LOGITECH™ MODULA-2

Version 3.0

USER’S MANUAL
Trademarks

LOGITECH and POINT are trademarks, and LOGIMOUSE is a registered trademark of LOGITECH, Inc.

IBM is a registered trademark of International Business Machine Corporation.

CodeView is a trademark, and Microsoft, MS, and MS-DOS are registered trademarks of Microsoft Corporation.

Intel is a registered trademark of Intel Corporation.

Hewlett-Packard, HP, and LaserJet are registered trademarks of Hewlett-Packard Corporation.

Byte is a registered trademark of McGraw-Hill, Inc.

UNIX and AT&T are registered trademarks of American Telephone and Telegraph Corporation.

PFIIXPLUS is a trademark of Phoenix Software Associates, LTD.

Olivetti is a registered trademark of Olivetti.

COMPAQ is a registered trademark of Compaq Computer Corporation.
LOGITECH SOFTWARE LICENSE AGREEMENT

This document is a legal agreement between you, the licensee, and Logitech, Inc ("Logitech"). By using this program, you are agreeing to become bound by the terms of this agreement. If you do not agree to the terms of this agreement, promptly return the disk package and the other items that are part of this product in their original package, with your payment receipt (the "receipt"), to Logitech for a full refund.

In consideration of payment of the License Fee, which is a part of the price evidenced by the receipt, Logitech grants to the licensee a nonexclusive right, without right to sublicense, to use this copy of this Logitech Software on a single Computer at a time. Logitech reserves all rights not expressly granted, and retains title and ownership of the Software, including all subsequent copies in any media. This Software and the accompanying written materials are copyrighted. You may copy the Software solely for backup purposes; all other copying of the Software or the written materials is expressly forbidden.

As the only warranty under this Agreement, and in the absence of accident, abuse or misapplication, Logitech warrants, to the original licensee only, that the disk(s) on which the Software is recorded is free from defects in materials and workmanship under normal use and service for a period of ninety (90) days from the date of payment as evidenced by a copy of the receipt. Logitech's only obligation under this Agreement is, at Logitech's option, to either (a) return payment as evidenced by a copy of the receipt or (b) replace the disk that does not meet Logitech's limited warranty and which is returned to Logitech with a copy of the receipt. This warranty gives you limited, specific legal rights. You may have other rights, which vary from state to state.

The software and accompanying written materials (including the user's manual) are provided "as is" without warranty of any kind including the implied warranties of merchantability and fitness for a particular purpose, even if Logitech has been advised of that purpose. Logitech specifically does not warrant that the operation of the software will be liable for any direct, indirect, consequential or incidental damages arising out of the use of or inability to use such product even if Logitech has been advised of the possibility of such damages. Some states do not allow the exclusion or limitation of liability for consequential or incidental damages, so the above limitation may not apply.
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Introduction</strong></td>
<td></td>
</tr>
<tr>
<td>How LOGITECH Modula-2 is Organized</td>
<td>2</td>
</tr>
<tr>
<td>How to Read This Manual</td>
<td>3</td>
</tr>
<tr>
<td>LOGITECH POLICIES AND SERVICES</td>
<td>5</td>
</tr>
<tr>
<td>Other LOGITECH Products</td>
<td>6</td>
</tr>
<tr>
<td><strong>Installation</strong></td>
<td></td>
</tr>
<tr>
<td>What do you need?</td>
<td>8</td>
</tr>
<tr>
<td>What have you purchased?</td>
<td>9</td>
</tr>
<tr>
<td>Configure Your Operating System</td>
<td>10</td>
</tr>
<tr>
<td>LOGITECH Modula-2 on Floppy Diskettes</td>
<td>12</td>
</tr>
<tr>
<td>LOGITECH Modula-2 on a Hard Disk</td>
<td>16</td>
</tr>
<tr>
<td>Special Notes</td>
<td>22</td>
</tr>
<tr>
<td><strong>Chapter 1 A Tutorial</strong></td>
<td></td>
</tr>
<tr>
<td>Task 1: Get Ready</td>
<td>24</td>
</tr>
<tr>
<td>Task 2: Bring up the PTDEMO.MOD File</td>
<td>25</td>
</tr>
<tr>
<td>Task 3: M2ASSIST — The Syntax Checker</td>
<td>26</td>
</tr>
<tr>
<td>Task 4: M2ASSIST — The Compiler</td>
<td>27</td>
</tr>
<tr>
<td>Task 5: M2ASSIST — Find Next Error</td>
<td>28</td>
</tr>
<tr>
<td>Task 6: M2ASSIST — The Linker</td>
<td>30</td>
</tr>
<tr>
<td>Task 7: M2ASSIST — Run .EXE File</td>
<td>32</td>
</tr>
<tr>
<td>Review: The POINT Entry/Run Cycle</td>
<td>34</td>
</tr>
</tbody>
</table>
### Table of Contents

