedure result parameter:

```
(* Pascal *)
...
var s1, s2, s3: string[10];
    str: string[255];
begin
  ...
  str := copy(s1, 1, 2);
  ...
  str := concat(s1,s2);
  ...
  str := concat(s1, 'abc', s2, ^G, s3);
  ...
  str := paramstr(5);
  ...
```

In these cases, the code produced by the Translator is correct except for the modifications needed to the position parameter of Copy :

```
(* Modula-2 after translation *)
...
FROM Strings IMPORT Copy, Concat;
FROM TParamater IMPORT ParamStr;
...
```

```

VAR
  s1, s2, s3: ARRAY [0..10-1] OF CHAR;
  str: ARRAY [0..255-1] OF CHAR;
BEGIN
  ...
  Copy(s1, 0, 2, str);          (* modify starting position from 1 to 0 *)
  ...
  Concat(s1, s2, str);
  ...
  Concat(s1, 'abc', str);
  Concat(str, s2, str);
  Concat(str, 'C', str);
  Concat(str, s3, str);
  ...
  ParamStr(5, str);
  ...

```

Using Copy, Concat and ParamStr as parameters of functions will produce Modula-2 code with the flag ?11. The code should be modified with the introduction of temporary variables used to hold the procedure result:

```

(* Pascal *)
...
var s1, s2, s3: string[10];
    str: string[255];
begin
  ...
  write(copy(s1, 1, 2));
  ...
  write(concat(s1,s2));
  ...
  write(concat(s1, 'abc', s2, ^G, s3));
  ...
  write(paramstr(5));
  ...

```

In these cases, the code produced by the Translator contains flag ?11 and manual modification is needed:

```
(* Modula-2 after translation *)
...
FROM Strings IMPORT Copy, Concat;
FROM TParamater IMPORT ParamStr;
...
VAR
  s1, s2, s3: ARRAY [0..10-1] OF CHAR;
  str: ARRAY [0..255-1] OF CHAR;
BEGIN
  ...
  WriteString(stdinout, ?11 Copy(s1, 1, 2, ?1UNDEF), 0);
  ...
  WriteString(stdinout, ?11 Concat(s1, s2, ?1UNDEF), 0);
  ...
  WriteString(stdinout, ?11 Concat(s1, 'abc', ?1UNDEF);
                    ?11 Concat(?1UNDEF, s2, ?1UNDEF);
                    ?11 Concat(?1UNDEF, 7C, ?1UNDEF);
                    ?11 Concat(?1UNDEF, s3, ?1UNDEF);
, 0);
  ...
  WriteString(stdinout, ?11 ParamStr(5, ?1UNDEF), 0);
```

The flag ?11 indicates that the following function is now a procedure and cannot be used as parameter (in this case for WriteString), thus the first step is to remove these procedures from the parameter list.

The flag ?1UNDEF indicates that a variable is expected in that position, but none is available. The user should declare a temporary variable of the type expected by the procedure and use that variable where ?1UNDEF indicates.

The manually modified version of the Modula-2 code is as follows:

```
(* Modula-2 manually modified *)
...
FROM Strings IMPORT Copy, Concat;
FROM TParamater IMPORT ParamStr;
...
VAR
  s1, s2, s3: ARRAY [0..10-1] OF CHAR;
  str: ARRAY [0..255-1] OF CHAR;
  tempStr: ARRAY [0..254] OF CHAR;      (* added temporary string *)
BEGIN
  ...
                                (* modify starting position from 1 to 0 and *)
Copy(s1, 0, 2, tempStr);          (* replace ?1UNDEF with tempStr *)
WriteString(stdout, tempStr, 0);  (* replace Copy() with tempStr *)
...
Concat(s1, s2, tempStr);          (* replace ?1UNDEF with tempStr *)
WriteString(stdout, tempStr, 0);  (* replace Concat() with tempStr *)
...
Concat(s1, 'abc', tempStr);       (* replace ?1UNDEF with tempStr *)
Concat(tempStr, s2, tempStr);
Concat(tempStr, 7C, tempStr);
Concat(tempStr, s3, tempStr);
WriteString(stdout, tempStr, 0);  (* replace Concat() with tempStr *)
...
ParamStr(5, tempStr);             (* replace ?1UNDEF with tempStr *)
WriteString(stdout, tempStr, 0);  (* replace ParamStr() with tempStr *)
```

## 2.7 Other Data Types

This section looks at the differences in data types found in both Turbo Pascal and Modula-2.

### 2.7.1 Integers, Cardinals and Subranges

Both types represent whole numbers and are stored using the same number of bits. The **CARDINAL** type is different from **INTEGER** in that the leftmost bit is used to contribute to its value. This contrasts with **INTEGER**s where the leftmost bit is used to store the sign. Thus, **CARDINAL**s have values ranging from zero to twice the range of positive **INTEGER** values. The range of a **CARDINAL** is 0..65535, while for an **INTEGER** it is -32768..32767.

Modula-2 programmers tend to employ more **CARDINAL**s than **INTEGER**s in their software. In many cases, this is optional. However, there are other instances where shifting from **INTEGER** to **CARDINAL** is necessary to correctly compile a program. One such instance is the use of the ordinal function **ORD**. In Turbo Pascal, it returns an **INTEGER**. In Modula-2 it returns a **CARDINAL**. This can create a type incompatibility when mixed with **INTEGER**s in an expression. The solution is to use a 'cast' function, in other words, to officially exchange the data type you are using with another one. For example, use the **INTEGER()** converter with Modula-2 functions that return **CARDINAL**s:

```
(* Pascal *)
...
type Color = (Red, Blue, Green, Yellow);
const Guess_Color : Color = Red;
var Choice : INTEGER;
begin
  ...
  if ORD(Guess_Color) = (Choice - 1) then
```

The Translator produces the Modula-2 code shown below. In this case, you can ignore the comments before the assignment "GuessColor := Red;". The translation is correct and the Modula-2 compiler will not flag an error. The compiler finds a type incompatibility in the expression at the IF statement. The **ORD(GuessColor)** returns a **CARDINAL**, while **(Choice-1)** returns an **INTEGER**.



```

(* Modula-2 translation *)
...
TYPE
  Color = (Red, Blue, Green, Yellow);
VAR
  GuessColor: Color;
VAR
  Choice: INTEGER;
BEGIN
(* !!! These variables were defined in Turbo Pascal as 'typed constants' *)
(*   and were initialized in the program declaration part           *)
(* !!! In Modula-2 'typed constants' become variables and are initialized *)
(*   in the module or procedure body part                           *)
  GuessColor := Red;
  ...
  IF ORD(GuessColor) = (Choice-1) THEN
  ...

```

The solution to the above problem is to convert the logical expression into:

```
ORD(GuessColor) = (Choice-1) into INTEGER(ORD(GuessColor)) = (Choice-1)
```

In other cases, you could have a variable or a constant defined as a data type, different from the one expected as a parameter in a procedure call. You would also have to cast the parameter, as follows:

```

(* Pascal *)
...
i := Random(MemW[$FFFF:$00]);
...

(* Modula-2 translation *)
...
i := RandomInt( MemWGet(OFFFFH, 000H));
...

```

The compiler will find two errors. First, 'RandomInt' expects an INTEGER while 'MemWGet' returns a WORD. Second, 0FFFFH is a hexadecimal constant with a value greater than 32767 so it is considered a CARDINAL, while 'MemWGet' expects only INTEGERS. The manual modification of the code will be as follows:

```
(* Modula-2 manual modification *)
...
i := RandomInt( INTEGER(MemWGet( INTEGER(0FFFFH), 000H)));
...
```

If you define a variable as a subrange, and you pass it as a VAR parameter in a procedure, you can generate some incompatibilities. For example:

```
(* Pascal *)
...
var keys: 0..100;
begin
  ...
  read(keys);
  ...
```

```
(* Modula-2 *)
...
VAR keys: [0..100]
BEGIN
  ...
  ReadInt(stdinput, keys);
  ...
```

^ The compiler will produce an error message for type incompatibility

The solution of 'casting' keys with the type function INTEGER() will not work because Modula-2 does not allow you to cast a VAR parameter. The following code is therefore incorrect:

```
ReadInt(stdinput, INTEGER(keys));
```

The correct solution is to introduce a temporary variable to hold the result of the procedure and later assign it to the actual variable. For example:

```
(* Modula-2 *)
...
VAR keys: [0..100]
    tempKeys: INTEGER;
...
BEGIN
    ...
    ReadInt(stdout, tempKeys);
    keys := tempKeys;
    ...

```

### 2.7.2 Reals

Reals are supported in Modula-2 as in Turbo Pascal. For a detailed description of the the Real data type, please refer to the chapter on Real Arithmetic in the [LOGITECH MODULA-2/86 User's Guide](#). If your programs use Reals, we strongly suggest that you study that chapter very carefully to take advantage of the powerful features LOGITECH provides for Reals, such as 8087/80287 support and mixed mode libraries.

You should pay particular attention when using a INTEGER/CARDINAL in REAL expressions or using a REAL in INTEGER/CARDINAL expressions. The Translator will NOT take care for automatic casting, so the compiler will complain about type incompatibility. To cast this data type, you cannot use the type identifier, but instead you should use the functions 'Float', 'Trunc' and 'Round' (from module FloatingUtilities). You can call these functions with both positive and negative numbers, therefore these are more powerful than the standard Modula-2 functions FLOAT(i) and TRUNC(r) which allow only positive numbers.

```
(* Pascal *)
...
var i: integer;
    r: real;
begin
    ...
    r := sqrt(r);
    r := sqrt(i);
    i := sqrt(r);
    i := sqrt(i);

    r := hi(r);
    r := hi(i);
    i := hi(r);
    i := hi(i);

(* Modula-2 with manual modification *)
...
VAR
    i: INTEGER;
    r: REAL;
BEGIN
    ...
    r := sqrt(r);
    r := sqrt(Float(i));
    i := Trunc(sqrt(r));
    i := Trunc(sqrt(Float(i)));

    r := Float(hi(Trunc(r)));
    r := Float(hi(i));
    i := hi(Trunc(r));
    i := hi(i);
```

Further examples on mixed REAL and integer expressions are in files `EXPR.PAS` and `EXPR.MOD`.

### 2.7.3 Bytes

The predefined Turbo Pascal 'Byte' data type is translated into the Modula-2 'BYTE' type imported from module SYSTEM.

This means that BYTE, as well as WORD, ADDRESS and other data types defined in SYSTEM, is 'system dependent'. In other words, programs that use this data type could face portability problems if moved to other systems where the implementation of such types is done differently.

Like the Turbo Pascal 'Byte' type, the Modula-2 BYTE occupies 8 bits. However 'BYTE' cannot be involved in basic math operations (+, -, \*, DIV, MOD, ...). A variable of type BYTE can only be assigned or passed as a parameter. If math operations are required either the user defines another data type as explained later, or the code involved in the operation with BYTE should be changed using the cast function.

```
(* Pascal *)
...
var
  a,b,c: byte;
  ch: char;
begin
  a := $10;
  b := ch;
  c := 10;
  a := b + c;
  ch := a;
  ...

(* Modula-2 translation *)
...
VAR
  i: INTEGER;
  a, b, c: BYTE;
  ch: CHAR;
```

```

BEGIN
  a := BYTE(010H);
  b := BYTE(ch);
  c := BYTE(10);
  a := b+c;          (* the compiler will find errors here *)
  ch := a;          (* the compiler will find errors here *)

```

```
(* Modula-2 manually modified *)
```

```
...
```

```
VAR
```

```

i: INTEGER;
a, b, c: BYTE;
ch: CHAR;

```

```
BEGIN
```

```

a := BYTE(010H);
b := BYTE(ch);
c := BYTE(10);
a := BYTE(CHR(ORD(CHAR(b))+ORD(CHAR(c))));(* MODIFIED *)
ch := CHAR(a);          (* MODIFIED *)

```

If you use the type 'Byte' NOT because you want to reserve 8 bits memory space, but because your programs need a data type with range 0..255, you can declare your own type byte such as 'TYPE Byte = [0..255];' to perform the all the math operations. However, this 'Byte' will use a 16 bit representation and cannot be properly used in byte mapping and other 8-bit oriented, low level operations.

## 2.8 WITH and CASE

### 2.8.1 WITH

Turbo Pascal and Modula-2 implement records in a similar way. Turbo Pascal enables the 'WITH' construct to include one or more record names, while Modula-2 permits one record name for each single 'WITH' construct. The Translator takes care of breaking down the Turbo Pascal 'WITH' into a series of constructs.

```
(* Pascal *)
type
  Home_Address = RECORD
      Street, City : string[30];
      State       : string[2];
      Zip         : string[9];
  END;
  Phone_Numbers = RECORD
      Home_Number,
      Work_Number : string[7];
  END;
  Personal_Info = RECORD
      Name : string30;
      Home : Home_Address;
      Phone : Phone_Numbers;
  END;

var
  Person : Personal_Info;

begin
  with Person, Home, Phone do begin
    write('Enter name ');
    readln(Name); writeln;
    write('Enter street address ');
    readln(Street); writeln;
    ...
  end
  ...
end
```

The Translator generates the following code with no editing required:

```
(* Modula-2 translation *)
...
TYPE
  HomeAddress = RECORD
    Street, City: ARRAY [0.. 30-1] OF CHAR;
    State: ARRAY [0.. 2-1] OF CHAR;
    Zip: ARRAY [0.. 9-1] OF CHAR;
  END;

  PhoneNumbers = RECORD
    HomeNumber, WorkNumber: ARRAY [0.. 7-1] OF CHAR;
  END;

  PersonalInfo = RECORD
    Name: ARRAY [0.. 30-1] OF CHAR;
    Home: HomeAddress;
    Phone: PhoneNumbers;
  END;

VAR
  Person: PersonalInfo;

BEGIN
  WITH Person DO
    WITH Home DO
      WITH Phone DO
        WriteString(stdinout, 'Enter name ', 0);
        ReadString(stdinout, Name);
        ReadLn(stdinout);
        WriteLn(stdinout);
      ...
    END
  END
END
...
```



## 2.8.2 CASE

Turbo Pascal supports the catch-all ELSE clause, not found in standard Pascal. If there is no ELSE clause and the examined identifier has a value that does not lie in the range of any CASE option, the program simply resumes after the CASE statement. The above condition generates a run-time "out-of-range" error in Modula-2 when range test is ON, thus the ELSE clause is mandatory. The Translator will insert it when a Turbo Pascal CASE construct with no ELSE clause is encountered. The inserted ELSE contains a do-nothing empty statement.

## 2.9 Absolute Statements and Untyped Variables

Both Turbo Pascal and Modula-2 support variables in a very similar way. There are a few exceptions as discussed below.

Modula-2 supports absolute variables in a different way than Turbo Pascal. Like Turbo, Modula-2 lets you define the address constant where you want the variable to be located, but WITHOUT the possibility to use standard identifiers (CSeg, DSeg ...) or to define a variable 'on top' of another variable (i.e. they both start at the same address).

The Translator does not perform any automatic translation on absolute variables, instead it will mark all the occurrences of absolute variables and the user must manually translate them.

For simple cases, absolute variables in Turbo Pascal can be hand translated by the user. For example, if we declare the following in Turbo Pascal:

```
VAR Abs_Name : STRING[80] Absolute $0000:$00EE;
```

it can be written in Modula-2 as,

```
VAR AbsName [00H:EEH] : ARRAY[0..79] OF CHAR;
```

Note that in Modula-2 the location is defined after the variable name, but before the variable type, and the address can only be expressed by constants.

The following is a slightly more complex Turbo Pascal declaration involving a record structure:

```
VAR Info = RECORD
    Name : STRING[30];
    Address, City : STRING[40];
    State : STRING[2];
    Zip : STRING[9] Absolute $55;
END;
```

In Modula-2 the equivalent declaration becomes:

```
VAR Info = RECORD
    Name : ARRAY[0..29] OF CHAR;
    Address, City : ARRAY[0..39] OF CHAR;
    State : ARRAY[0..1] OF CHAR;
    Zip[0H:55H] : ARRAY[0..8] OF CHAR;
END;
```

More advanced types of absolute variables, where a reference segment address is used (Cseg, Dseg ...), are not supported by Modula-2.

Turbo Pascal allows absolute variables to be 'overlaid' using other variable names as addresses.

For example:

```
Name : STRING[9];
Long : Byte Absolute Name;
BEGIN
    if Long = $FF then ...
```

The equivalent in Modula-2 requires you to rewrite part of the code because of the change in data type introduced:

```
Name : ARRAY [0..8] OF CHAR;
Long : POINTER TO BYTE; (* use a POINTER notation *)
BEGIN
    Long^ := ADR(Name); (* loads the pointer with the address of Name *)
                    (* this is a new statement to initialize Long *)
    IF Long^ = FFH THEN ...
```

The same technique can be used in procedures with Untyped Variable Parameters:

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

(* Modula-2 translation *)
PROCEDURE SwitchVar(VAR A1p, A2p: ARRAY OF BYTE;
                   Size: INTEGER);

  TYPE
    A = ARRAY [1..MaxInt] OF BYTE;
  VAR
    A1: A ?6(* absolute[A1p ] *) ;    (* <--- to be fixed !!!! *)
    A2: A ?6(* absolute[A2p ] *) ;    (* <--- to be fixed !!!! *)
    Tmp: BYTE;
    Count: INTEGER;
  BEGIN
    FOR Count := 1 TO Size DO
      Tmp := BYTE(A1[Count]);
      A1[Count] := A2[Count];
      A2[Count] := Tmp;
    END;
  END SwitchVar;
```

```

(* Modula-2 user modified version *)
PROCEDURE SwitchVar(VAR A1p, A2p: ARRAY OF BYTE;
                   Size: INTEGER);

TYPE
  A = ARRAY [1..MaxInt] OF BYTE;
VAR
  A1: POINTER TO A;          (* MODIFIED ... POINTER TO ... *)
  A2: POINTER TO A;          (* MODIFIED ... POINTER TO ... *)
  Tmp: BYTE;
  Count: INTEGER;
BEGIN
  A1 := ADR(A1p);            (* MODIFIED ... initialization ... *)
  A2 := ADR(A2p);            (* MODIFIED ... initialization ... *)

  FOR Count := 1 TO Size DO
    Tmp := BYTE(A1^[Count]); (* MODIFIED ... A1 -> A1^ ... *)
    A1^[Count] := A2^[Count]; (* MODIFIED ... A1 -> A1^, A2 -> A2^ ... *)
    A2^[Count] := Tmp;       (* MODIFIED ... A2 -> A2^ ... *)
  END;
END SwitchVar;

```

## 2.10 Inline Machine Code

LOGITECH MODULA-2/86 supports inline machine code statements as in Turbo Pascal. In Modula-2, you use the standard procedure 'CODE' (module SYSTEM) with a different syntax and few limitations, instead of Turbo Pascal 'inline'. The Translator takes care to generate correct CODE statements, separated by commas. CODE does not support the special symbols '>' or '<' available in Turbo Pascal, but the Translator provides a correct translation for them.

The parameters for the CODE are limited to constant expressions with values less than 255. Therefore, it is not possible to directly translate variable identifiers, procedure identifiers, function identifiers, or a location counter reference in the Turbo Pascal inline statement. In Turbo Pascal, this use of identifiers puts the offset of the item, relative to its base segment, into the code. In Modula-2 the same offset value must be passed as a constant.

To determine the offset of variables, you can use the decoder utility 'M2DECOD' or evaluate them by hand, following the description in the LOGITECH MODULA-2/86 User's Guide about the memory organization.

For function/procedure identifiers or references to the location pointer, only an iterative step using the decoder can solve the problem. This solution is very inflexible and dangerous. After any change of the module, or even a recompilation with a different compiler version, the correctness of the CODE procedure must be checked.

If your inline code is used to refer to a variable to be loaded into a register, instead of using inline code you can use the LOGITECH MODULA-2/86 standard procedures GETREG and SETREG from module SYSTEM. These procedures allow you to set the value of a register from an expression or to get the value of a register into a variable. For more detailed information, refer to the LOGITECH MODULA-2/86 User's Guide. An example of inline code using reference to a variable is:

```
(* Pascal *)
...
var somevar: integer;
begin
  ...
  inline ($8B/$46/<somevar>); (* MOV AX, somevar[BP] *)
  ...