#### Chapter 2 Modula-2 for the Pascal Programmer 35

- **2.1 Types of Modules** .......................................................... 35
  - **2.1.1 Program Modules** .......................................................... 36
  - **2.1.2 Definition Modules** ...................................................... 36
  - **2.1.3 Implementation Modules** ......................................... 36
- **2.2 First Steps From Pascal To Modula-2** .......................... 38
- **2.3 More Differences** ............................................................. 39
- **2.4 Functions and Procedures** ................................................. 47
- **2.5 Use of Modules** ............................................................... 50
  - **2.5.1 User Definable Modules** ........................................... 50
  - **2.5.2 Importing Procedures with Identical Names** .......... 56
  - **2.7.3 Standard Library Modules** ...................................... 57

#### Chapter 3 The Compiler 69

- **3.1 How to Use the Compiler** ................................................. 70
- **3.2 Compiler Organization** ..................................................... 72
- **3.3 Compiler Output Files** ...................................................... 73
- **3.4 Compilation of a Program Module** .................................. 74
- **3.5 Compilation of a Definition Module** .............................. 77
- **3.6 Compilation of an Implementation Module** ..................... 78
- **3.7 Symbol Files Needed for Compilation** ............................ 79
- **3.8 Compiler Options** ............................................................. 80
  - **3.8.1 Table of Available Options** ....................................... 81
  - **3.8.2 Description of the Options** ....................................... 82
    - **3.8.2.1 The Optimize Option** ......................................... 86
- **3.9 Compiler Directives in Modules** .................................... 90
- **3.10 Compiler Messages** ........................................................ 91
  - **3.10.1 Source Text Errors** ................................................. 91
  - **3.10.2 Compiler Operational Messages and Errors** .......... 91
- **3.11 Compiler Table Limits** ................................................... 95
- **3.12 Compiler Error Messages** .............................................. 96

#### Chapter 4 Linking Modula-2 Files 101
### Table of Contents

#### Chapter 5 Version Checking 103

5.1 Module Keys and Version Checking ............................................................ 104
5.2 Version Errors and How to Fix Them .......................................................... 104
5.3 Version Errors During Compilation ............................................................. 105
5.4 Version Errors During Linking ................................................................. 106
5.5 Version Errors During Loading ................................................................. 108

#### Chapter 6 Interfacing Other Languages 109

6.1 Assembly Language with Modula-2 Conventions ........................................ 110
6.2 Use of Low-Level Features ........................................................................ 115
6.3 Foreign Definition ..................................................................................... 116
  6.3.1 Conventions Used.................................................................................. 116
    6.3.1.1 Symbol Name Convention............................................................ 117
    6.3.1.2 Parameter Passing Convention ..................................................... 118
    6.3.1.3 Default Data Segment When Using "C" Qualifier ....................... 118
    6.3.2 Examples of Foreign Definition Modules ............................................ 119
      6.3.2.1 "main" in Modula-2, and "foreign module" in C ...................... 119
      6.3.2.2 "main" in C and "foreign module" in Modula-2 ...................... 120
      6.3.2.3 "main" in Modula-2 and "foreign module" in Assembly .......... 121