(* Modula-2 *)
...
FROM SYSTEM IMPORT AX, SETREG;
...
VAR somevar: INTEGER;
BEGIN
  ...
  SETREG(AX, somevar); (* MOV AX, somevar[BP] *)
  ...
```

## 2.11 Turbo Pascal Predefined Variables

In Turbo Pascal, the user can access some predefined variables to perform system related operations. Some of these variables are available in LOGITECH MODULA-2/86 either as variables or as procedures.

Predefined files (input, output, con, trm, aux, lst, kbd, usr) are defined as File variables in module TKernelIO. The variable stdinout is used in all Read/Write operations that do not use a system or user file variable (i.e readln(ch)).

### 2.11.1 User I/O Drivers

As in Turbo Pascal, the user can assign the address of self-defined driver procedures to procedure variables defined in the module TKernelIO. The Translator will correctly translate the Pascal code, but some manual modification should be applied to avoid compiler errors.

In Turbo Pascal the standard variables which contain the address of driver procedures (ConStPtr, ConInPtr, ...) are defined as integers. Instead, in Modula-2, we use the more correct approach of defining them as PROCEDURE VARIABLES (see module TKernelIO in file TKERNELI.DEF). Thus, the loading of these variables should be changed to conform to the new data type.

The user defined procedure to be installed as driver should be compatible with the procedure type definition of the variable. For example the procedure to be loaded in 'conStPtr' should be of type 'StatusProc', that is a function returning a BOOLEAN, while the procedure for 'auxOutPtr' should be of type 'WriteProc', that is a procedure with a CHAR parameter passed by value.

Here is a sample of translation:

```
(* Pascal *)
...
function UserIn: char;
var ch: char;
begin
  read(ch);
  userin := ch;
end;

function UserOut(ch: char);
begin
  write(ch);
end;
...
```

```

begin
  ...
  usrinptr := ofs(UserIn);
  usROUTptr := ofs(UserOut);
  ...

(* Modula-2 translation *)
...
FROM MemoryOperations          (* there is no need for this import *)
  IMPORT FillChar, Move, Hi, ... (* if the call to 'Ofs' procedure is *)
  ...                          (* removed in the body          *)
  ... ...   Dseg, SSeg, MemAvail;
...

PROCEDURE UserIn(): CHAR;
VAR
  ch: CHAR;
VAR UserInResult:CHAR;          (* this procedure could be optimized *)
BEGIN
  ReadBuffer(on);
  ReadChar(stdinput, ch);
  ReadBuffer(off);
  UserInResult := ch;
  RETURN UserInResult;
END UserIn;

PROCEDURE UserOut(ch: CHAR);
BEGIN
  WriteChar(stdoutput, ch, 0);
END UserOut;
...
BEGIN
  ...
  usrinPtr := Ofs(ADDRESS(UserIn())); (* to be modified *)
  usROUTPtr := Ofs(ADDRESS(UserOut)); (* to be modified *)
  ...

```

```

(* Modula-2 user modified version *)
...
  (* import removed *)
...

PROCEDURE UserIn(): CHAR;
VAR
  ch: CHAR;
BEGIN
  (* here we have performed some optimization *)
  ReadBuffer(on);          (* removing the local variable UserInResult *)
  ReadChar(stdinout, ch);
  ReadBuffer(off);
  RETURN ch;
END UserIn;

PROCEDURE UserOut(ch: CHAR);
BEGIN
  WriteChar(stdinout, ch, 0);
END UserOut;

...
BEGIN
  ...
  usrInPtr := UserIn;      (* this is the correct initialization *)
  usrOutPtr := UserOut;    (* of procedure variables      *)
  ...

```

### 2.11.2 ErrorPtr

Turbo Pascal allows you to install a user defined error handler to be called in case of I/O or Run Time errors. The same feature is available with LOGITECH MODULA-2/86 using a global variable exported from module TKernelIO.

In Turbo Pascal the variable that contains the address of the error handler procedure (errorPtr) is defined as an integer. Instead, in Modula-2, we use the more correct approach of defining it as a PROCEDURE VARIABLE (see module TKernelIO in file TKERNELI.DEF). Thus, the loading of this variable should be changed to conform to the new data type.



The user defined procedure to be installed as an error handler should of type 'ErrorProc = PROCEDURE( INTEGER, INTEGER );', that is, a procedure with two parameters, both integers and both passed by value.

**First parameter:** The first parameter passed is the error type and number. The most significant byte, contains the error type, and the least significant byte, contains the error number. The error numbers (Low Byte of first parameter) are the same as described in Appendices F and G of the Turbo Pascal Manual with the exception of the following run-time errors:

```
03 Sqrt argument error
04 Ln argument error
10 String length error
11 Invalid string index
90 Index out of range (* in Modula-2 run time error 90 is mapped *)
                          (* into run time error 91          *)
F0 Overlay not found
```

These error numbers will never be generated by the LOGITECH MODULA-2/86 Run Time System. Other Modula-2 run time errors (see type Status in module System) are passed to the error handler procedure as code F1.

The following error types (High Byte of first parameter) are defined:

```
0 User Break (Ctrl-C).
1 I/O error.
2 Run-time error.
```

In case of a user interrupt (Ctrl-C), the Low Byte is always 1.

**Second parameter:** The second parameter has been declared for completeness of translation, but is not used and therefore has no relevance.

**NOTE ON TERMINATION ROUTINE**

**Error Type 0 and 1:** If the implementation of the user error handler includes the 'halt' statement (i.e. translated into 'Terminate(normal)') at run time the execution will be as follow:

```
...  
User Break or I/O error occurs  
User Error handler and Terminate(xxx) executed  
Modula-2 Run Time System epilog  
control back to the O.S.
```

If the implementation of the user error handler DOES NOT include the 'halt' statement (i.e. translated into 'Terminate(normal)') at run time the execution will be as follows:

```
...  
User Break or I/O error occurs  
User Error handler executed  
Files Library Error handler executed (* will write a message on screen *)  
Modula-2 Run Time System epilog  
control back to the O.S.
```

**Error Type 2:** In case of a Run Time error the LOGITECH MODULA-2/86 will first generate a Memory Dump (to be analyzed by the Post Mortem Debugger) and after will call the Termination routines.

To avoid a recursive call to the Terminate routine, the user defined error handler should not have any further call to Terminate (in case of error type 2).

T-Pascal	Modula - 2
PROCEDURE Error ( ... );	PROCEDURE Error ( ... );
BEGIN	BEGIN
..	..
CASE errorType OF	CASE errorType OF
0 {ctrlC} : ... ;	0 (*ctrlC*) :
	..
	System.Terminate(normal);
1 {IOError} : ... ;	1 (*IOError*) :
	..
	System.Terminate(normal);
2 {RunTime} : ... ;	2 (*RunTime*) :
	... ;
END;	END;
Halt;	
END;	END Error;

LOGITECH MODULA-2/86 allows to install both Initialization and Termination routines (see module System in the LOGITECH MODULA-2/86 User's Guide). These routines allow the user to install procedures that are called before and/or after the execution of a program.

Here is a sample of translation:

```
(* Pascal *)
...
procedure error ( errno, erraddr : integer );
var
  errtype : integer;
```

```

begin
  errtype := errno div 256;
  errno   := errno mod 256;
  case errtype of
    0 : writeln( ' User Break (Ctrl - C) ' );
    1 : begin
          writeln( ' I/O error ' );
          writeln( ' error number := ', errno );
        end;
    2 : begin
          writeln( ' Run - time error ' );
          writeln( ' error number := ', errno );
        end;
  end;
  halt;
end;
...
begin
  ...
  errorptr := ofs( error );
  ...

(* Modula-2 translation *)
...
FROM MemoryOperations          (* there is no need for this import *)
  IMPORT FillChar, Move, Hi, ... (* if the call to 'Ofs' procedure is *)
  ...                          (* removed in the body          *)
  ... ...   Dseg, SSeg, MemAvail;
...
PROCEDURE error(errno, erraddr: INTEGER);
  VAR
    errtype: INTEGER;

```

```

BEGIN
  errtype := errno DIV 256;
  errno := errno MOD 256;
  CASE errtype OF
    0:
      WriteString(stdout, ' User Break (Ctrl - C) ', 0);
      WriteLn(stdout);
    | 1:
      WriteString(stdout, ' I/O error ', 0);
      WriteLn(stdout);
      WriteString(stdout, ' error number := ', 0);
      WriteInt(stdout, errno, 0);
      WriteLn(stdout);
    | 2:
      WriteString(stdout, ' Run - time error ', 0);
      WriteLn(stdout);
      WriteString(stdout, ' error number := ', 0);
      WriteInt(stdout, errno, 0);
      WriteLn(stdout);
  ELSE
    END;
  Terminate(normal);  (* !!! NOTE: this call should be removed from *)
                      (* here and placed ONLY for case 0 and 1      *)
                      (* in case 2 a correct termination is handled *)
                      (* by the LOGITECH MODULA-2/86 Run Time System *)

  END error;
...
BEGIN
  ...
  errorPtr := Ofs(ADDRESS(error));  (* to be modified *)
  ...

(* Modula-2 user modified version *)
...
(* import removed *)
...
PROCEDURE error(errno, erraddr: INTEGER);

```

```

VAR
  errtype: INTEGER;
BEGIN
  errtype := errno DIV 256;
  errno := errno MOD 256;
  CASE errtype OF
    0:
      WriteString(stdout, ' User Break (Ctrl - C) ', 0);
      WriteLn(stdout);
      Terminate(normal);          (* here is the right place *)
  | 1:
      WriteString(stdout, ' I/O error ', 0);
      WriteLn(stdout);
      WriteString(stdout, ' error number := ', 0);
      WriteInt(stdout, errno, 0);
      WriteLn(stdout);
      Terminate(normal);          (* here is the right place *)
  | 2:
      WriteString(stdout, ' Run - time error ', 0);
      WriteLn(stdout);
      WriteString(stdout, ' error number := ', 0);
      WriteInt(stdout, errno, 0);
      WriteLn(stdout);
  ELSE
    END;
                                (* Terminate removed *)

  END error;
...
BEGIN
  ...
  errorPtr := error;  (* this is the correct initialization *)
                   (* of procedure variables          *)
  ...

```

### 2.11.3 Mem, MemW, Port and PortW

The Turbo Pascal predefined variables Mem, MemW, Port and PortW are supported in LOGITECH MODULA-2/86 by functions and procedures (module MemoryOperations). Any reference to these variables is automatically translated into the equivalent procedure call.

The list of these procedures includes:

- MemGet : to load a byte from the Mem array into a variable
- MemSet : to assign a byte from a variable to the Mem array
- MemWGet : to load a word from the MemW array into a variable
- MemWSet : to assign a word from a variable to the MemW array
  
- PortGet : to load a byte from the Port array into a variable
- PortSet : to assign a byte from a variable to the Port array
- PortWGet : to load a word from the PortW array into a variable
- PortWSet : to assign a word from a variable to the PortW array

Examples of translations are:

```
(* Pascal *)
```

```
Ptr := Mem[0000:$0081];
```

```
(* Modula-2 *)
```

```
Ptr := MemGet(0, 81H);
```

```
(* Pascal *)
```

```
MemW[ Seg(Ptr) : Ofs(Ptr)] := Ptr_Location;
```

```
(* Modula-2 *)
```

```
MemWSet(Seg(Ptr), Ofs(Ptr), PtrLocation);
```

```
(* Pascal *)
```

```
ch := Port[40];
```

```
(* Modula-2 *)
```

```
ch := PortGet(40);
```

```
(* Pascal *)
PortW[$56] := 10;

(* Modula-2 *)
PortWSet(56H, 10);
```

Note: An alternative solution to the use of Mem or MemW might be the use of variables at absolute addresses. Instead of Port or PortW, one could also use the procedures INBYTE, OUTBYTE, INWORD, or OUTWORD, of module SYSTEM.

#### 2.11.4 HeapTop, Mark and Release

The variable 'HeapPtr' is translated into the variable 'curProcess<sup>^^</sup>.heapTop' of module System. We suggest that you study both module System and SYSTEM (yes, they are different!) to know more about system dependent functions and variables available with LOGITECH MODULA-2/86.

Mark and Release functions are translated into InstallHeap and RemoveHeap from module Storage. These Modula-2 procedures work in a balanced way, in other words, ReleaseHeap releases the last heap installed by InstallHeap. Unbalanced use of these procedures produces unpredictable errors. Revise your code carefully before running it.

For more details on LOGITECH MODULA-2/86 memory organization, refer to the chapter in the MODULA-2/86 User's Guide.

#### 2.12 I/O Operations

The Modula-2 language does not define any I/O or file management operations. All I/O is performed using procedures imported from library modules. With a Modula-2 system you get a set of library modules that implements the I/O operations. The LOGITECH MODULA-2/86 Base Language System comes with such modules (for example, InOut, FileSystem, Terminal, Directories, DiskDirectories). An additional example of a library module implementing simple MS-DOS oriented file management routines is the module FileIO. FileIO is available as example in full source format (file FILEIO.DEF FILEIO.MOD) in the Translator diskette.



With the Translator, you receive an additional set of Modula-2 library modules, some of which implement I/O operations similar to the ones already performed by modules of the Base Language System, but with the same interface as used in Turbo Pascal. For example, to open a file, instead of learning the specifications of procedure 'Lookup' from 'FileSystem', you can use the more familiar (to Turbo Pascal users) sequence of procedures 'Assign' and 'Reset' from module 'TFileIO'.

Screen oriented operations are defined in module ScreenHandler.

File management procedures are defined in five modules:

- TKernelIO  
File type definition, predefined files, device I/O, compiler directives, error handler pointer
- TFileIO  
Operation on the entire file (all kinds of files)
- TTextIO  
Formatted I/O (text files)
- TRealIO  
Formatted I/O for Reals (text files)
- TBinaryIO  
Operations on files of records, and untyped files

The additional module 'CMMNFiles' contains internal common data structures and procedures used by the previous modules, but not accessible to the end user.

This set of library modules supports the Turbo Pascal's three kinds of files:

- Text file, equivalent to FILE OF CHAR.
- File of records, equivalent to binary file.
- Untyped files, also equivalent to binary file.

The Translator takes care to generate the correct Modula-2 procedure calls for each Turbo Pascal file management operation. The following three different file types are mapped into the unique file type 'File' from 'TKernelIO'. The Translator keeps track of the parent Turbo Pascal file type in further operations. It also takes care to produce correct Modula-2 procedure calls in the presence of Turbo Pascal calls with a variable number of parameters. For more details, refer to Appendix A with the complete procedure mapping schemata used by the Translator.

- When reading from the standard input, all the editing facilities available in Turbo Pascal (ESC, DEL, BACKSPACE, Ctrl-D, Ctrl-X, ...) are supported.
- Read and Write operations on standard input/output files are translated using the predefined standard input/output file variable 'stdinout'.
- Consecutive Read operations from the standard input file should be preceded by 'ReadBuffer(on);' and terminated by 'ReadBuffer(off);' to ensure correct buffering. For example:

```
(* Pascal *)
...
var a, b: integer;
    c: char;
begin
  read(a,b,c);          (* using standard input *)
  ...

(* Modula-2 *)
...
VAR a, b: INTEGER;
    c: CHAR;
BEGIN
  ReadBuffer(on);
  ReadInt(stdinout, a);
  ReadInt(stdinout, b);
  ReadChar(stdinout, c);
  ReadBuffer(off);
  ...
```

The following demonstrates the use of files. In the distribution diskette you will find three example programs using files written in Turbo Pascal and their equivalent translated and modified versions in Modula-2 (files WORD.PAS, WORD.MOD, PAGE.PAS, PAGE.MOD, LOGITEL.PAS, LOGITEL.MOD).

The Turbo Pascal program 'word' requests the user for a text filename and a number of words to be searched for in that file. The program displays the text file and counts the number of times the words occur in the text. The Translator will translate the program correctly, but because the program uses the function Pos (see section on Strings), you need to modify it. The Modula-2 file WORD.MOD contains the modified version with comments inserted to indicate where the program was edited.

The Turbo Pascal program 'page' prints paginated textfiles. The program prompts the user for the filename and the number of lines per page. The version produced by the Translator (file PAGE.MOD) is correct.

The next example deals with a larger program. The program 'logitel' is a very simple motel management system. It demonstrates the use of typed binary files and screen control. The Logitel motel has one hundred rooms. The clerk sitting at the main desk uses this program to check guests in and out, to add expenses and to look at the status of a particular room. The Translator generates a Modula-2 version that requires few modifications.

In Turbo Pascal null string assignments are performed by statements such as:

```
(* Pascal *)  
...  
Name := '';  
...  
if Filename = '' then ...  
...
```

The equivalent Modula-2 form is as follows:

```
(* Modula-2 *)  
...  
Name[0] := 0C;  
...  
IF Filename[0] = 0C THEN ...  
...
```

The Modula-2 file LOGITEL.MOD contains the modified version with comments inserted to indicate where the program was edited.

### 2.13 Bit Manipulation Operators 'AND', 'OR', 'NOT', 'XOR', 'SHR', 'SHL'

Turbo Pascal allows the use of the operators AND, OR, NOT, XOR both for logical operations on boolean operands and for arithmetic operations on integer operands. These operators perform the logical operation on each single bit of the operands. For example:

```
(* Pascal *)
...
var i, j, k: integer;
begin
  i := 0;           (* i is 0 *)
  j := not -15;    (* j is 14 *)
  k := not i;      (* k is -1 *)
  i := 12 and 22;  (* i is 4 *)
  j := j or 7;     (* j is 15 *)
...

```

The Modula-2 language supports the operators AND, OR, NOT only for logical operations on boolean operands. The logical operator XOR can be substituted with a sequence of AND, OR, NOT.

$(a \text{ XOR } b)$  is equivalent to  $((\text{NOT } a \text{ AND } b) \text{ OR } (a \text{ AND NOT } b))$

The use of AND, OR, NOT, XOR for bitwise arithmetic operations on integer operands is not allowed in Modula-2. To overcome this limitation, these operators are implemented as functions 'And', 'Or', 'Not', 'Xor' in module MemoryOperations. In the same module, you will find two additional bitwise functions 'Shl' and 'Shr' to completely support the Turbo Pascal functionalities.

When the Translator encounters the operators XOR, SHL, SHR not available in Modula-2, it will mark them with the flag ?9. If you use AND, OR, NOT as arithmetic operators the Translator will not mark them, but the compiler will issue a compiler error.

If, in your Pascal program, you are using the operator XOR as a logical operator, you will have to modify your program as explained before. If you are using the operators for bit manipulation, you need to make changes on the translated version to generate a correct Modula-2 program.

```
(* Pascal *)
...
var i, j, k: integer;
begin
  ...
  i := (i and j) or (k shr 3);
  ...

(* Modula-2 *)
...
FROM MemoryOperations IMPORT And, Or, Shr;
...
VAR i, j, k: INTEGER;
BEGIN
  ...
  i := Or(And(i,j),Shr(k,3));
  ...
```

## 2.14 Functions and Procedures

Functions and procedures are very similar in Turbo Pascal and Modula-2. The following sections note a few important points:

### 2.14.1 Result of Functions

Modula-2 has dropped the keyword 'FUNCTION' and replaced it with 'PROCEDURE'. In addition, Modula-2 does not use the function name to hold the return value, instead a local variable (with the same function data type) must be declared in the procedure body and explicitly returned when leaving the procedure. The Translator automatically creates a local variable using the "<functionName>Result" name convention and generates the appropriate "RETURN <functionName>Result" both at the end of the procedure body and in all the places where the Turbo Pascal statement 'Exit' is called.

### 2.14.2 Exit

In Turbo Pascal, this statement is used to exit a routine or a statement block (in the main program). The Translator converts it according to context. A Turbo Pascal 'Exit' in a function is translated into a 'RETURN' followed by the return value, otherwise the Translator produces a simple 'RETURN' statement.

### 2.14.3 Halt

In Turbo Pascal, the 'Halt' statement leads back to DOS or to the main Turbo Pascal command menu. In Modula-2 'HALT' first performs a post-mortem dump and then exits to DOS. The Translator transforms the call 'Halt;' into the more correct statement 'Terminate(normal);' while it translates 'Halt(x);' into 'SetErrorCode(x);' followed by 'Terminate(normal);'.

### 2.14.4 Forward Declarations

A FORWARD declaration is not needed in Modula-2. The language definition allows you to declare a procedure heading and its block before the declaration of a procedure you are calling. Modula-2 automatically handles mutual calling routines. Thus, the 'FORWARD' reference is removed and the 'shorthand' routine heading at the body is replaced by the full heading declaration.

## 2.15 Overlay, Chain and Execute

Modula-2 programs are not limited to 64K of code. LOGITECH MODULA-2/86 allows you to build a program with total maximum size of one Megabyte (code plus data). For more details on module and procedure code and data sizes, refer to the chapter on Memory Organization in the LOGITECH MODULA-2/86 User's Guide.

### 2.15.1 Overlay

Because of these high limits the overlay declaration is ignored by the Translator and is transformed into a comment. Thus all the procedures will be linked in one executable file (.LOD).

If your application still needs an overlay mechanism, you can use the LOGITECH MODULA-2/86 Overlay System. You will find detailed information in the chapter Memory Organization in the LOGITECH MODULA-2/86 User's Guide and in the definition module Program.

**2.15.2 Chain and Execute**

These routines were created in Turbo Pascal to overcome the 64K code limitations.

The procedure Chain is not supported by the Translator, so a flag ?1 will be generated. The user needs to modify the programs to take advantage of the ability to generate large code with LOGITECH MODULA-2/86.

The procedure Execute is supported by the Translator which generates a call to Execute from module TExec. The Modula-2 implementation of Execute is more powerful than the Turbo Pascal implementation, because it allows you to run any PC-DOS/MS-DOS executable program (.COM, .EXE). The module TExec implements a Turbo Pascal compatible version of Execute while a more complete interface to run DOS programs is available from module Exec. If you need to run Modula-2 subprograms/overlay (.LOD) from your main program you should refer to module Program.

### 3 ADVANCED SOFTWARE ENGINEERING USING MODULA-2

Modula-2 allows you to implement your application using advanced software engineering techniques. These techniques include modular software development using library modules, concurrency using coroutines, data abstraction using opaque types and procedures passed as parameters. To optimize the benefits of translating your Pascal programs into Modula-2, we strongly suggest that you study these features of Modula-2 in an introductory book on the language. (There is a bibliography of books on Modula-2 in an Appendix in your MODULA-2/86 User's Guide.) If you invest the time to learn these techniques, you will develop better modular code and hence, save development time.

A complete and working example of a library module using some of these advanced features is in the module FileIO (files FILEIO.DEF, FILE.MOD). This module implements an interface to the MS-DOS File System and can be used in your application when you need simple and fast file management routines.

In this chapter we focus on two software engineering techniques: Library Modules and Opaque Types. The source code of the sample programs used in this chapter is on the diskettes.

#### 3.1 Creating Library Modules

Library modules are the most important feature in the Modula-2 language. In this section we discuss and demonstrate the creation of such libraries from translated Turbo Pascal programs. We take a number of different approaches with the same example to show how each approach works, and how it differs from the other approaches.

##### 3.1.1 From a Turbo Pascal Program to a Single MODULA-2/86 Module

The example we provide deals with a four-function calculator, handling complex numbers (composed of real and imaginary parts). The program requests that the user enter a basic operation, or the letter 'Q' to quit. Upon typing an operation symbol, the program requests the real and imaginary parts of the two operands and displays the result. The Translator detects if the user tries to divide by a zero complex number and consequently, displays an error message. If the user types a character that is not in the set ('Q','+','-','\*','/') then the Translator performs the complex addition.



Assume that we have written the library of complex operations as part of a complete Turbo Pascal program. After testing it thoroughly, we want to translate it into a Modula-2 program and then build a library module. The Turbo Pascal source program is shown below.

The program defines the 'Complex' type as a record containing two 'REAL' fields that make up a complex number. The listing shows three procedures and a function to perform the four basic complex operations. Two additional procedures handle complex I/O.

```
in file COMPLEXC.PAS:

PROGRAM Complex_Calc;
(* Program to simulate a four-function complex calculator *)

TYPE Complex = RECORD
    Rel, (* Real part *)
    Imag (* Imaginary part *) : REAL;
END;

VAR C1, C2, C3 : Complex;
    Correct : BOOLEAN;
    Operation, Dummy : CHAR;

PROCEDURE Add_Complex(C1, C2 : Complex;    (* input *)
                    VAR Result : Complex (* output *));
(* Procedure to add two complex numbers *)

BEGIN
    Result.Rel := C1.Rel + C2.Rel;
    Result.Imag := C1.Imag + C2.Imag
END;

PROCEDURE Subt_Complex(C1, C2 : Complex;    (* input *)
                    VAR Result : Complex (* output *));
(* Procedure to subtract two complex numbers *)

BEGIN
    Result.Rel := C1.Rel - C2.Rel;
    Result.Imag := C1.Imag - C2.Imag
END;
```

```

PROCEDURE Mult_Complex(C1, C2 : Complex;   (* input *)
                      VAR Result : Complex (* output *));
(* Procedure to multiply two complex numbers *)

BEGIN
  Result.Rel := C1.Rel * C2.Rel - C1.Imag * C2.Imag;
  Result.Imag := C1.Rel * C2.Imag + C2.Rel * C1.Imag
END;

FUNCTION Div_Complex(C1, C2 : Complex;   (* input *)
                    VAR Result : Complex (* output *)) : BOOLEAN;
(* Function to divide two complex numbers and return TRUE *)
(* if operation is successful, FALSE for division by zero *)

VAR OK : BOOLEAN;
    SumSqr : REAL;

BEGIN
  IF (C2.Rel <> 0) OR (C2.Imag <> 0)
  THEN BEGIN
    OK := TRUE;
    SumSqr := SQR(C2.Rel) + SQR(C2.Imag);
    Result.Rel := (C1.Rel * C2.Rel + C1.Imag * C2.Imag)
                  / SumSqr;
    Result.Imag := (C2.Rel * C1.Imag - C1.Rel * C2.Imag)
                  / SumSqr
  END
  ELSE
    OK := FALSE;
    Div_Complex := OK;
  END;

PROCEDURE Read_Complex(VAR C : Complex (* output *));
(* Procedure to read complex number *)

```

```

BEGIN
  WRITE('Enter real part '); READLN(C.Rel);
  WRITE('Enter imaginary part '); READLN(C.Imag);
  WRITELN;
END;

PROCEDURE Write_Complex(C : Complex (* input *));
(* Procedure to output a complex number *)
BEGIN
  WRITELN('Complex number = ',C.Rel,' + i ',C.Imag)
END;

BEGIN (*----- MAIN -----*)
  REPEAT
    ClrScr;
    WRITE('Enter operation [Q = quit] ');
    READLN(Operation); WRITELN;
    Operation := UpCase(Operation);
    IF Operation <> 'Q'
    THEN BEGIN
      WRITELN('Enter first complex number');
      Read_Complex(C1);
      WRITELN('Enter second complex number');
      Read_Complex(C2); WRITELN;
      Correct := TRUE;
      CASE Operation OF
        '+' : Add_Complex(C1, C2, C3);
        '-' : Subt_Complex(C1, C2, C3);
        '*' : Mult_Complex(C1, C2, C3);
        '/' : BEGIN
          Correct := Div_Complex(C1, C2, C3);
          IF NOT Correct
          THEN WRITELN('Divide by zero error ');
        END
      ELSE Add_Complex(C1, C2, C3);
    END
  UNTIL Operation = 'Q';
END

```

```

END;
  IF Correct THEN Write_Complex(C3);
  WRITELN; WRITELN;
  WRITE('Press <CR> to continue ');
  READLN(Dummy); WRITELN; WRITELN;
END;
UNTIL Operation = 'Q';
END.

```

The Translator converts the above Turbo Pascal program into the following correct Modula-2 program. Note that the position of the comment is different in the Modula-2 program than in the Pascal program. If the new position is not acceptable, please modify the Modula-2 program accordingly. As a reminder, in Modula-2, a comment that starts with '(\*' and ends with \*)', can be placed at any place in the code and can be nested for many levels.

in file COMPLEXC.MOD:

```

MODULE ComplexCalc;

FROM FloatingUtilities IMPORT Frac, Int, Round, Float, Trunc;
FROM ScreenHandler
  IMPORT ClrEol, ClrScr, DelLine, InsLine, GotoXY, WhereX, WhereY,
  CrtInit, CrtExit, LowVideo, NormVideo, HighVideo, SetAttribute,
  GetAttribute, normalAtt, boldAtt, reverseAtt, underlineAtt,
  blinkAtt, boldUnderlineAtt, blinkUnderlineAtt, boldBlinkAtt,
  reverseBlinkAtt, boldUnderlineBlinkAtt;
FROM TRealIO IMPORT ReadReal, WriteReal;
FROM TTextIO
  IMPORT ReadInt, ReadCard, ReadChar, ReadString, ReadLn, ReadBuffer,
  WriteInt, WriteCard, WriteChar, WriteString, WriteBool, Writeln,
  Eoln, SeekEof, SeekEoln;
FROM TKernelIO
  IMPORT File, FileType, OptionMode, StatusProc, ReadProc, WriteProc,
  stdout, input, output, con, trm, kbd, lst, aux, usr,
  conStPtr, conInPtr, auxInPtr, usrInPtr, conOutPtr, lstOutPtr,
  auxOutPtr, usrOutPtr, errorPtr, IOresult, KeyPressed, IOBuffer,
  IOCheck, DeviceCheck, CtrlC, InputFileBuffer, OutputFileBuffer;

```

```
(* Program to simulate a four-function complex calculator *)
```

```
TYPE
```

```
Complex = RECORD
    Rel, (* Real part *)
    (* Imaginary part *) Imag: REAL;
END;
```

```
VAR
```

```
C1, C2, C3: Complex;
Correct: BOOLEAN;
Operation, Dummy: CHAR;
```

```
(* input *)
```

```
(* output *) (* Procedure to add two complex numbers *)
```

```
PROCEDURE AddComplex(C1, C2: Complex;
    VAR Result: Complex);
```

```
BEGIN
```

```
Result.Rel := C1.Rel+C2.Rel;
```

```
Result.Imag := C1.Imag+C2.Imag
```

```
END AddComplex;
```

```
(* input *)
```

```
(* output *) (* Procedure to subtract two complex numbers *)
```

```
PROCEDURE SubtComplex(C1, C2: Complex;
    VAR Result: Complex);
```

```
BEGIN
```

```
Result.Rel := C1.Rel-C2.Rel;
```

```
Result.Imag := C1.Imag-C2.Imag
```

```
END SubtComplex;
```

```
(* input *)
```

```
(* output *) (* Procedure to multiply two complex numbers *)
```

```

PROCEDURE MultComplex(C1, C2: Complex;
                     VAR Result: Complex);
BEGIN
  Result.Rel := C1.Rel*C2.Rel-C1.Imag*C2.Imag;

  Result.Imag := C1.Rel*C2.Imag+C2.Rel*C1.Imag
END MultComplex;

(* input *)
(* output *) (* Function to divide two complex numbers and return TRUE *)
(* if operation is successful, FALSE for division by zero *)

PROCEDURE DivComplex(C1, C2: Complex;
                    VAR Result: Complex): BOOLEAN;

VAR
  OK: BOOLEAN;
  SumSqr: REAL;

  VAR DivComplexResult:BOOLEAN;
BEGIN
  IF (C2.Rel <> Float(0)) OR (C2.Imag <> Float(0)) THEN

    OK := TRUE;
    SumSqr := (C2.Rel*C2.Rel)+(C2.Imag*C2.Imag);
    Result.Rel := (C1.Rel*C2.Rel+C1.Imag*C2.Imag)/SumSqr;

    Result.Imag := (C2.Rel*C1.Imag-C1.Rel*C2.Imag)/SumSqr

  ELSE
    OK := FALSE
  END;
  DivComplexResult := OK;
  RETURN DivComplexResult
END DivComplex;

(* output *) (* Procedure to read complex number *)

PROCEDURE ReadComplex(VAR C: Complex);

```

```

BEGIN
  WriteString(stdout, 'Enter real part ', 0);
  ReadBuffer(on);
  ReadReal(stdinout, C.Rel);
  Readln(stdinout);
  ReadBuffer(off);
  WriteString(stdout, 'Enter imaginary part ', 0);
  ReadBuffer(on);
  ReadReal(stdinout, C.Imag);
  Readln(stdinout);
  ReadBuffer(off);
  Writeln(stdinout);
END ReadComplex;

(* input *) (* Procedure to output a complex number *)

PROCEDURE WriteComplex(C: Complex);
BEGIN
  WriteString(stdout, 'Complex number = ', 0);
  WriteReal(stdinout, C.Rel, 0, -1);
  WriteString(stdout, ' + i ', 0);
  WriteReal(stdinout, C.Imag, 0, -1);
  Writeln(stdinout)
END WriteComplex;

BEGIN (*----- MAIN -----*)
  REPEAT

    ClrScr;
    WriteString(stdout, 'Enter operation [Q = quit] ', 0);
    ReadBuffer(on);
    ReadChar(stdinout, Operation);
    Readln(stdinout);
    ReadBuffer(off);
    Writeln(stdinout);
    Operation := CAP(Operation);
    IF
      Operation <> 'Q' THEN

```

```
WriteString(stdinout, 'Enter first complex number', 0);
Writeln(stdinout);
ReadComplex(C1);
WriteString(stdinout, 'Enter second complex number', 0);
Writeln(stdinout);
ReadComplex(C2);
Writeln(stdinout);
Correct := TRUE;
CASE Operation OF
  '+' :
    AddComplex(C1, C2, C3)
  | '-' :
    SubtComplex(C1, C2, C3)
  | '*' :
    MultComplex(C1, C2, C3)
  | '/' :

    Correct := DivComplex(C1, C2, C3);
  IF
    NOT Correct THEN
      WriteString(stdinout, 'Divide by zero error ', 0);
      Writeln(stdinout)
    END;

  ELSE
    AddComplex(C1, C2, C3)
  END;
  IF Correct THEN
    WriteComplex(C3)
  END;
  Writeln(stdinout);
  Writeln(stdinout);
  WriteString(stdinout, 'Press <CR> to continue ', 0);
  ReadBuffer(on);
  ReadChar(stdinout, Dummy);
  Readln(stdinout);
```



```
    ReadBuffer(off);
    Writeln(stdout);
    Writeln(stdout);
END;
UNTIL Operation = 'Q';
END ComplexCalc.
```

### 3.1.2 How to Create a Modula-2 Library Module

We are now ready to edit the above Modula-2 program to create a library module. In Modula-2, a module consists of two separate parts: a definition part and an implementation part. The definition module defines the items exported. These include constants, data types, variables, and procedures (for procedures, you only need to state the heading). The implementation module contains local constants, data types, variables and procedures. In addition, the implementation module contains the body of the exported procedures.

You compile the definition module before the implementation module. This enables the compiler to detect any discrepancies between the definition and implementation modules. In a sense, the definition module functions as an agreement or contract. It lists the promises made to 'client' programs. The implementation module must 'deliver' accordingly.

Note that constants, types and variables listed in the definition module must not necessarily appear in the implementation module (with the exception of opaque data types).

Because Modula-2 stresses the use of separately compiled modules, application programs often call on library modules which in turn call on other library modules, and so on. This generates a chain of module calls. Modula-2 supports a practical aspect of modular software development which minimizes the need for recompilation. When an implementation module is modified, it is recompiled and then the application program should be relinked. You do **not** need to recompile all modules between the altered library and the application program. If the definition and the implementation of a module are changed, then you may need to edit, recompile, and relink other modules and the application programs.

Now you are ready to create the definition and implementation modules for the complex operations library. You should make duplicate source files for the translated program. Since both modules need the same name, call them 'ComplexLib0'. The library filenames are 'COMPLEXL.DEF' for the definition module and 'COMPLEXL.MOD' for the implementation module. These files are already included in your diskette.

Invoke a text editor and start editing the definition module source file, following the steps indicated below:

- 1 Rename the module heading to 'DEFINITION MODULE ComplexLib0'.
- 2 Look at the import lists. In our example, none are needed in any definition module declaration. Hence, delete the import lists. In general, the definition module must import any items it needs to define its data types and procedures.
- 3 Insert an 'EXPORT QUALIFIED' statement, listing all exported items.
- 4 The example has no constants. In case there were any to be made accessible to client programs, they would be listed in the definition module.
- 5 The data type declaration contains the single exported record type. Leave it intact so client programs are aware of the detailed structure of the complex type. This is known as a transparent data type export.
- 6 The variable declaration section is removed, since it belongs to the application program. In general, list any variables the module is exporting.
- 7 The procedure and function bodies, and the original main section are deleted.
- 8 The module is terminated with 'END ComplexLib0.'

The edited definition module is shown below:

```
in file COMPLEXL.DEF:

DEFINITION MODULE ComplexLib0;

(* ANY IMPORT LISTS ARE PLACED HERE *)

EXPORT QUALIFIED Complex, AddComplex, SubtComplex,
                MultComplex, DivComplex,
                ReadComplex, WriteComplex;

(* ANY EXPORTED CONSTANTS ARE PLACED HERE *)
```

```
TYPE
  Complex = RECORD
    ReI, (* Real part *)
    (* Imaginary part *) Imag: REAL;
  END;

(* ANY EXPORTED VARIABLES ARE PLACED HERE *)

(* EXPORTED PROCEDURE/FUNCTION HEADINGS ARE LISTED BELOW *)