#### Chapter 7 The Symbolic Post-Mortem Debugger 123

7.1 LOGITECH PMD Files .............................................................................. 125
7.2 The PMD and Your Hardware .................................................................. 125
7.3 How to Run the Post-Mortem Debugger .................................................... 126
7.4 PMD Configuration .................................................................................. 128
  7.4.1 Screen configuration.......................................................................... 128
  7.4.2 On-line Help ...................................................................................... 128
7.5 Post-Mortem Debugger Options ................................................................ 129
7.6 User interface ......................................................................................... 130
  7.6.1 Windows ........................................................................................... 130
  7.6.2 Mouse Functions ............................................................................... 132
  7.6.3 Keyboard Functions .......................................................................... 134
    7.6.3.1 How to scroll ............................................................................. 134
    7.6.3.2 Select a window object ............................................................... 134
    7.6.3.3 Call the menu ............................................................................ 135
    7.6.3.4 Respond to a prompt ................................................................. 136
    7.6.3.5 Move mouse cursor with the keyboard ................................... 136
# Table of Contents

7.7 Windows and Commands ................................................................. 137
  7.7.1 Call Window .............................................................................. 140
  7.7.2 Module Window .......................................................................... 142
  7.7.3 Data Window .............................................................................. 143
  7.7.4 Text Window ............................................................................... 148
  7.7.5 Raw Window ............................................................................... 149
  7.7.6 Message Window ......................................................................... 150
  7.7.7 Markers ........................................................................................ 150
  7.7.8 Selecting an Item for Display ...................................................... 150
  7.7.9 Relation between windows ........................................................ 150
    7.7.9.1 Update made from the Call window ....................................... 150
    7.7.9.2 Update made from the Module window ................................... 150
    7.7.9.3 Update from the Data and Text windows ................................ 150
  7.8 Consistency Checks ........................................................................ 151
  7.9 Messages ......................................................................................... 153

Chapter 8 Implementation Features .................................................. 157

8.1 System Dependent Facilities .......................................................... 158
  8.1.1 Language Extensions ................................................................... 159
  8.1.2 Address Arithmetic ..................................................................... 159
    8.1.2.1 Interpretation of Objects of Type ADDRESS ............................ 160
    8.1.2.2 Operations Involving Objects of Type ADDRESS ........................ 160
    8.1.2.3 Dereferencing Pointers ......................................................... 163
  8.1.3 The Module SYSTEM ................................................................. 164
    8.1.3.1 Constants Exported from Module SYSTEM ............................... 165
    8.1.3.2 Types Exported from Module SYSTEM .................................... 165
    8.1.3.3 Functions Exported from Module SYSTEM ............................... 166
    8.1.3.4 Procedures Exported from Module SYSTEM ............................. 167
  8.1.4 Data Representation .................................................................... 171
  8.1.5 Type Conversion and Type Transfer .......................................... 173
  8.2 Priorities and Interrupts ................................................................. 175
    8.2.1 Use of Priorities ........................................................................ 175
    8.2.2 Priority Levels .......................................................................... 176
    8.2.3 Interrupt Handling ..................................................................... 177
      8.2.3.1 Standard Method with IOTRANSFER .................................. 177
      8.2.3.2 Faster Method with IOTRANSFER ....................................... 177
      8.2.3.3 Low Level Interrupt Handling ............................................ 178
      8.2.3.4 How to Cope With Non-Reentrancy of MS-DOS .................. 180
8.2.4 Implementation Notes ................................................................. 181
  8.2.4.1 The Device Mask ................................................................. 181
  8.2.4.2 The Priority Masks ............................................................... 181
  8.2.4.3 The Interrupt Controller Mask ............................................. 182
  8.2.4.4 Monitor Entry and Exit ......................................................... 183
8.2.5 The Definition Module for "InputDevice" ...................................... 184
8.2.6 The Implementation Module for "InputDevice" ............................. 184

8.3 DOSCALL ........................................................................................ 187
  8.3.1 Extensions for DOS 2.0 ............................................................ 194