PROCEDURE AddComplex(C1, C2: Complex; (* input *)
  VAR Result: Complex); (* output *)
(* Procedure to add two complex numbers *)

PROCEDURE SubtComplex(C1, C2: Complex; (* input *)
  VAR Result: Complex (* output *));
(* Procedure to subtract two complex numbers *)

PROCEDURE MultComplex(C1, C2: Complex; (* input *)
  VAR Result: Complex (* output *));
(* Procedure to multiply two complex numbers *)

PROCEDURE DivComplex(C1, C2: Complex; (* input *)
  VAR Result: Complex (* output *)): BOOLEAN;
(* Function to divide two complex numbers and return TRUE *)
(* if operation is successful, FALSE for division by zero *)

PROCEDURE ReadComplex(VAR C: Complex (* output *));
(* Procedure to read complex number *)

PROCEDURE WriteComplex(C: Complex (* input *));
(* Procedure to output a complex number *)

END ComplexLib0.
```

Now we study the implementation source file 'COMPLEXL.MOD'. Initially, it contains a copy of the translated program. The following is the sequence of editing steps to follow to obtain the implementation module:

- 1 Rename the module heading to 'IMPLEMENTATION MODULE ComplexLib0'.
- 2 Examine the import lists. In our example, all are needed, except the 'ScreenHandler' module, used by the client application program. The 'ScreenHandler' import list is deleted.
- 3 Next we remove the declaration of the 'Complex' record because it has been already declared in the DEFINITION MODULE.
- 4 The variable declaration section is also removed, since it belongs to the application program, not to the generic library.
- 5 The procedure and function bodies are maintained and the original main section is deleted.
- 6 In our example, the module needs no initialization body. In case you are writing one that does, the implementation module will have a main body section for that purpose.
- 7 The module is terminated with 'END ComplexLib0'.

The edited implementation module is shown below:

in file COMPLEXL.MOD:

```
IMPLEMENTATION MODULE ComplexLib0;
```

```
FROM FloatingUtilities IMPORT Frac, Int, Round, Float, Trunc;
```

```
FROM TRealIO IMPORT ReadReal, WriteReal;
```

```
FROM TTextIO
```

```
  IMPORT ReadInt, ReadCard, ReadChar, ReadString, ReadLn, ReadBuffer,  
  WriteInt, WriteCard, WriteChar, WriteString, WriteBool, WriteLn,  
  Eoln, SeekEof, SeekEoln;
```

```

FROM TKernelIO
  IMPORT File, FileType, OptionMode, StatusProc, ReadProc, WriteProc,
  stdout, input, output, con, trm, kbd, lst, aux, usr,
  conStPtr, conInPtr, auxInPtr, usrInPtr, conOutPtr, lstOutPtr,
  auxOutPtr, usrOutPtr, errorPtr, IOresult, KeyPressed, IOBuffer,
  IOCheck, DeviceCheck, CtrlC, InputFileBuffer, OutputFileBuffer;

PROCEDURE AddComplex(C1, C2: Complex;      (* input *)
                   VAR Result: Complex); (* output *)
(* Procedure to add two complex numbers *)
BEGIN
  Result.Rel := C1.Rel+C2.Rel;
  Result.Imag := C1.Imag+C2.Imag
END AddComplex;

PROCEDURE SubtComplex(C1, C2: Complex;    (* input *)
                    VAR Result: Complex); (* output *)
(* Procedure to subtract two complex numbers *)
BEGIN
  Result.Rel := C1.Rel-C2.Rel;
  Result.Imag := C1.Imag-C2.Imag
END SubtComplex;

PROCEDURE MultComplex(C1, C2: Complex;    (* input *)
                    VAR Result: Complex); (* output *)
(* Procedure to multiply two complex numbers *)
BEGIN
  Result.Rel := C1.Rel*C2.Rel-C1.Imag*C2.Imag;
  Result.Imag := C1.Rel*C2.Imag+C2.Rel*C1.Imag
END MultComplex;

PROCEDURE DivComplex(C1, C2: Complex;      (* input *)
                   VAR Result: Complex): BOOLEAN; (* output *)
(* Function to divide two complex numbers and return TRUE *)
(* if operation is successful, FALSE for division by zero *)

VAR
  OK: BOOLEAN;
  SumSqr: REAL;

```

```
VAR DivComplexResult:BOOLEAN;
BEGIN
  IF (C2.Rel <> Float(0)) OR (C2.Imag <> Float(0)) THEN

    OK := TRUE;
    SumSqr := (C2.Rel*C2.Rel)+(C2.Imag*C2.Imag);
    Result.Rel := (C1.Rel*C2.Rel+C1.Imag*C2.Imag)/SumSqr;

    Result.Imag := (C2.Rel*C1.Imag-C1.Rel*C2.Imag)/SumSqr

  ELSE
    OK := FALSE
  END;
  DivComplexResult := OK;
  RETURN DivComplexResult
END DivComplex;

PROCEDURE ReadComplex(VAR C: Complex);      (* output *)
(* Procedure to read complex number *)
BEGIN
  WriteString(stdinput, 'Enter real part ', 0);
  ReadBuffer(on);
  ReadReal(stdinput, C.Rel);
  Readln(stdinput);
  ReadBuffer(off);
  WriteString(stdinput, 'Enter imaginary part ', 0);
  ReadBuffer(on);
  ReadReal(stdinput, C.Imag);
  Readln(stdinput);
  ReadBuffer(off);
  Writeln(stdinput);
END ReadComplex;

PROCEDURE WriteComplex(C: Complex);        (* input *)
(* Procedure to output a complex number *)
```

```
BEGIN
  WriteString(stdout, 'Complex number = ', 0);
  WriteReal(stdout, C.ReI, 0, -1);
  WriteString(stdout, ' + i ', 0);
  WriteReal(stdout, C.Imag, 0, -1);
  Writeln(stdout)
END WriteComplex;

END ComplexLib0.
```

Let us go back to the original translated Modula-2 program and make some changes on the source file to convert it into a program that calls on our newly created 'ComplexLib0' library.

The following procedure accomplishes this:

- 1 Remove the unnecessary 'FloatingUtilities', 'TRealIO' import lists.
- 2 Insert an import list for the 'ComplexLib0'.
- 3 Remove the original declaration of the 'Complex' type.
- 4 Remove all procedure and function definition since they are all imported from the 'ComplexLib0'.

The new version of the application program is shown below.

```
in file COMPLEXC.MO1:

MODULE ComplexCalc;

  FROM ScreenHandler
  IMPORT ClrEol, ClrScr, DelLine, InsLine, GotoXY, WhereX, WhereY,
  CrtInit, CrtExit, LowVideo, NormVideo, HighVideo, SetAttribute,
  GetAttribute, normalAtt, boldAtt, reverseAtt, underlineAtt,
  blinkAtt, boldUnderlineAtt, blinkUnderlineAtt, boldBlinkAtt,
  reverseBlinkAtt, boldUnderlineBlinkAtt;
```

```

FROM TTextIO
  IMPORT ReadInt, ReadCard, ReadChar, ReadString, ReadLn, ReadBuffer,
  WriteInt, WriteCard, WriteChar, WriteString, WriteBool, Writeln,
  Eoln, SeekEof, SeekEoln;
FROM TKernelIO
  IMPORT File, FileType, OptionMode, StatusProc, ReadProc, WriteProc,
  stdout, input, output, con, trm, kbd, lst, aux, usr,
  conStPtr, conInPtr, auxInPtr, usrInPtr, conOutPtr, lstOutPtr,
  auxOutPtr, usrOutPtr, errorPtr, IOresult, KeyPressed, IOBuffer,
  IOCheck, DeviceCheck, CtrlC, InputFileBuffer, OutputFileBuffer;
FROM ComplexLib0
  IMPORT Complex, AddComplex, SubtComplex, MultComplex,
  DivComplex, ReadComplex, WriteComplex;

```

```
(* Program to simulate a four-function complex calculator *)
```

```

VAR
  C1, C2, C3: Complex;
  Correct: BOOLEAN;
  Operation, Dummy: CHAR;

```

```
BEGIN (*----- MAIN -----*)
```

```
  REPEAT
```

```

    ClrScr;
    WriteString(stdout, 'Enter operation [Q = quit] ', 0);
    ReadBuffer(on);
    ReadChar(stdout, Operation);
    ReadLn(stdout);
    ReadBuffer(off);
    Writeln(stdout);
    Operation := CAP(Operation);
    IF
      Operation <> 'Q' THEN

```



```
WriteString(stdout, 'Enter first complex number', 0);
Writeln(stdout);
ReadComplex(C1);
WriteString(stdout, 'Enter second complex number', 0);
Writeln(stdout);
ReadComplex(C2);
Writeln(stdout);
Correct := TRUE;
CASE Operation OF
  '+':
    AddComplex(C1, C2, C3)
  | '-':
    SubtComplex(C1, C2, C3)
  | '*':
    MultComplex(C1, C2, C3)
  | '/':

    Correct := DivComplex(C1, C2, C3);
    IF
      NOT Correct THEN
        WriteString(stdout, 'Divide by zero error ', 0);
        Writeln(stdout)
    END;

ELSE
    AddComplex(C1, C2, C3)
END;
IF Correct THEN
  WriteComplex(C3)
END;
Writeln(stdout);
Writeln(stdout);
WriteString(stdout, 'Press <CR> to continue ', 0);
```

```

    ReadBuffer(on);
    ReadChar(stdinout, Dummy);
    Readln(stdinout);
    ReadBuffer(off);
    Writeln(stdinout);
    Writeln(stdinout);
END;
UNTIL Operation = 'Q';
END ComplexCalc.

```

**A second approach.** Suppose that the developer of the Turbo Pascal complex calculator has already created a generic Pascal file that implements data and procedures to be included by application programs. Following a typical situation, the data declarations are often stored in a separate file. We assume that file 'COMPLEXL.TYP' contains the definition for the 'Complex' type and that the library of routines are in file 'COMPLEXL.LIB'.

in file COMPLEXL.TYP:

```

TYPE Complex = RECORD
    Rel, (* Real part *)
    Imag (* Imaginary part *) : REAL;
END;

```

in file COMPLEXL.LIB:

```

PROCEDURE Add_Complex(C1, C2 : Complex;    (* input *)
                    VAR Result : Complex (* output *));
(* Procedure to add two complex numbers *)

BEGIN
    Result.Rel := C1.Rel + C2.Rel;
    Result.Imag := C1.Imag + C2.Imag
END;

PROCEDURE Subt_Complex(C1, C2 : Complex;    (* input *)
                    VAR Result : Complex (* output *));
(* Procedure to subtract two complex numbers *)

```

```

BEGIN
    Result.Rel := C1.Rel - C2.Rel;
    Result.Imag := C1.Imag - C2.Imag
END;

PROCEDURE Mult_Complex(C1, C2 : Complex;    (* input *)
                      VAR Result : Complex (* output *));
(* Procedure to multiply two complex numbers *)

BEGIN
    Result.Rel := C1.Rel * C2.Rel - C1.Imag * C2.Imag;
    Result.Imag := C1.Rel * C2.Imag + C2.Rel * C1.Imag
END;

FUNCTION Div_Complex(C1, C2 : Complex;    (* input *)
                    VAR Result : Complex (* output *)) : BOOLEAN;
(* Function to divide two complex numbers and return TRUE *)
(* if operation is successful, FALSE for division by zero *)

VAR OK : BOOLEAN;
    SumSqr : REAL;

BEGIN
    IF (C2.Rel <> 0) OR (C2.Imag <> 0)
    THEN BEGIN
        OK := TRUE;
        SumSqr := SQR(C2.Rel) + SQR(C2.Imag);
        Result.Rel := (C1.Rel * C2.Rel + C1.Imag * C2.Imag)
                      / SumSqr;
        Result.Imag := (C2.Rel * C1.Imag - C1.Rel * C2.Imag)
                      / SumSqr
    END
    ELSE
        OK := FALSE;
    Div_Complex := OK;
END;

PROCEDURE Read_Complex(VAR C : Complex (* output *));
(* Procedure to read complex number *)

```

```

BEGIN
  WRITE('Enter real part '); READLN(C.Rel);
  WRITE('Enter imaginary part '); READLN(C.Imag);
  WRITELN;
END;

PROCEDURE Write_Complex(C : Complex (* input *));
(* Procedure to output a complex number *)
BEGIN
  WRITELN('Complex number = ',C.Rel,' + i ',C.Imag)
END;

```

The main application program, using these generic files, would look like the following listing:

```

in file COMPLEXC.PA1:

PROGRAM Complex_Calc;
(* Program to simulate a four-function complex calculator *)
(* This version uses included files. *)

(* Get "Complex" type definition *)
(*$I COMPLEXL.TYP *)

VAR C1, C2, C3 : Complex;
    Correct : BOOLEAN;
    Operation, Dummy : CHAR;

(* Get procedures and function for complex operations *)
(*$I COMPLEXL.LIB *)

BEGIN (*----- MAIN -----*)
  REPEAT
    ClrScr;
    WRITE('Enter operation [Q = quit] ');
    READLN(Operation); WRITELN;
    Operation := UpCase(Operation);
    IF Operation <> 'Q'

```

```

THEN BEGIN
  WRITELN('Enter first complex number');
  Read_Complex(C1);
  WRITELN('Enter second complex number');
  Read_Complex(C2); WRITELN;
  Correct := TRUE;
  CASE Operation OF
    '+' : Add_Complex(C1, C2, C3);
    '-' : Subt_Complex(C1, C2, C3);
    '*' : Mult_Complex(C1, C2, C3);
    '/' : BEGIN
      Correct := Div_Complex(C1, C2, C3);
      IF NOT Correct
      THEN WRITELN('Divide by zero error ');
      END
    ELSE Add_Complex(C1, C2, C3);
  END;
  IF Correct THEN Write_Complex(C3);
  WRITELN; WRITELN;
  WRITE('Press <CR> to continue ');
  READLN(Dummy); WRITELN; WRITELN;
END;
UNTIL Operation = 'Q';
END.

```

To convert Turbo Pascal with included files, simply process the application program through the Translator and obtain a complete Modula-2 program. You edit exactly as in the first case to yield the sought Modula-2 library modules. This approach is the same for all application programs using included Pascal files.

If one had already translated Turbo Pascal libraries (i.e. COMPLEXL.TYP, COMPLEXLLIB) into Modula-2 versions, how would one deal with converting other Turbo Pascal application programs that 'include' the same Pascal libraries? The answer is simple: either rename the include files or move these files to a disk or directory not accessed by the Translator. When the Translator is unable to find an included file, it continues the conversion process, placing warning messages at what it perceives to be 'undefined' Pascal routines and data objects. You then edit the converted Modula-2 program to include import statements for the previously translated libraries.

### 3.2 Data Abstraction Using Opaque Types

In the above example, you exported the type 'Complex' and made its internal structure known to the client programs.

This means that other programmers can write additional routines for your program, to manipulate complex numbers, such as complex math function routines.

Suppose you wish to modify the structure of your program or to represent complex numbers using polar coordinates, as opposed to the rectangular ones. While the new polar coordinates also use two reals (angle and modulus), the mathematical operations involved in the calculations are quite different. This change is not recommended for transparent exported types, since it creates a version conflict with routines written by others.

#### 3.2.1 How to Hide Internal Representations of Data Types

Modula-2 allows the programmer to maintain control over exported data types by hiding their internal structures. The client programs are limited to the operations exported.

Applying the opaque type concept to the 'ComplexLib0' library module, perform the following changes:

- In the definition module replace the transparent exported 'Complex' with an opaque type declaration:

```
TYPE Complex; (* Opaque export *)
```

- The complete definition of type 'Complex' is relocated into the implementation module and is rewritten as a pointer structure to reflect the use of polar coordinate representation of complex numbers:

```
(* Definition uses polar coordinates *)
TYPE Complex = POINTER TO RECORD
    Angle, Modulus : REAL;
END;
```

The pointer type is mandatory for structured opaque types.

- The change in the 'Complex' record structure is echoed wherever complex operations and I/O are performed. In our example, this includes all the routines in the implementation module.

The library modules are shown below. You should keep in mind that while we have internally switched from rectangular to polar representation, the client application program is unaware of the change. As far as it is concerned, the complex calculator is still operating within rectangular coordinates.

```

in file COMPLEXL.DE2:

DEFINITION MODULE ComplexLib0;
(* New version to export opaque type *)

(* ANY IMPORT LISTS ARE PLACED HERE *)

EXPORT QUALIFIED Complex, AddComplex, SubtComplex,
                MultComplex, DivComplex,
                ReadComplex, WriteComplex;

(* ANY EXPORTED CONSTANTS ARE PLACED HERE *)

TYPE Complex; (* Opaque export *)

(* ANY EXPORTED VARIABLES ARE PLACED HERE *)

(* EXPORTED PROCEDURE/FUNCTION HEADINGS ARE LISTED BELOW *)

PROCEDURE AddComplex(C1, C2: Complex; (* input *)
                    VAR Result: Complex); (* output *)
(* Procedure to add two complex numbers *)

PROCEDURE SubtComplex(C1, C2: Complex; (* input *)
                    VAR Result: Complex (* output *));
(* Procedure to subtract two complex numbers *)

PROCEDURE MultComplex(C1, C2: Complex; (* input *)
                    VAR Result: Complex (* output *));
(* Procedure to multiply two complex numbers *)

```

```

PROCEDURE DivComplex(C1, C2: Complex; (* input *)
                    VAR Result: Complex (* output *)): BOOLEAN;
(* Function to divide two complex numbers and return TRUE *)
(* if operation is successful, FALSE for division by zero *)

PROCEDURE ReadComplex(VAR C: Complex (* output *));
(* Procedure to read complex number *)

PROCEDURE WriteComplex(C: Complex (* input *));
(* Procedure to output a complex number *)

END ComplexLib0.

```

The implementation which uses polar coordinates is available in file 'COMPLEXL.MO3'. You can switch back to rectangular coordinates which are more practical to use. Notice that there are two new local procedures, 'MakeComplex' and 'BreakComplex'. The first is needed to create a complex number from the real and imaginary parts while the second returns the latter parts from a complex number, regardless of its internal representation.

In file 'COMPLEXL.MO4' you find the version of the library using rectangular coordinates and exporting the opaque complex type. Because of the different internal representation using rectangular coordinates, there is no need to define and use local procedures like 'MakeComplex' and 'BreakComplex'. You do everything using normal dereferencing to improve performance.

Notice that in both versions we have added at the end of the implementation module the following code:

```

VAR ch: CHAR;

BEGIN
  Writeln(stdout);
  WriteString(stdout, 'xxx Coordinate Version using Opaque Type', 0);
  Writeln(stdout);
  WriteString(stdout, 'Press <CR> to continue', 0);
  Writeln(stdout);
  ReadChar(stdout, ch);
END ComplexLib0.

```



This part of the module is called the `ModuleBody` and is a part of code belonging to the module and executed `BEFORE` your main program is started. The `Module Body` is often referred to as the `Module Initialization Code` because it can be used to explicitly initialize data structures, load error handling routines, and perform all the procedures needed to insure the correct behavior of the module before your application will use it.

### 3.2.2 Space Allocation for Opaque Types

The implementation of an opaque type as given in the previous pages (files `COMPLEXL.DE2`, `COMPLEXL.MO3`, `COMPLEXL.MO4`) is not complete. As you can see from the implementation modules the allocation of the memory space used to store a complex type is done in the library module by the routine `'MakeComplex'` using the statement `NEW`. This routine, local to the module, and therefore not available to other client modules, takes care to allocate the memory space but there is no equivalent routine to deallocate that memory space when the complex number is no longer needed. Unfortunately, the `'life'` of the complex number depends on the application using it. To make the sample clear we have decided that a complex number would be allocated any time an operation is requested on it. This will lead to an increased use of the heap space without any way to release that space. This approach, although reasonable for a sample, cannot be used in building a reusable module library using `OPAQUE` types. A more correct approach is described in the following files.

The definition module (file `COMPLEXL.DE5`) has been modified to include two new functions `'CreateComplex'` and `'DestroyComplex'` that allow the application to control the `'life'` of variables of data type `Complex` without knowing the implementation. Both implementation modules (file `COMPLEXL.MO6` and file `COMPLEXL.MO7`) have been modified to include the implementation of these routines and to remove the `NEW` call from the `'MakeComplex'` routine.

Finally, the application module (file `COMPLEXC.MO8`) has been modified to generate the appropriate calls to create or destroy complex numbers allowing the application to control their existence.

### 3.3 Summary

All the sample files used in this chapter are available on diskette. The following is a summary of all the filenames used.

#### A) Normal translation

```
COMPLEXC.PAS -> COMPLEXC.MOD
```

#### B) Building a simple library module

```
COMPLEXC.PAS -> COMPLEXL.DEF  library definition
               -> COMPLEXL.MOD  library implementation
               -> COMPLEXC.M01  application
```

#### C) Using opaque types with two different implementations

```
COMPLEXC.PAS -> COMPLEXL.DE2  library definition with opaque type
               -> COMPLEXL.M03  library implementation using polar coordinates
               -> COMPLEXL.M04  library implementation using rectangular coord.
               -> COMPLEXC.M01  application, same as b)
```

#### D) Using opaque types and create/destroy procedure

```
COMPLEXC.PAS -> COMPLEXL.DE5  library definition with opaque type and
                           create/destroy
               -> COMPLEXL.M06  library implementation using polar coordinates
                           and create/destroy
               -> COMPLEXL.M07  library implementation using rectangular
                           coordinates and create/destroy
               -> COMPLEXC.M08  application with create/destroy
```

The correct sequence in compiling and linking these modules is:

- 1 Compile the definition module (COMPLEXL.DEF)
- 2 Compile the implementation module (COMPLEXL.MOD)
- 3 Compile the application/program module (COMPLEXC.MO1)
- 4 Link the application/program module (COMPLEXC.LNK)
- 5 Run the application/program module (COMPLEXC.LOD)

From DOS you can issue the following commands:

C> m2c complexl.def complexl complexc.mo1 (\* to compile \*)  
 C> m2l complexc (\* to link \*)  
 C> m2 complexc (\* to run \*)

If you have the LOGITECH MODULA-2/86 Base Language System 512LK you can type:

C> m2c complexl.def complexl complexc.mo1 (\* to compile \*)  
 C> m2l complexc (\* to link \*)  
 C> m2 complexc (\* to run \*)

From the MOD editor, to visit a file, type F3. To compile, type F5. To link, type F6. To run, type ALT F9 r complexc<CR>.

## 4 CALLING TURBO PASCAL'S EXTERNAL PROCEDURES FROM MODULA-2

LOGITECH MODULA-2/86 uses its own object file format (.LNK, .LOD files). However, this does not ease the use of code written in assembler (or another language). The LOGITECH MODULA-2/86 User's Guide describes one method of interfacing assembly code libraries. This method is based on linking the assembly code to the Modula-2 Run-Time System.

In this chapter we describe a more dynamic way to call routines written in assembler. This method can be used to call from Modula-2 external procedures developed for Turbo Pascal. The assembler routines are loaded at execution time and can then be called from the application. We call such a set of installable routines a 'driver' and they can be used for the implementation of the hardware dependent part of a program. An application can then decide at run-time, depending on the existing hardware, which version of the driver to load. However, this method is not restricted to the use of hardware specific code, but can be used for any code that may be written more efficiently in assembler.

We will also describe how the features of external procedures in Turbo Pascal can be implemented with this method of interfacing assembly code languages. Before we begin, we review how external procedures are used in Turbo Pascal. For example, we implement three simple screen handling routines in assembler, which we want to call from the Pascal program.

All files used in this chapter are available on diskette.

### 4.1 The Turbo Pascal Approach

The following is a sample Pascal program which uses these procedures:

```
in file TESTOUT.PAS:

PROGRAM TestOut;

(*$V-*)
TYPE
  ANYSTRING = STRING[255];

PROCEDURE OutChar(ch: CHAR); EXTERNAL 'PAS-OUT.BIN';
  (* write a single character on the screen *)
```

```
PROCEDURE OutLn; EXTERNAL OutChar[3];
  (* go to beginning of next line on the screen *)

PROCEDURE OutString(s: ANYSTRING); EXTERNAL OutChar[6];
  (* write a string on the screen *)

VAR
  i: INTEGER;

BEGIN
  OutString('This is a test of the assembler screen output interface');
  OutLn;
  FOR i := 1 TO 80 DO OutChar('-');
  OutLn;
END.
```

The following shows how to implement these procedures in assembler, following the procedure calling conventions of Turbo Pascal. To know the offsets of the procedure entrypoints, we put a jump table to our actual procedure code at the beginning of the code:

in file PAS-OUT.ASM:

```
CODE    SEGMENT

CGROUP  GROUP  CODE
ASSUME  CS: CGROUP

        JMP    OutChar    ; is at offset 0
        JMP    OutLn      ; is at offset 3
        JMP    OutString  ; is at offset 6
```

For the standard procedure 'Prolog' and 'Epilog', we define two macros:

ENTER and LEAVE:

```

ENTER MACRO
    PUSH BP
    MOV BP, SP
ENDM

LEAVE MACRO parameterSize
    MOV SP, BP
    POP BP
    RET parameterSize ; remove parameters from stack
ENDM

```

Now, we can implement the three procedures. The record structures 'OutCharParam' and 'OutStringParam' describe the stack at address SS:BP. For Turbo Pascal, the procedures must be declared as **near procedures**.

```

OutCharParam STRUC
    DW ? ; old BP
    DW ? ; return address
    char DB ?
OutCharParam ENDS

OutChar PROC NEAR
    ENTER 0
    MOV DL, [BP].char
    MOV AH, 2
    INT 21H ; DOS function 2, standard output
    LEAVE 2
OutChar ENDP

```

```
OutLn PROC NEAR
```

```
    ENTER
    MOV DL, 0DH ; CR
    MOV AH, 2
    INT 21H ; write CR with DOS function 2
    MOV DL, 0AH ; LF
    MOV AH, 2
    INT 21H ; write LF with DOS function 2
    LEAVE
```

```
OutLn ENDP
```

```
OutStringParam STRUC
```

```
    DW ? ; old BP
    DW ? ; return address
```

```
    str DB 256 DUP (?)
```

```
OutStringParam ENDS
```

```
OutString PROC NEAR
```

```
    ENTER 0
    PUSH DS
    PUSH SS
    POP DS ; get segment of string address
    LEA SI, [BP].str ; get offset of string address
    LODSB ; load length of string
    CMP AL, 0
    JZ stop ; empty string
    XOR AH, AH
    MOV CX, AX
```

```
nextch:
```

```
    LODSB ; load next character
```

```
    MOV DL, AL
```

```
    MOV AH, 2
```

```
    INT 21H ; write character
```

```
    LOOP nextch ; if not at end of string or null character continue
```

```
stop:
```

```
        POP  DS
        LEAVE 256
OutString ENDP

CODE ENDS

END
```

The following steps lead to an executable version of our sample program:

- 1 Edit the assembler program in file PAS-OUT.ASM.
- 2 Assemble the file PAS-OUT.ASM. The result is file PAS-OUT.OBJ.
- 3 Link the file PAS-OUT.OBJ. The result is file PAS-OUT.EXE.
- 4 Produce a file PAS-OUT.BIN with the conversion utility EXE2BIN.
- 5 With the Turbo editor edit the program TESTOUT.PAS.
- 6 Compile the program TESTOUT.PAS into the file TESTOUT.COM.

This compilation step automatically binds the code of our assembler code in PAS-OUT.BIN in the final executable file TESTOUT.COM.

- 7 Run the program TESTOUT.COM.

## 4.2 The Modula-2 Approach

The Turbo Pascal feature of external procedures doesn't exist in Modula-2. However, with the help of the module 'Drivers', it is possible, to achieve a similar result. The following shows how the program 'TestOut' looks in LOGITECH MODULA-2/86:

The declaration of the Procedure Headings for External Procedures are replaced by the declaration of a RECORD variable, with fields of procedure types.



Since the concept of external procedures does not exist in Modula-2, the compiler or linker cannot directly include the assembler code in the final executable file. Therefore, we load the assembler code at run-time. For this reason, the sample program will import the module 'Drivers'. The following sample module shows how the assembler code is loaded and executed. The procedure calls to the external procedures in Pascal are replaced by calls to our procedure variables in the record 'external'.

in file TESTOUT.MOD:

```
MODULE TestOut;
```

```
  IMPORT Drivers;
```

```
  VAR
```

```
    external: RECORD
```

```
      OutChar : PROCEDURE (CHAR);
```

```
        (* write a single character on the screen *)
```

```
      OutLn   : PROCEDURE ();
```

```
        (* go to beginning of next line on the screen *)
```

```
      OutString: PROCEDURE (ARRAY OF CHAR);
```

```
        (* write a string on the screen *)
```

```
    END;
```

```
    out: Drivers.Driver;
```

```
    done: BOOLEAN;
```

```
    i: INTEGER;
```

```
BEGIN
```

```
  Drivers.InstallDriver(out, external, 'MOD-OUT.BIN', TRUE, done);
```

```
  IF done THEN (* assembler code loaded *)
```

```
    WITH external DO
```

```
      OutString('This is a test of the assembler screen output interface');
```

```
      OutLn;
```

```
      FOR i := 1 TO 80 DO OutChar(' ') END;
```

```
      OutLn;
```

```
    END; (* WITH *)
```

```
    Drivers.UninstallDriver(out);
```

```
  END.
```

```
END TestOut.
```

The implementation of the assembler code for Modula-2 is similar to the implementation done for Turbo Pascal. The procedures must be declared as far procedures, and the jump table at the beginning of the module is replaced by a table of entry point offsets of the procedures. The order of the entrypoints must correspond to the order of the procedure variables in the record 'external':

```
in file MOD-OUT.ASM:
```

```
CODE SEGMENT 'CODE'
CGROUP GROUP CODE
ASSUME CS: CGROUP
```

```
ENTER MACRO
```

```
    PUSH BP
    MOV BP, SP
ENDM
```

```
LEAVE MACRO parameterSize
```

```
    MOV SP, BP
    POP BP
    RET parameterSize ; remove parameters from stack
ENDM
```

```
EntryPointOffsets DW OutChar, OutLn, OutString
```

```
OutCharParam STRUC
```

```
    DW ? ; old BP
    DD ? ; return address
    char DB ?
    dummy DB ?
```

```
OutCharParam ENDS
```

```
OutChar PROC FAR
```

```
    ENTER
    MOV DL, [BP].char
    MOV AH, 2
    INT 21H ; DOS function 2, standard output
    LEAVE 2
```

```
OutChar ENDP
```

```

OutLn PROC FAR
    ENTER
    MOV DL, 0DH ; CR
    MOV AH, 2
    INT 21H      ; write CR with DOS function 2
    MOV DL, 0AH ; LF
    MOV AH, 2
    INT 21H      ; write LF with DOS function 2
    LEAVE
OutLn ENDP

OutStringParam STRUC
    DW ? ; old BP
    DD ? ; return address
    strAddr DD ?
    strHigh DW ?
OutStringParam ENDS

OutString PROC FAR
    ENTER

    LDS SI, [BP].strAddr ; get address of string parameter
    MOV CX, [BP].strHigh ; get HIGH of string parameter
    INC CX                ; number of bytes of string parameter

    LODSB                ; load first character of string
    CMP AL, 0            ; is it null character?
    JZ stop              ; if yes, empty string; nothing to write

nextch:
    MOV DL, AL
    MOV AH, 2
    INT 21H              ; write character

```

```
        LODSB             ; load next character
        CMP  AL, 0        ; is it null character?
        LOOPNE nextch    ; if not at end of string or null character continue
stop:

        LEAVE 6
OutString ENDP

CODE ENDS

END
```

The following steps lead to an executable version of our sample program:

- 1 Edit the assembler program in file MOD-OUT.ASM.
- 2 Assemble the file MOD-OUT.ASM. This results in MOD-OUT.OBJ.
- 3 Link the file MOD-OUT.OBJ. The result is a file MOD-OUT.EXE.
- 4 Produce a file MOD-OUT.BIN with the conversion utility EXE2BIN.
- 5 Edit (with the MOD editor) the program TESTOUT.MOD.
- 6 Compile the program TESTOUT.MOD into the file TESTOUT.LNK.
- 7 Link the file TESTOUT.LNK into the file TESTOUT.LOD
- 8 Generate the file TESTOUT.EXE, using the LOD2EXE utility.
- 9 Run the program TESTOUT.EXE.  
At run-time, our program will load the assembler code of MOD-OUT.BIN in memory.

Now, let's look at the definition module 'Drivers'. Its implementation module is listed at the end of this appendix.

in file DRIVERS.DEF:

```
DEFINITION MODULE Drivers;
  FROM SYSTEM IMPORT
    WORD;

  EXPORT QUALIFIED
    Driver, InstallDriver, UninstallDriver;

  TYPE
    Driver; (* driver handle; hidden structure of a driver descriptor *)

  PROCEDURE InstallDriver(VAR d: Driver; VAR driverProc: ARRAY OF WORD;
                          fn: ARRAY OF CHAR; bin: BOOLEAN; VAR done: BOOLEAN);
    (* Loads the driver from the file with the name 'fn'.
       Initializes the driver handle 'd' and the user supplied record
       of procedure variables 'driverProc'.
       The boolean parameter 'bin' indicates whether the file to be loaded
       is a binary memory image or a relocatable file in the EXE file format.
       No search strategy is performed at this level.
    *)

  PROCEDURE UninstallDriver(d: Driver);
    (* Unloads the driver with the handle 'd'.
       Should only be called if InstallDriver was successful.
    *)

END Drivers.
```

In the example 'TestOut', we used the procedure 'InstallDriver' to load a binary memory image. The use of a binary memory image forces some restrictions on the way the assembler code can be written. The assembler program should only consist of one segment (size is less than 64KB), and no segment fixup information should be generated (no far references should be used within the assembler program). For small assembler code routines, these restrictions should not pose any problem. The advantage of a binary memory image is faster loading due to a smaller file and no need for fixups.

The procedure 'InstallDriver' is also able to load a file in the DOS EXE file format. In this case, the restrictions for the assembler code are less severe. The assembler program can use as many segments as possible. The table of entrypoints of the exported procedures must be at offset zero of the first segment. In this case, the entrypoints must be full 32 bit addresses with offset and segment values. The loader will not put a program segment prefix (PSP) in front of the loaded program. Therefore, the assembler code should not try to use the PSP. The current implementation restricts the overall size of the EXE program to 64KB. This is because the loader allocates the memory to load the program on the heap. The allocatable size is limited to 64KB.

### 4.3 Summary

It is possible to implement a method to call external procedures written in assembler from a Modula-2 program. In comparison with the way Turbo Pascal handles external procedures, the implementation for Modula-2 adds some overhead at load time to load the assembler code. Also the procedure calls are indirect procedure calls through the use of procedure variables, rather than direct calls to the procedures. However, with this dynamic loading, one has the possibility to use this feature to provide several different implementations of the external procedures and to decide at run-time (rather than link-time) which of them to use.

```
in file DRIVERS.MOD:
```

```
IMPLEMENTATION MODULE Drivers;  
  FROM SYSTEM IMPORT  
    ADR, WORD, ADDRESS, TSIZE, DOSCALL;  
  FROM Storage IMPORT  
    ALLOCATE, DEALLOCATE;
```

TYPE

```
FileName = ARRAY [0..79] OF CHAR;  
(* holds complete DOS filename, including drive, path, name  
and extension  
*)
```

```
DriverDescriptor = RECORD  
    loadMemory: ADDRESS;  
    loadSize: CARDINAL; (* in bytes *)  
END;
```

```
Driver = POINTER TO DriverDescriptor;  
(* implement hidden driver handle *)
```

```
(* header information about the load module in a .EXE file *)
```

```
EXEHdr = RECORD  
    signature,  
    imageLengthMod512, (* used to determine load size *)  
    pagesInFile, (* used to determine load size *)  
    (* 1 page = 512 bytes *)  
    relocationItemCount,  
    headerParagraphs, (* used to determine load size *)  
    minFreeParagraphs,  
    maxFreeParagraphs,  
    relativeSS,  
    initialSP,  
    checksum,  
    (* - (16-bit sum of words in file) *)  
    initialIP,  
    relativeCS,  
    relocationTableOffset,  
    overlayNumber: CARDINAL;  
END;
```

```
PROCEDURE InstallDriver(VAR d: Driver; VAR driverProc: ARRAY OF WORD;  
    fn: ARRAY OF CHAR; bin: BOOLEAN; VAR done: BOOLEAN);
```

```

VAR
  i: CARDINAL;
  a: ADDRESS;
  dummy: CARDINAL;
  binSize: CARDINAL;
  error: CARDINAL;
  read: CARDINAL;
  exeHdr: EXEHdr;
  file: CARDINAL; (* DOS file handle *)
  fileName: FileName;
  paramBlock: RECORD
    at: CARDINAL; (* load address in paragraphs *)
    rel: CARDINAL; (* relocation factor in paragraphs *)
  END;
BEGIN
  NEW(d); (* create driver descriptor *)
  WITH d^ DO
    (* copy filename into local variable.
       here we assume that HIGH(fileName) >= HIGH(fn),
       otherwise an index range error will occur
    *)
    FOR i := 0 TO HIGH(fn) DO
      fileName[i] := fn[i];
    END;
    fileName[HIGH(fn) + 1] := 0C;
    (* make sure there is a terminating null character *)

    done := FALSE; (* initial value *)

    DOSCALL(3DH, ADR(fileName), 0, file, error);
    (* open file *)
    IF error = 0 THEN
      IF bin THEN
        DOSCALL(42H, file, 2, 0, 0, dummy, binSize, error);
        (* get file size, by positioning at EOF *)
        IF error = 0 THEN
          loadSize := binSize;
          DOSCALL(42H, file, 0, 0, 0, dummy, dummy, error);
          (* reposition at beginning of file *)

```



```
END;
ELSE (* an EXE file *)
  DOSCALL(3FH, file, TSIZE(EXEHdr), ADR(exeHdr), read, error);
  (* read the header of the EXE file *)
  IF (error = 0) AND (read = TSIZE(EXEHdr)) THEN
    (* successful operation *)

    (* now we evaluate the size in bytes of code and data of the
    driver; using the information from the EXE-header
    *)
    loadSize := (exeHdr.pagesInFile * 32 - exeHdr.headerParagraphs) * 16;
    IF exeHdr.imageLengthMod512 > 0 THEN (* incomplete page *)
      loadSize := loadSize - 512 + exeHdr.imageLengthMod512
    END;
  END;
END;
INC(loadSize, 16); (* add a little for paragraph rounding *)

ALLOCATE(loadMemory, loadSize);
(* allocate memory to load driver *)

(* now we will adjust the load address, to point to a paragraph address
(i.e. offset is 0)
*)
a := loadMemory;
a.SEGMENT := a.SEGMENT + a.OFFSET DIV 16 + 1;
a.OFFSET := 0;
(* 'a' points now to next paragraph address after 'loadMemory' *)

IF bin THEN
  DOSCALL(3FH, file, binSize, a, read, error);
  (* read file in load memory *)
  IF (error = 0) AND (read = binSize) THEN
    done := TRUE;
```

```

    END;
    DOSCALL(3EH, file, error);
    (* close file *)
ELSE
    DOSCALL(3EH, file, error);
    (* close file *)

    (* prepare parameter block for DOS program load call *)
    paramBlock.at := a.SEGMENT;
    (* paragraph address of location to load driver *)
    paramBlock.rel := a.SEGMENT;
    (* relocation factor; is same as load address *)
    DOSCALL(4BH, ADR(fileName), ADR(paramBlock), 3, error);
    (* load overlay DOS function call *)
    IF error = 0 THEN
        done := TRUE; (* we did it *)
    END;
END; (* IF *)
IF NOT done THEN
    DEALLOCATE(loadMemory, loadSize);
    (* deallocate memory, load was not successful *)
END;
(*
ELSE
    (* file not found *)
*)
END; (* IF *)
END; (* WITH *)
IF done THEN
    (* copy entry points in user entry point table *)
    FOR i := 0 TO HIGH(driverProc) BY 2 DO
        driverProc[i] := a^; (* load offset value *)
        INC(a, 2);
        IF bin THEN
            driverProc[i+1] := WORD(a.SEGMENT); (* set segment value *)
        ELSE
            driverProc[i+1] := a^; (* load segment value *)
            INC(a, 2);
        END;
    END;

```

```
    END;
ELSE
    DISPOSE(d); (* destroy driver descriptor, not needed anymore *)
END;
END InstallDriver;

PROCEDURE UninstallDriver(d: Driver);
BEGIN
    WITH d^ DO
        DEALLOCATE(loadMemory, loadSize);
        (* remove memory, where driver was loaded *)
    END; (* WITH *)
    DISPOSE(d);
    (* destroy driver descriptor; no check is done, whether it was
       previously allocated
       *)
END UninstallDriver;

END Drivers.
```

## APPENDIX A

## MAPPING OF TURBO PASCAL PROCEDURES TO MODULA -2

The LOGITECH Translator translates all Turbo Pascal functions and procedures into Modula-2 procedures. The mapping used is detailed in this appendix. For example, the Turbo Pascal function 'Addr (x)' will be translated into 'ADR (x)', while 'Halt (i)' will be translated into 'SetErrorCode (i)' followed by 'Terminate (normal)', while 'Halt' alone will be translated into 'Terminate (normal)' only.

Special cases:

- And, Or, Not, Xor, Shl and Shr are described in Chapter Two.
- Mem, MemW, Port and PortW are described in Chapter Two.
- HeapPtr and ErrorPtr are described in Chapter Two.
- Graphic functions are described at the end of this appendix.

## Turbo Pascal standard Procedures

TURBO PASCAL	MODULA-2
Abs ( i : Integer ) : Integer;	ABS ( x );
Abs ( r : Real ) : Real;	result type = argument type;
	MODULE SYSTEM
Addr( anyType ) : Pointer;	ADR ( anyType ) : ADDRESS;
	MODULE TFileIO
Append ( VAR f : File );	Append ( VAR f : File );

ArcTan ( r : Real ) : Real;	MODULE MathLib0     arctan ( r : REAL ) : REAL;
-----	
Assign ( VAR f : File; name : String);	MODULE TFileIO     AssignFile ( VAR f : File;                    name : ARRAY OF CHAR;                    type : FileType);     NOTE: type is the type corresponding to f in        Turbo Pascal.
-----	
BlockRead (VAR f : File; VAR dest : Type; num : Integer (* optional *) [ VAR res : Integer ]);	MODULE TBinaryIO     BlockRead (VAR f : File;            VAR dest : ARRAY OF BYTE;            num : CARDINAL);   BlockRWResult ( VAR res : INTEGER);     NOTE : FileType Of File must be untypedFile;        e.g.   BlockRead( f, dest, num );            will be translated into :                    BlockRead( f, dest, num );   and   BlockRead( f, dest, num, res );            into :                    BlockRead( f, dest, num );                    BlockRWResult( res );
-----	

<pre>BlockWrite      (VAR f      : File;                  VAR dest  : Type;                  num       : Integer (* optional *) [ VAR res  : Integer ]);</pre>	<pre>MODULE TBinaryIO BlockWrite (VAR f      : File;            VAR dest   : ARRAY OF BYTE;            num       : CARDINAL);  NOTE : FileType Of File must be untypedFile; es: BlockWrite( f, dest, num );            will be translated into :            BlockWrite( f, dest, num ); and BlockWrite( f, dest, num, res );            into :            BlockWrite( f, dest, num );            BlockRWRResult( res );</pre>
<pre>Chain ( var f : File );</pre>	<pre>NOT supported, see section on Overlay</pre>
<pre>ChDir( path : String );</pre>	<pre>MODULE TDiskDirectory ChDir( path : ARRAY OF CHAR );</pre>
<pre>Chr ( i : Integer ) : Char;</pre>	<pre>CHR( i : INTEGER ) : CHAR</pre>
<pre>Close ( VAR f      : File );</pre>	<pre>MODULE TFileIO Close ( VAR f : File );</pre>
<pre>ClrEol;</pre>	<pre>MODULE ScreenHandler ClrEol;</pre>
<pre>ClrScr;</pre>	<pre>MODULE ScreenHandler ClrScr;</pre>



Delete ( VAR s : String; pos : Integer; len : Integer );	MODULE Strings  Delete ( VAR s : ARRAY OF CHAR; pos : CARDINAL; len : CARDINAL );
Delline;	MODULE ScreenHandler  Delline;
Dispose( var p : Pointer );	DISPOSE ( p );  NOTE: To use this procedure one should import procedure DEALLOCATE from MODULE Storage.
Dseg : Integer;	MODULE MemoryOperations  Dseg(): INTEGER;
Eof : Boolean;	MODULE TFileIO
Eof ( VAR f : File ) : Boolean;	Eof ( VAR f : File ) : BOOLEAN;
Eoln : boolean;	MODULE TTextIO
Eoln ( VAR f : Text ) : Boolean;	Eoln ( VAR f : File ) : BOOLEAN;  NOTE : FileType Of File must be Text.
Erase ( VAR f : File );	MODULE TFileIO  Erase ( VAR f : File );
Execute ( var f : File );	MODULE TExec  Execute( VAR f : File );



Exit;	In a PROCEDURE it is translated with   RETURN;     In a FUNCTION it is translated with   RETURN var;     where var is of the type to be returned   by the function.
-----	
Exp ( r : Real ) : Real;	MODULE MathLib0     exp ( r : REAL ) : REAL;
-----	
FilePos ( VAR f : File ) : Integer;	MODULE TBinaryIO     FilePos ( VAR f : File ) : INTEGER;     NOTE : FileType of f must be untyped File or   fileOfRecord.
-----	
FileSize ( VAR f : File ) : Integer;	MODULE TBinaryIO     FileSize ( VAR f : File ) : INTEGER;     NOTE : FileType of f must be untypedFile or   fileOfRecord
-----	
FillChar ( VAR dest : Integer; length : Integer; data : Byte );	MODULE MemoryOperations     FillChar( VAR dest : ARRAY OF BYTE;   length : Integer;   data : BYTE );     NOTE : Turbo.dest is equivalent to   Modula-2.dest.
-----	
Flush ( VAR f : File );	MODULE TFileIO     Flush ( VAR f : File );
-----	

Form (...);	NOT supported, see Introduction
-----	
	MODULE FloatingUtilities
Frac ( r : Real ) : Real;	Frac ( r : REAL ) : REAL;
-----	
	MODULE Storage
FreeMem(var p : Pointer; i : Integer )	DEALLOCATE( p, i );
-----	
	MODULE TDiskDirectory
GetDir( drive : Integer; VAR path : String );	GetDir( drive : INTEGER; VAR path : ARRAY OF CHAR );
-----	
	MODULE Storage
GetMem (var p : Pointer; i : Integer );	ALLOCATE ( p, i );
-----	
	MODULE ScreenHandler
GoToXY( x, y : Integer );	GoToXY( x, y : INTEGER );
-----	
Halt( i : Integer );	SetErrorCode( i );   Terminate( normal );
Halt	Terminate( normal );
	NOTE : - SetErrorCode is imported from MODULE   ErrorCodes;   - Terminate is from MODULE System   - normal is imported from MODULE System
-----	
	MODULE MemoryOperations
Hi ( i : Integer ) : Integer;	Hi ( i : WORD ) : INTEGER;
-----	

HighVideo;	MODULE ScreenHandler HighVideo;
-----	-----
Insert ( s : String; VAR d : String; pos : Integer );	MODULE Strings Insert ( s : ARRAY OF CHAR; VAR d : ARRAY OF CHAR; pos : CARDINAL );
-----	-----
InsLine;	MODULE ScreenHandler InsLine;
-----	-----
Int ( r : Real ) : Real;	Int ( r : REAL ) : REAL;
-----	-----
Intr ( interruptNo : Integer; var result : Record )	MODULE TDOS Intr( interruptNo : INTEGER; VAR regs : ARRAY OF WORD); NOTE : see definition module
-----	-----
IOresult : Interger	MODULE TKernelIO IOresult () : CARDINAL
-----	-----
KeyPressed : Boolean;	MODULE TKernelIO KeyPressed () : BOOLEAN;
-----	-----
Length ( s : String ) : Integer;	MODULE Strings Length ( VAR s : ARRAY OF CHAR ): CARDINAL;
-----	-----
Ln ( r : Real ) : Real;	MODULE MathLibO ln ( r : REAL ) : REAL;
-----	-----

<p>Lo ( i : Integer ) : Integer;</p>	<p>MODULE MemoryOperations     Lo ( i : WORD ) : INTEGER;</p>
<p>LongFilePos ( VAR f : File ) : Real;</p>	<p>MODULE TBinaryIO   LongFilePos ( VAR f : File ) : REAL;   NOTE : FileType of f must be untypedFile   or fileOfRecord.</p>
<p>LongFileSize ( VAR f : File ) : Real;</p>	<p>MODULE TBinaryIO   LongFileSize ( VAR f : File ) : REAL;   NOTE : FileType of f must be untypedFile   or fileOfRecord.</p>
<p>LongSeek ( VAR f : File Of Type; pos : Real);</p>	<p>MODULE TBinaryIO   LongSeek( VAR f : File;   pos : REAL);   NOTE : FileType of f must be untypedFile   or fileOfRecord.</p>
<p>LowVideo;</p>	<p>MODULE ScreenHandler   LowVideo;</p>
<p>Mark ( var p : Pointer );</p>	<p>MODULE Storage   InstallHeap;   NOTE: Mark and Realease can be translated   into Modula-2 InstallHeap and   RemoveHeap ONLY in the case that we   release the last heap we have marked !</p>

MaxAvail : INTEGER;	NOTE: is translated in MemAvail.
-----	MODULE MemoryOperations
MemAvail : INTEGER;	MemAvail ( ) : INTEGER;
-----	MODULE TDiskDirectory
MkDir( path : String );	MkDir( path : ARRAY OF CHAR );
-----	MODULE MemoryOperations
Move ( VAR source : Type; VAR dest : Type; length : Integer );	Move ( VAR source : ARRAY OF BYTE; VAR dest : ARRAY OF BYTE; length : INTEGER );
-----	MODULE TDOS
MsDos ( Func : Integer; param : Record );	MsDos ( VAR regs : ARRAY OF WORD );   NOTE : see definition module
-----	NEW ( p );
New ( var p : Pointer );	NOTE : To use this procedure one should import   procedure ALLOCATE from MODULE Storage.
-----	MODULE ScreenHandler
NormVideo;	NormVideo;
-----	MODULE Sounds
NoSound;	NoSound;
-----	ODD( x ) : BOOLEAN;
Odd( i : Integer ) : Boolean;	
-----	

Ofs ( p ) : Integer;	MODULE MemoryOperations     Ofs( p : ADDRESS ) : INTEGER;     NOTE : If p is a procedure the translation is :         Ofs( ADDRESS( p ) );           otherwise for a variable of any type :         Ofs( ADR( p ) );
-----	
Ord ( x : Scalar ) : Integer;	ORD( x : AnyEnumeration ) : CARDINAL;
-----	
OvrPath	NOT supported, see section on Overlay.
-----	
ParamCount : Integer;	MODULE TParameter     ParamCount ( ) : INTEGER;
-----	
ParamStr( n : Integer ) : String;	MODULE TParameter     ParamStr (    n : INTEGER;              VAR par : ARRAY OF CHAR );
-----	
Pos ( pattern String; source : String ) : Integer;	MODULE Strings     Pos ( pattern : ARRAY OF CHAR;         source : ARRAY OF CHAR ):CARDINAL;
-----	
Pred( x : Scalar ) : Scalar;	VAL( type-of-x, ORD( x ) - 1 );     NOTE : Turbo Pascal never returns range error:         to have the same result in Modula-2         one should disable range test.
-----	

Ptr ( seg : Integer; off : Integer ) : Pointer;	MODULE MemoryOperations Ptr ( seg, off : INTEGER ) : ADDRESS; e.g. pointer := Ptr( Cseg, \$80 ); becomes: pointer := Ptr( Cseg(), 80H );
<hr style="border-top: 1px dashed black;"/>	
Randomize;	MODULE Random Randomize; NOTE : see definition module.
<hr style="border-top: 1px dashed black;"/>	
Random ( range : Integer ) : Integer;	MODULE Random RandomInt ( range : INTEGER ):INTEGER;
<hr style="border-top: 1px dashed black;"/>	
Random : Real;	MODULE Random RandomReal() : Real;
<hr style="border-top: 1px dashed black;"/>	
Read ( VAR f: File Of Type; VAR v: Type         );	MODULE TBinaryIO Read( VAR f     : File; VAR v     : ARRAY OF BYTE); NOTE : Assign on f has been done with FileType = fileOfRecord; the actual parameter of v is a variable of any type.
<hr style="border-top: 1px dashed black;"/>	
Read ( VAR f : Text; VAR i : Integer);	MODULE TTextIO ReadInt ( VAR f : File; VAR i : INTEGER); NOTE : Assign on f has been done with FileType = Text;
<hr style="border-top: 1px dashed black;"/>	

<pre> Read ( VAR f : Text;       VAR r : Real); </pre>	<pre> MODULE TRealIO     ReadReal ( VAR f : File;             VAR r : REAL);     NOTE : Assign on f has been done with         FileType = Text; </pre>
<pre> Read ( VAR f : Text;       VAR c : Char); </pre>	<pre> MODULE TTextIO     ReadChar ( VAR f : File;             VAR c : CHAR);     NOTE : Assign on f has been done with         FileType = Text; </pre>
<pre> Read ( VAR f : Text;       VAR s : String); </pre>	<pre> MODULE TTextIO     ReadString ( VAR f : File;              VAR s : ARRAY OF CHAR);     NOTE : Assign on f has been done with         FileType = Text; </pre>
<pre> Readln ( VAR f : Text ); </pre>	<pre> MODULE TTextIO     ReadLn ( VAR f : File );     NOTE : Assign on f has been done with         FileType = Text; </pre>
<pre> Read ( var1, var2, .. , varN ); </pre>	<pre> ReadBuffer( on ); Read&lt;typeVar1&gt; ( stdout, var1 ); Read&lt;typeVar2&gt; ( stdout, var2 );       .... Read&lt;typeVarN&gt; ( stdout, varN ); ReadBuffer( off ); </pre>



Readln ( var1, var2, .. , varN );	ReadBuffer( on );   Read<typeVar1> ( stdout, var1 );   Read<typeVar2> ( stdout, var2 );                   ....   Read<typeVarN> ( stdout, varN );   ReadLn           ( stdout );   ReadBuffer( off );
Read ( filVar, var1, .. , varN );	Read<typeVar1> ( fileVar, var1 );                   ....   Read<typeVarN> ( fileVar, varN );
Readln ( filVar, var1, .. , varN );	Read<typeVar1> ( fileVar, var1 );                   ....   Read<typeVarN> ( file, varN );   ReadLn           ( file );
Release( var p : Pointer );	MODULE Storage   RemoveHeap;     NOTE: Mark and Realease can be translated         into Modula-2 InstallHeap and         RemoveHeap ONLY in the case that we         release the last heap we have marked !
Rename ( VAR f : File; name : String );	MODULE TFileIO     Rename ( VAR f : File; name : ARRAY OF CHAR);

```
Reset ( VAR f : File );
```

```
MODULE TFileIO
```

```
Reset ( VAR f : File ;  
        size : CARDINAL );
```

```
NOTE : size has a different meaning depending  
on the file type:
```

```
Text Files : is the size of the disk  
file buffer. In Turbo Pascal it is  
specified when the file variable is  
declared i.e. f : Text [ 200 ] ;
```

```
File of Records : is the record size.
```

```
Untyped Files : is the size of each  
record/block transferred by  
BlockRead/Write operations.
```

```
If size = 0 for TextFile and Untyped  
Files default value ( 128 bytes ) while  
for File of Record run time error 5  
will occur.
```

---

Rrewrite ( VAR f : File );	MODULE TFileIO  Rrewrite ( VAR f : File ; size : CARDINAL );  NOTE : size has a different meaning depending on the file type:  Text Files : is the size of the disk file buffer. In Turbo Pascal it is specified when the file variable is declared i.e. f : Text [ 200 ] ;  File of Records : is the record size.  Untyped Files : is the size of each record/block transferred by BlockRead/Write operations.  If size = 0 for TextFile and Untyped Files default value ( 128 bytes ) while for File of Record run time error 5 will occur.
Rmdir( path : String );	MODULE TDiskDirectory  Rmdir( path : ARRAY OF CHAR );
Round ( r : Real ) : Integer;	MODULE FloatingUtilities  Round( r : REAL ) : INTEGER;

Seek ( VAR f : File Of type; pos : Integer);	MODULE TBinaryIO   Seek ( VAR f : File;   pos : CARDINAL);     NOTE : FileType of f must be untypedFile or   fileOfRecord.
<hr/>	
SeekEof : Boolean;  SeekEof ( VAR f : File ) : Boolean;	MODULE TTextIO   SeekEof ( VAR f : File ) : BOOLEAN;     NOTE : FileType of File must be Text;
<hr/>	
SeekEoln : Boolean;  SeekEoln ( VAR f : Text ) : Boolean;	MODULE TTextIO   SeekEoln ( VAR f : File ) : BOOLEAN;     NOTE :FileType of File must be Text;
<hr/>	
Seg ( p ) : Integer;	MODULE MemoryOperations   Seg( p : ADDRESS ) : INTEGER;     NOTE: Same as in Ofs.
<hr/>	
Sin ( r : Real ) : Real;	MODULE MathLib0   sin ( r : REAL ) : REAL;
<hr/>	
SizeOf( var variable ) : Integer;	MODULE TParameter   SIZE( variable) : CARDINAL;
<hr/>	
SizeOf( <type identifier> ) : Integer	MODULE TParameter   TSIZE( anytype ) : CARDINAL;
<hr/>	

	MODULE Sounds
Sound ( hertz : Integer );	Sound ( hertz : INTEGER );
-----	
Sqr ( i : Integer ) : Integer;	( i * i )
-----	
Sqr ( r : Real ) : Real;	( r * r )
-----	
	MODULE MathLib0
Sqrt( r : Real ) : Real;	sqrt ( r : REAL ) : REAL;
-----	
	MODULE MemoryOperations
Sseg : Integer;	Sseg(): INTEGER;
-----	
	MODULE TNumberConversion
Str( i : Integer	IntToStr( i : INTEGER
(* option *)[:n];	n : INTEGER
VAR s : String );	VAR s : ARRAY OF CHAR );
-----	
	MODULE TRealNumberConversion
Str( r : real;	RealToStr( r : REAL;
(* option *)[:n];	n : INTEGER
(* option *)[:m];	m : INTEGER
VAR s : String );	VAR s : ARRAY OF CHAR );
-----	
Succ( x : Scalar ) : Scalar;	VAL( type-of-x, ORD( x ) + 1 );
	NOTE : Turbo Pascal never returns range error
	to have the same result in Modula-2
	one should disable range test.
-----	
	MODULE MemoryOperations
Swap ( i : Integer ) : Integer;	Swap ( i : WORD ) : INTEGER;
-----	

	MODULE FloatingUtilities
Trunc ( r : Real ) : Integer;	Trunc( r : REAL ) : INTEGER;
-----	
	MODULE TFileIO
Truncate( VAR f : File );	Truncate( VAR f : File );
-----	
UpCase ( ch : Char ) : Char;	CAP( ch : CHAR ) : CHAR;
-----	
	MODULE TRealNumberConversion
Val( s : String;	StrToReal( s : ARRAY OF CHAR;
VAR r : Real;	VAR r : REAL;
VAR p : Integer );	VAR p : INTEGER );
-----	
	MODULE TNumberConversion
Val( s : String;	StrToInt( s : ARRAY OF CHAR;
VAR i : Integer;	VAR i : INTEGER;
VAR p : Integer );	VAR p : INTEGER );
-----	
	MODULE ScreenHandler
WhereX : Integer;	WhereX ( ) : INTEGER;
-----	
	MODULE ScreenHandler
WhereY : Integer;	WhereY ( ) : INTEGER;
-----	

<pre>Write ( VAR f: File Of Type;         VAR v: Type       );</pre>	<pre>MODULE TBinaryIO  Write( VAR f   : File;        VAR v   : ARRAY OF BYTE);  NOTE : Assign on f has been done with        FileType = fileOfRecord;        the actual parameter of v is a        variable of any type.</pre>
<pre>Write ( VAR f : Text;         i : Integer         (* optional *)[:n]);</pre>	<pre>MODULE TTextIO  WriteInt ( VAR f : File;            i : INTEGER;            n : CARDINAL); (* default 0 *)  NOTE : Assign on f has been done with        FileType = Text;</pre>
<pre>Write ( VAR f : Text;         r : Real         (* optional *)[:n]         (* optional *)[:m]);</pre>	<pre>MODULE TRealIO  WriteReal ( VAR f : File;             r : REAL;             n : CARDINAL; (* default 18 *)             m : INTEGER); (* default -10 *)  NOTE : Assign on f has been done with        FileType = Text;</pre>
<pre>Write ( VAR f : Text;         b : Boolean         (* optional *)[:n]);</pre>	<pre>MODULE TTextIO  WriteBool ( VAR f : File;             b : BOOLEAN;             n : CARDINAL); (* default 0 *)  NOTE : Assign on f has been done with        FileType = Text;</pre>

<pre>Write ( VAR f : Text;         c : Char         (* optional *)[:n]);</pre>	<pre>MODULE TTextIO WriteChar ( VAR f : File;             c : CHAR;             n : CARDINAL); (* default 0 *)  NOTE : Assign on f has been done with        FileType = Text;</pre>
<pre>Write ( VAR f : Text;         s : String         (* optional *)[:n]);</pre>	<pre>MODULE TTextIO WriteString ( VAR f : File;               s : ARRAY OF CHAR;               n : CARDINAL);(* default 0 *)  NOTE : Assign on f has been done with        FileType = Text;</pre>
	<pre>(*) n is the parameter corresponding to the width. In Turbo Pascal can be specified after a colon. e.g. Write ( file, i : 4 ); will be translated into WriteInt ( file, i, 4 );  default for n is 0  e.g. Write ( file, i); will be translated into WriteInt ( file, i, 0 );  for reals default for n is 18 default for m is -10</pre>



	MODULE TTextIO
Writeln ( VAR f : Text );	WriteLn ( VAR f : File );
	NOTE : Assign on f has been done with         FileType = Test;
-----	
Write ( var1, var2, .. , varN );	Write<typeVar1> ( stdout, var1 );
	Write<typeVar2> ( stdout, var2 );
	....
	Write<typeVarN> ( stdout, varN );
-----	
Write ( filVar, var1, .. , varN );	Write<typeVar1> ( fileVar, var1 );
	Write<typeVar2> ( filVar, var2 );
	....
	Write<typeVarN> ( filVar, varN );
-----	
Writeln ( var1, var2, .. , varN );	Write<typeVar1> ( stdout, var1 );
	Write<typeVar2> ( stdout, var2 );
	....
	Write<typeVarN> ( stdout, varN );
	WriteLn ( stdout );
-----	
Writeln ( filVar, var1, .. , varN );	Write<typeVar1> ( fileVar, var1 );
	Write<typeVar2> ( filVar, var2 );
	....
	Write<typeVarN> ( filVar, varN );
	WriteLn ( filVar );
-----	

## Graphic Functions

The Modula-2 graphic libraries are constituted by the following modules:

- GraphBasic, which corresponds to the Turbo 'Basic graphic routines'.
- GraphExtended, which corresponds to the Turbo 'Extended graphic routines'.
- GraphTurtle, which corresponds to the Turbo 'Turtle graphics routines'.

The Modula-2 interface is the same as the Turbo interface with the following exceptions:

TextMode;		MODULE GraphBasic
TextMode( mode : Integer );		TextModeReset;
		TextMode( mode : INTEGER );
		MODULE GraphBasic
GetPic ( VAR buffer : anyType;		GetPic( VAR buffer : ARRAY OF BYTE;
x1, y1,		x1, y1,
x2, y2 : Integer);		x2, y2 : INTEGER );
		MODULE GraphBasic
PutPic ( VAR buffer : anyType;		PutPic( VAR buffer : ARRAY OF BYTE;
x, y : Integer);		x, y : INTEGER );

### NOTE:

The constants, such as BW40, BW80, C40, C80 and the color constants, are exported from Module 'GraphBasic'.

The constants North, East, South and West are exported from Module 'GraphTurtle'.

## APPENDIX B

## MAPPING OF TURBO PASCAL COMPILER DIRECTIVES TO MODULA-2

The Turbo Pascal compiler options are translated into Modula-2 in both LOGITECH MODULA-2/86 compiler options and procedure calls.

When a compiler option is translated into a procedure call the effect at run time for the program is the same as if the compiler would have produced a different code. All these procedures are defined in module 'TKernelIO'.

The Translator takes care to place the procedure calls in the right place. The Translator translates options that are global to the whole program into procedure calls placed at the beginning of the module body, while it translates options that are local into a procedure placed locally.

Global directives are: B, C, D, G, and P.

Local directives are: I, K, and R.

We suggest that you also study the Modula-2 Compiler Directives in the MODULA-2/86 User's Guide to take advantage of all the possibilities the LOGITECH MODULA-2/86 Compiler offers you. Some of these include:

- stack, range and index test
- code generation for 80286
- code generation for the Math Coprocessor 8087/80287
- alignments
- statistics
- interactive/batch operations
- listing control

## Turbo Pascal to Modula-2 Compiler Directive Mapping

Turbo Pascal	Modula-2
{\$A+/-}	NOT SUPPORTED -- CP/M-80 Compiler Directive
{\$B+}	IOBuffer( on );(* default *)
{\$B-}	IOBuffer( off );
	NOTE : IOBuffer is slightly different from the B option because Modula-2 does not have a read without file parameter and so the effect of {\$B-} is to buffer all the files that refer to the console.
{\$C+}	CtrlC( on ); (* default *)
{\$C-}	CtrlC( off );
{\$D+}	DeviceCheck( on ); (* default *)
{\$D-}	DeviceCheck( off );
{\$Fxx}	NOT SUPPORTED -- see LOGITECH MODULA-2/86 User's Guide Installation Chapter
{\$Gsize}	InputFileBuffer ( size ); (* default 128 *)

{I+}	IOCheck ( on ); (* default *)
{I-}	IOCheck ( off );
{I fname}	Include Files are fully supported, see section on Translator options
-----	
{K+}	(*S+*) (* LOGITECH MODULA-2/86 Comp. Dir. stack test on *)
{K-}	(*S-*) (* LOGITECH MODULA-2/86 Comp. Dir. stack test off *) (* default on *)
-----	
{Psize}	OutputFileBuffer ( size ); (* default 128 *)
-----	
{R+}	(*T+,R+*) (* LOGITECH MODULA-2/86 range & index test on *)
{R-}	(*T-,R-*) (* LOGITECH MODULA-2/86 range & index test off *) (* default on *)
-----	
{V+/-}	NOT SUPPORTED -- see Modula-2 language feature Open Array
-----	
{Wx}	NOT SUPPORTED -- CP/M-80 Compiler Directive
-----	
{U+/-}	NOT SUPPORTED -- see LOGITECH MODULA-2/86 User's Guide Aborting a program
-----	
{X+/-}	NOT SUPPORTED -- CP/M-80 Compiler Directive
-----	

## APPENDIX C

## COMPATIBILITY BETWEEN TURBO PASCAL AND MODULA-2 DATA FILES

This appendix addresses the problems of data representation in files created using Turbo Pascal and Modula-2. When translating a Pascal program that uses files into a Modula-2 program, you should be aware of the format used to store data in files. Most of these problems are due to the different internal data representation used by Turbo Pascal and Modula-2. The incompatibilities you may encounter for the different types of files are:

- **TEXT FILES**

Text files are compatible, in other words from the Modula-2 program you can read all the text files produced by Turbo Pascal. The only problem occurs when reading a real number, produced with Modula-2, from a Turbo Pascal program. The syntax of Turbo does not accept the Modula-2 three-digit exponent for a real.

- **UNTYPED FILES**

The blocks of bytes transferred with the block read/write operation remain unchanged. The possible incompatibilities depend on the contents of these blocks. Check to see if you use in the block any data types that are not compatible.

- **FILE OF RECORD**

Files of record are, in general, not compatible. In fact, Turbo Pascal and Modula-2 have a different internal representation for some data types.

To be able to use a (incompatible) data file generated using a Turbo Pascal program from the Modula-2 translated version, you should first write a specialized utility program to convert the original data file (Turbo Pascal data format), into a new data file (Modula-2 data format). After this conversion, the Modula-2 program will run correctly. A possible alternative is to regenerate the data files with the translated Modula-2 program.

## REPRESENTATION OF TYPES

TYPE	Turbo Pascal	Modula-2
BOOLEAN	1-BYTE : 0 = FALSE, 1 = TRUE	Same representation as T-Pascal
CHAR	1-BYTE : ASCII character set	Same representation as T-Pascal
INTEGER	2-BYTE : -32768 .. 32767, two's complement notation, least significant byte first	Same representation as T-Pascal
ENUMERATION and SUBRANGE	<p>1-BYTE If: integer subranges with both bounds in the range 0..255, declared scalars with less than 255 possible values. This byte contains the original value of the number.</p> <p>2-BYTES If: integer subranges with one or both bounds not within the range 0..255, and declared scalars with more than 256 possible values. These bytes contain a 2's complement 16-bit value with the least significant byte stored first.</p>	<p>Enumeration type: 1 BYTE, elements are numbered 0..255.</p> <p>Subrange type: same representation as the base type.</p>

REAL	<p>6-BYTES :</p> <p>A floating point value with a 40-bit mantissa and an 8-bit 2's exponent. The exponent is stored in the first byte and the mantissa in the next 5-bytes with the least significant byte first. The exponent uses binary format with an offset of 80H. If the exponent is zero, the floating point value is considered to be zero.</p> <p>The value of the mantissa is obtained by dividing the 40-bit unsigned integer by <math>2^{40}</math>. The mantissa is always normalized, i.e. the most significant bit (bit 7 of the fifth byte) should be interpreted as a 1. The sign of the mantissa is stored in this bit, a 1 indicating a negative number.</p>	<p>8-BYTES :</p> <p>A floating point value with a 52-bit mantissa, 11-bit exponent and 1-bit sign. The exponent is stored in the last word bits 1..11 (bit 0 being the sign ), the mantissa in bit 12..15 of the last word and in the other 3 words, with the least significant byte first. The exponent uses binary format with an offset of 3FFH. If the exponent is 0 the floating point value is 0.</p> <p>The value of the mantissa is obtained by dividing the 52-bit unsigned integer by <math>2^{52}</math>. The mantissa is always normalized, i.e. the implicit binary point (before bit 12 of the last word) is interpreted as following a 1. The sign bit indicates negative numbers with a 1.</p>
-----		
STRING	<p>A string occupies as many bytes as the maximum length plus one. The first byte ( element 0 ) contains the current length of the string.</p>	<p>A string ( ARRAY [0..length-1] ) occupies 'length' bytes. The string with less than length elements is terminated by 0C . No current length is stored in the string</p>
-----		



ARRAY	An array is stored as a contiguous sequence of elements, with the index in ascending order, the right-most index varying most quickly.	Same representation as T-Pascal, but each array element may have a different representation in accordance with its type.
SETS	An element in a set occupies one bit, and the maximum number of elements in a set is 256. The number of bytes occupied by a set variable is calculated as $(\text{Max DIV } 8) - (\text{Min DIV } 8) + 1$ .	2-BYTES: The maximum number of elements in a set is 16. If we number the elements of a set from 0 to 15, the representation in a memory word is : 7 6 .. 1 0    15 14 .. 9 8 ( low byte )   ( high byte )
POINTER	4-BYTES: The first two bytes (lower address ) hold the offset value (lower byte first) and the second two bytes hold the segment value (lower byte first). NOTE: NIL = 0000:0000;	Same representation as T-Pascal  NOTE : NIL = FFFF:FFFF;
RECORD	The first field of a record is stored at the lowest memory address.	Same representation as T-Pascal, but each field may have a different representation in accordance with its type
FILE	76 BYTES for text files there are 128 bytes more to store the buffer.	see module TKernelIO

## APPENDIX D

## INDEX OF LIBRARY MODULES

For quick reference, we briefly describe in this section the set of library modules provided with the Translator. More detailed information is available in each definition module file (.DEF). Modules with names starting with 'T..' implement functions similar to the one already available in other LOGITECH MODULA-2/86 Base Language System Library Modules.

- **MODULE Delay**  
Interrupts the program execution for a given delay time.
- **MODULE Exec**  
Handles the execution of any DOS program from a Modula-2 program.
- **MODULE FloatingUtilities**  
Operations on fractional/integer part of a real number.
- **MODULE GraphBasic**
- **MODULE GraphExtended**
- **MODULE GraphTurtle**  
Graphics functions.
- **MODULE LongSet**  
Procedures for the management of sets larger than 16 elements.
- **MODULE MemoryOperations**  
Miscellaneous memory oriented operations like bitwise operations, fast move memory, procedures to get segments and offset, operations on predefined arrays Mem, MemW, Port, PortW.
- **MODULE Random**  
Generates random numbers.

- **MODULE Sounds**  
Generates sounds.
- **MODULE ScreenHandler**  
Screen management procedures.
- **MODULE TKernelIO**
- **MODULE TFileIO**
- **MODULE TTextIO**
- **MODULE TRealIO**
- **MODULE TBinaryIO**  
File Management procedures.
- **MODULE TNumberConversion**
- **MODULE TRealNumberConversion**  
Conversion between integers or reals and strings.
- **MODULE TParameter**  
Handles the command parameters string.
- **MODULE TDiskDirectory**  
Interface to directory functions.
- **MODULE TDOS**  
Interface to PC-DOS.
- **MODULE TExec**  
Execute any DOS program from a Modula-2 program (Turbo Pascal interface).

The following are other modules for which the Translator generates references: These are available with the LOGITECH MODULA-2/86 Base Language System.

- **MODULE ErrorCode**  
Handles return code to the operating system.
- **MODULE MathLib0**  
Real Math functions.

- **MODULE Storage**  
Standard dynamic storage management.
- **MODULE Strings**  
Variable length character string handler.
- **MODULE SYSTEM**  
Implementation and system dependent types, variables and procedures.
- **MODULE System**  
Additional system dependent facilities.

## LIST OF ALL IDENTIFIERS USED IN LIBRARY MODULES

Identifier	Module
ADDRESS.....	(SYSTEM)
ADR.....	(SYSTEM)
ALLOCATE .....	(Storage)
And .....	(MemoryOperations)
Append.....	(TFileIO)
Arc .....	(GraphExtended)
arctan.....	(MathLib0)
Assign.....	(Strings)
AssignFile.....	(TFileIO)
aux.....	(TKernelIO)
auxInPtr.....	(TKernelIO)
auxOutPtr.....	(TKernelIO)
Available .....	(Storage)
Back.....	(GraphTurtle)
Black.....	(GraphBasic)
Blink .....	(GraphBasic)
blinkAtt.....	(ScreenHandler)
blinkUnderlineAtt.....	(ScreenHandler)
BlockRWResult.....	(TBinaryIO)
BlockRead.....	(TBinaryIO)
BlockWrite.....	(TBinaryIO)
Blue.....	(GraphBasic)
boldAtt.....	(ScreenHandler)
boldBlinkAtt.....	(ScreenHandler)
boldUnderlineAtt.....	(ScreenHandler)
boldUnderlineBlinkAtt.....	(ScreenHandler)
Brown.....	(GraphBasic)
BuildSet.....	(LongSet)
BW40.....	(GraphBasic)
BW80.....	(GraphBasic)
BYTE.....	(SYSTEM)

C40.....	(GraphBasic)
C80.....	(GraphBasic)
CardToStr.....	(TNumberConversion)
ChDir .....	(TDiskDirectory)
Circle.....	(GraphExtended)
ClearScreen.....	(GraphTurtle)
Close.....	(TFileIO)
ClrEol.....	(ScreenHandler)
ClrScr .....	(ScreenHandler)
CODE.....	(SYSTEM)
ColorTable .....	(GraphExtended)
CompareStr.....	(Strings)
con.....	(TKernelIO)
Concat.....	(Strings)
conInPtr .....	(TKernelIO)
conOutPtr.....	(TKernelIO)
conStPtr.....	(TKernelIO)
Copy.....	(Strings)
cos.....	(MathLib0)
curProcess.....	(System)
CrtExit .....	(ScreenHandler)
CrtInit.....	(ScreenHandler)
Cseg .....	(MemoryOperations)
CtrlC.....	(TKernelIO)
Cyan.....	(GraphBasic)
DarkGray.....	(GraphBasic)
DEALLOCATE .....	(Storage)
DelLine.....	(ScreenHandler)
Delay.....	(Delay)
Delete.....	(Strings)
DeviceCheck.....	(TKernelIO)
DISABLE.....	(SYSTEM)
DOSCALL.....	(SYSTEM)
Draw .....	(GraphBasic)
Dseg.....	(MemoryOperations)

East .....	(GraphTurtle)
ENABLE .....	(SYSTEM)
entier .....	(MathLib0)
Eof.....	(TFileIO)
Eoln .....	(TTextIO)
EqualSet.....	(LongSet)
Erase.....	(TFileIO)
ErrorProc.....	(TKernelIO)
errorPtr.....	(TKernelIO)
Exclude .....	(LongSet)
Execute .....	(TExec)
exp .....	(MathLib0)
File .....	(TKernelIO)
FilePos .....	(TBinaryIO)
FileSize .....	(TBinaryIO)
FileType.....	(TKernelIO)
FillChar.....	(MemoryOperations)
FillPattern.....	(GraphExtended)
FillScreen.....	(GraphExtended)
FillShape .....	(GraphExtended)
Float .....	(FloatingUtilities)
Flush.....	(TFileIO)
Forwd .....	(GraphTurtle)
Frac.....	(FloatingUtilities)
GetAttribute.....	(ScreenHandler)
GetDir.....	(TDiskDirectory)
GetDotColor .....	(GraphExtended)
GetPic .....	(GraphExtended)
GETREG .....	(SYSTEM)
GotoXY .....	(ScreenHandler)
GraphBackground .....	(GraphBasic)
GraphColorMode .....	(GraphBasic)
GraphMode .....	(GraphBasic)
GraphWindow.....	(GraphBasic)
Green .....	(GraphBasic)

Heading.....	(GraphTurtle)
Hi.....	(MemoryOperations)
HiRes.....	(GraphBasic)
HiResColor.....	(GraphBasic)
HideTurtle.....	(GraphTurtle)
HighVideo.....	(ScreenHandler)
Home.....	(GraphTurtle)
IOBuffer.....	(TKernelIO)
INBYTE.....	(SYSTEM)
IOCheck.....	(TKernelIO)
IOresult.....	(TKernelIO)
InSet.....	(LongSet)
Include.....	(LongSet)
InitProcedure.....	(System)
input.....	(TKernelIO)
InputFileBuffer.....	(TKernelIO)
InsLine.....	(ScreenHandler)
Insert.....	(Strings)
InstallHeap.....	(Storage)
Int.....	(FloatingUtilities)
IntToStr.....	(TNumberConversion)
Intr.....	(TDOS)
INWORD.....	(SYSTEM)
kbd.....	(TKernelIO)
KeyPressed.....	(TKernelIO)
Length.....	(Strings)
LightBlue.....	(GraphBasic)
LightCyan.....	(GraphBasic)
LightGray.....	(GraphBasic)
LightGreen.....	(GraphBasic)
LightMagenta.....	(GraphBasic)
LightRed.....	(GraphBasic)
ln.....	(MathLib0)
Lo.....	(MemoryOperations)
LongFilePosition.....	(TBinaryIO)
LongFileSize.....	(TBinaryIO)



LongSeek .....	(TBinaryIO)
LowVideo .....	(ScreenHandler)
lst .....	(TKernelIO)
lstOutPtr .....	(TKernelIO)
Magenta .....	(GraphBasic)
MakeEmptySet .....	(LongSet)
MemAvail .....	(MemoryOperations)
MemGet .....	(MemoryOperations)
MemSet .....	(MemoryOperations)
MemWGet .....	(MemoryOperations)
MemWSet .....	(MemoryOperations)
MkDir .....	(TDiskDirectory)
Move .....	(MemoryOperations)
MsDos .....	(TDOS)
NEWPROCESS .....	(SYSTEM)
NoSound .....	(Sounds)
NoWrap .....	(GraphTurtle)
normalAtt .....	(ScreenHandler)
NormVideo .....	(ScreenHandler)
North .....	(GraphTurtle)
Not .....	(MemoryOperations)
Ofs .....	(MemoryOperations)
OptionMode .....	(TKernelIO)
Or .....	(MemoryOperations)
OUTBYTE .....	(SYSTEM)
output .....	(TKernelIO)
OutputFileBuffer .....	(TKernelIO)
OUTWORD .....	(SYSTEM)
Palette .....	(GraphBasic)
ParamCount .....	(TParameter)
ParamStr .....	(TParameter)
Pattern .....	(GraphExtended)
PenDown .....	(GraphTurtle)
PenUp .....	(GraphTurtle)
Plot .....	(GraphBasic)

PortGet.....	(MemoryOperations)
PortSet.....	(MemoryOperations)
PortWGet.....	(MemoryOperations)
PortWSet.....	(MemoryOperations)
Pos.....	(Strings)
PROCESS.....	(SYSTEM)
ProcessDescriptor.....	(System)
ProcessPtr.....	(System)
Ptr.....	(MemoryOperations)
PutPic.....	(GraphExtended)
RandomInt.....	(Random)
RandomReal.....	(Random)
Randomize.....	(Random)
Read.....	(TBinaryIO)
ReadCard.....	(TTextIO)
ReadChar.....	(TTextIO)
ReadInt.....	(TTextIO)
ReadLn.....	(TTextIO)
ReadProc.....	(TKernelIO)
ReadReal.....	(TRealIO)
ReadString.....	(TTextIO)
real.....	(MathLib0)
RealToStr.....	(TRealNumberConversion)
Red.....	(GraphBasic)
RegPack.....	(TDOS)
RemoveHeap.....	(Storage)
Rename.....	(TFileIO)
Reset.....	(TFileIO)
reverseAtt.....	(ScreenHandler)
reverseBlinkAtt.....	(ScreenHandler)
Rewrite.....	(TFileIO)
RmDir.....	(TDiskDirectory)
Round.....	(FloatingUtilities)
RTSVECTOR.....	(SYSTEM)
Seek.....	(TBinaryIO)
SeekEof.....	(TTextIO)
SeekEoln.....	(TTextIO)

Seg.....	(MemoryOperations)
SetAttribute.....	(ScreenHandler)
SetDifference.....	(LongSet)
SetErrorCode.....	(ErrorCode)
SetHeading.....	(GraphTurtle)
SetIncluded.....	(LongSet)
SetIntersection.....	(LongSet)
SetPenColor.....	(GraphTurtle)
SetPosition.....	(GraphTurtle)
SETREG.....	(SYSTEM)
SetUnion.....	(LongSet)
Shl.....	(MemoryOperations)
ShowTurtle.....	(GraphTurtle)
Shr.....	(MemoryOperations)
sin.....	(MathLib0)
SIZE.....	(SYSTEM)
Sound.....	(Sounds)
South.....	(GraphTurtle)
sqrt.....	(MathLib0)
Sseg.....	(MemoryOperations)
Status.....	(System)
StatusProc.....	(TKernelIO)
stdinout.....	(TKernelIO)
StrToCard.....	(TNumberConversion)
StrToInt.....	(TNumberConversion)
StrToReal.....	(TRealNumberConversion)
Swap.....	(MemoryOperations)
SWI.....	(SYSTEM)
SymmetricSetDifference.....	(LongSet)
TermProcedure.....	(System)
Terminate.....	(System)
TextBackground.....	(GraphBasic)
TextColor.....	(GraphBasic)
TextMode.....	(GraphBasic)
TextModeReset.....	(GraphBasic)
TRANSFER.....	(SYSTEM)
trm.....	(TKernelIO)
Trunc.....	(FloatingUtilities)

Truncate.....	(TFileIO)
TSIZE.....	(SYSTEM)
TurnLeft.....	(GraphTurtle)
TurnRight.....	(GraphTurtle)
TurtleDelay.....	(GraphTurtle)
TurtleThere.....	(GraphTurtle)
TurtleWindow.....	(GraphTurtle)
underlineAtt.....	(ScreenHandler)
usr.....	(TKernelIO)
usrInPtr.....	(TKernelIO)
usrOutPtr.....	(TKernelIO)
West.....	(GraphTurtle)
WhereX.....	(ScreenHandler)
WhereY.....	(ScreenHandler)
White.....	(GraphBasic)
Window.....	(GraphBasic)
WORD.....	(SYSTEM)
Wrap.....	(GraphTurtle)
Write.....	(TBinaryIO)
WriteBool.....	(TTextIO)
WriteCard.....	(TTextIO)
WriteChar.....	(TTextIO)
WriteInt.....	(TTextIO)
WriteLn.....	(TTextIO)
WriteProc.....	(TKernelIO)
WriteReal.....	(TRealIO)
WriteString.....	(TTextIO)
.....	
Xcor.....	(GraphTurtle)
Xor.....	(MemoryOperations)
Ycor.....	(GraphTurtle)
Yellow.....	(GraphBasic)

## INDEX

- 80286 176
- 8087 code 16
- 8087/80287 8, 176
  
- Abs 153
- Absolute 18, 86
- Absolute variables 21, 86
- Addr 153
- Alignments 176
- And 18, 153
- Append 153
- ArcTan 154
- ARRAY 56, 182
- Array of characters 58
- Assembler 137, 147
- Assign 154
- Aux, .i.lst 91
  
- BCD 3
- Binary memory image 147
- Bit Manipulation Operators 105
- BlockRead 154
- BlockWrite 155
- BOOLEAN 180
- BW40 175
- BW80 175
- Bytes 82
  
- C40 175
- C80 175
- CARDINAL 77
- CASE 86
- Chain 18, 107, 108, 155
- CHAR 180
- ChDir 155
- Chr 155
  
- Close 155
- ClrEol 155
- ClrScr 155
- Code 89
- Comment 113
- Comments 13
- CompareStr 63
- Compatible 179
- Compiler directives 13, 102
- Con 91
- Concat 18, 61, 62, 67, 72, 156
- Conditional statements 14
- Constant string 20, 21
- Constants 30
- Control character 37
- Control characters 33, 65
- Conversion 179
- Convert 179
- Coprocessor 8
- Copy 18, 67, 72, 156
- Copy, Concat 15
- Cos 156
- CrtExit 156
- CrtInit 156
- CSeg 86, 87, 156
- CtrlC 177
  
- Data file 179
- Data files 179
- Data representation 179
- Decimal 3
- Decimal point 14
- Definition module 118, 119
- Delay 156, 183
- Delete 66, 157
- DelLine 157
- Device I/O 102
- DeviceCheck 177
- Devices 16

Directories 11, 12  
 Dispose 157  
 Driver 146  
 DSeg 86, 87, 157  
  
 East 175  
 Empty 57  
 Emulation 16  
 ENUMERATION 180  
 Eof 157  
 Eoln 157  
 Erase 157  
 Error 18  
 Error handler 102  
 Error Handling 19  
 Error messages 16  
 ErrorCode 184  
 ErrorPtr 18, 93, 153  
 EXCL 43  
 Exec 183  
 Execute 15, 18, 107, 108, 157  
 Exit 106, 107, 158  
 Exp 158  
 EXPORT 119  
 EXTERNAL 137  
 External procedure 22  
 External procedures 137, 141  
 External Subprogram 19  
  
 File 16, 91, 101, 182  
 FILE OF RECORD 179  
 File of records 16  
 FileIO 109  
 FilePos 158  
 Files of records 102  
 FileSize 158  
 FillChar 158  
 Flag ?11 75  
 Flag ?1UNDEF 75  
  
 Flags 19  
 Float 80  
 FloatingUtilities 8, 183  
 Flush 158  
 Form 159  
 Formatted I/O 102  
 Forward 107  
 Forward declarations 15  
 Frac 159  
 FreeMem 159  
 Function 15  
 Function returning 18  
 Functions 69, 153  
  
 GetDir 159  
 GetMem 159  
 GETREG 90  
 Goto 18, 21, 30  
 GoToXY 159  
 GraphBasic 183  
 GraphExtended 183  
 Graphic 22  
 Graphic functions 153  
 GraphTurtle 183  
  
 Halt 107, 159  
 HeapPtr 101, 153  
 HeapTop 101  
 Hexadecimal constants 33  
 Hi 159  
 HighVideo 160  
  
 I/O 101  
 I/O drivers 18, 91  
 Identifiers 14  
 Implementation 118  
 Implementation module 118,  
 121, 131  
 IMPORT list 14

IN 43  
 In-line 22  
 INCL 43  
 Include 11, 13  
 Include file 11  
 Include Files 178  
 Included 127, 130  
 Included files 130  
 Incompatibilities 179  
 Indent 11  
 Index test 176, 178  
 Initialized variables 18, 30, 32  
 Inline 89  
 Inline code 18  
 Input 91  
 Insert 66, 160  
 Inset 52  
 InsLine 160  
 Installation 7  
 InstallDriver 146  
 InstallHeap 101  
 Int 160  
 INTEGER 77, 180  
 Intr 160  
 IntToStr 67  
 IOresult 160  
  
 Kbd 91  
 KeyPressed 160  
  
 Label 18, 21, 30  
 Length 59, 66, 160  
 Libraries 16  
 Library modules 17, 109, 183  
 Limits 20  
 Listing control 176  
 Ln 160  
 Lo 161  
 LongFilePos 161  
  
 LongFileSize 161  
 LongSeek 161  
 LongSet 42, 183  
 Loops. 14  
 Lower case 14  
 LowVideo 161  
  
 Mark 101, 161  
 Mark String 55  
 MathLib0 184  
 MaxAvail 162  
 Maxint 16, 30  
 Mem 18, 100, 153  
 MemAvail 162  
 MemGet 100  
 MemoryOperations 183  
 MemSet 100  
 MemW 18, 100, 153  
 MemWGet 100  
 MemWSet 100  
 Mkdir 162  
 Module 'Drivers' 142  
 MODULE Drivers 146  
 Module Initialization 134  
 Module name 13  
 ModuleBody 134  
 Move 162  
 MsDos 162  
  
 New 162  
 NormVideo 162  
 North 175  
 NoSound 162  
 Not 18, 105, 153  
  
 Object file 137  
 Odd 162  
 OF CHAR 56  
 Ofs 163

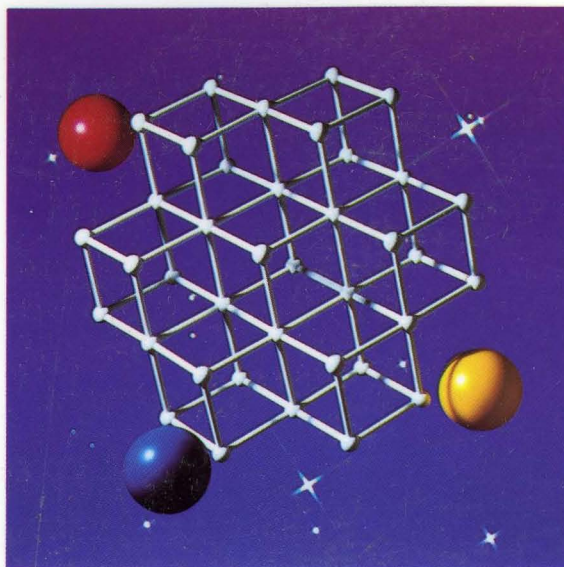
Opaque type 131  
 Opaque Types 131, 134, 135  
 Open Arrays 67  
 Options 10, 11, 176  
 Or 18, 105, 153  
 ORD 77, 163  
 Output 91  
 Overlay 15, 18, 107, 108  
 Overlays 19  
 OvrPath 18, 163  
  
 ParamCount 163  
 ParamStr 15, 18, 72, 163  
 Path 11  
 Paths 11  
 Pi 16, 30  
 POINTER 182  
 Port 18, 100, 153  
 PortGet 100  
 PortSet 100  
 PortW 18, 100, 153  
 PortWGet 100  
 PortWSet 100  
 Pos 66, 163  
 Post-mortem debugger 2  
 PRED 15, 163  
 Predefined files 16, 102  
 Predefined Variables 90  
 Procedures 153  
 Program 107  
 Program name 13  
 Ptr 164  
 PutPic 175  
  
 Random 8, 164, 183  
 Randomize 164  
 Range 176, 178  
 Read 164, 165, 166  
 ReadBuffer 103  
 ReadLn 165, 166  
 REAL 181  
 Real Arithmetic 3, 8  
 Real number 14  
 Reals 16, 80  
 RealToStr 67  
 RECORD 182  
 Release 101, 166  
 RemoveHeap 101  
 Rename 166  
 Reserved words 14  
 Reset 167  
 Result of Functions 106  
 RETURN 106  
 Rewrite 168  
 Rmdir 168  
 Round 80, 168  
  
 ScreenHandler 102, 184  
 Seek 169  
 SeekEof 169  
 SeekEoln 169  
 Seg 169  
 Set 14, 16  
 Set constructor 47  
 Set of characters 42, 47  
 SETREG 90  
 Sets 18, 42, 182  
 Shl 18, 105, 153  
 Shr 18, 105, 153  
 Sin 169  
 SizeOf 169  
 Sound 170  
 Sounds 184  
 South 175  
 Spelling 14  
 Sqr 170  
 Sqrt 170  
 Sseg 170



Stack 176  
Stack test 178  
Standard identifiers 14, 16, 17  
Stdinout 91, 103  
Storage 185  
Str 67, 170  
String 11, 21, 30, 55, 181  
String constant 37  
String Functions 66  
String Operator '+' 61  
Strings 11, 14, 15, 21, 55, 58, 185  
Strings operations 18  
StrToInt 67  
StrToReal 67  
Structured constant 41  
Structured data types 15  
Structured statements 14  
Subprogram 15, 22  
Subprograms 108  
Subrange 15, 180  
SUCC 15, 170  
Swap 170  
SYSTEM 185  
  
TBinaryIO 8, 184  
TDiskDirectory 184  
TDOS 184  
TExec 184  
Text files 102, 179  
TextMode 175  
TFileIO 184  
The Run-Time Debugger 2  
TKernelIO 184  
TNumberConversion 184  
TParameter 184  
Translation Rules 13  
Translator's Capacities 23  
TRealIO 8, 184  
TRealNumberConversion 8, 184

Trm 91  
Trunc 80, 171  
Truncate 171  
TTextIO 184  
Turbo-BCD 19  
Typed constants 30, 32  
  
Underscore 31  
Underscores 14  
UninstallDriver 146  
Untyped files 16, 102, 179  
Untyped Variable Parameters 88  
Untyped Variables 86  
UpCase 171  
Upper 14  
Uppercase 14  
Usr 91  
  
Val 67, 171  
  
Warning messages 33  
West 175  
WhereX 171  
WhereY 171  
WITH 83  
Write 172, 173, 174  
Writeln 174  
  
Xor 18, 105, 153

# LOGITECH™ MODULA-2 VERSION 3.0



## TURBO PASCAL TO LOGITECH™ MODULA-2 TRANSLATOR

 LOGITECH

Logitech U.S.A.  
Corporate Headquarters  
6505 Kaiser Drive  
Fremont, CA 94555  
Tel: 415-795-8500

Logitech Switzerland  
European Headquarters  
CH-1111 Romanel/Morges  
Switzerland  
Tel: 41-21-869-9656

Logitech Taiwan  
Far East Headquarters  
15 R&D Road 2  
Science Based Industrial Park  
Hsinchu, Taiwan, ROC  
Tel: 886-35-77-8241

Algol-Logitech Italy  
Via Durazzo 2  
20134 Milano MI  
Italy  
Tel: 39-2-215-5622