8.4 Decimals .......................................................................................... 201
  8.4.1 Internal and External Format ..................................................... 201
  8.4.2 Types ......................................................................................... 201
  8.4.3 Variables .................................................................................... 202
  8.4.4 Conversion and Status Procedures .......................................... 203
  8.4.5 Arithmetic Operations ............................................................... 204
  8.4.6 Pictures ..................................................................................... 205
  8.4.7 Picture Characters ..................................................................... 206
  8.4.8 Procedure StrToDec ................................................................. 206
  8.4.9 Procedure DecToStr ................................................................. 206
  8.4.10 Error Propagation ................................................................. 206

8.5 REAL Arithmetic ................................................................................ 208
  8.5.1 Simple Use of REAL Arithmetic ............................................... 209
  8.5.2 Choices for Using REAL Arithmetic ......................................... 210
  8.5.3 Accuracy of the Computations .................................................. 214
  8.5.4 Memory Requirements ............................................................. 214
  8.5.5 Performance ............................................................................ 215

8.6 RTS .................................................................................................. 216
  8.6.1 Organization ............................................................................ 216
  8.6.2 RTSMain .................................................................................. 217
  8.6.3 RTErrror .................................................................................. 217
  8.6.4 Language Dependent Facilities ............................................... 217
  8.6.5 Interrupt Handling ................................................................. 217
  8.6.6 I/O Device Control ................................................................. 217
  8.6.7 Arithmetic Coprocessors ......................................................... 217

8.7 Graphics .......................................................................................... 218
# Chapter 9 Libraries

## 9.1 Library Search Strategy

- **9.1.1 Default Names**
- **9.1.2 The Default Search Strategy**
- **9.1.3 The Query Search Strategy**

## 9.2 Library .DEF Files

- **ASCII**
- **BitBlockOps**
- **BitByteOps**
- **BitWordOps**
- **BlockOps**
- **Break**
- **Calendar**
- **CardinalIO**
- **Chronometer**
- **Conversions**
- **DateFormat**
- **DebugPMD**
- **DebugTrace**
- **Decimals**
- **Delay**
- **Devices**
- **Directories**
- **DiskDirectory**
- **DiskFiles**
- **Display**
- **DOS3**
- **DOS31**
- **DosError**
- **DOSMemory**
- **DurationOps**
- **DynMem**
- **ErrorCode**
- **Exec**
- **FileMessage**
- **FileNames**
- **FileSystem**
- **FloatingUtilities**
- **Graphics**
- **InOut**
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keyboard</td>
<td>293</td>
</tr>
<tr>
<td>LoadPath</td>
<td>294</td>
</tr>
<tr>
<td>LogiFile</td>
<td>295</td>
</tr>
<tr>
<td>LongIO</td>
<td>298</td>
</tr>
<tr>
<td>Lookup</td>
<td>299</td>
</tr>
<tr>
<td>MathLib0</td>
<td>300</td>
</tr>
<tr>
<td>Mouse</td>
<td>302</td>
</tr>
<tr>
<td>NumberConversion</td>
<td>307</td>
</tr>
<tr>
<td>Options</td>
<td>310</td>
</tr>
<tr>
<td>Overlay</td>
<td>312</td>
</tr>
<tr>
<td>Processes</td>
<td>314</td>
</tr>
<tr>
<td>Random</td>
<td>316</td>
</tr>
<tr>
<td>RealConversions</td>
<td>317</td>
</tr>
<tr>
<td>RealInOut</td>
<td>319</td>
</tr>
<tr>
<td>RS232Code</td>
<td>321</td>
</tr>
<tr>
<td>RS232Int</td>
<td>323</td>
</tr>
<tr>
<td>RS232Polling</td>
<td>325</td>
</tr>
<tr>
<td>RTSCoroutine</td>
<td>327</td>
</tr>
<tr>
<td>RTSDevice</td>
<td>328</td>
</tr>
<tr>
<td>RTSIntPROC</td>
<td>330</td>
</tr>
<tr>
<td>RTSM87</td>
<td>331</td>
</tr>
<tr>
<td>RTSMain</td>
<td>332</td>
</tr>
<tr>
<td>SimpleTerm</td>
<td>336</td>
</tr>
<tr>
<td>Sounds</td>
<td>337</td>
</tr>
<tr>
<td>Storage</td>
<td>338</td>
</tr>
<tr>
<td>Strings</td>
<td>340</td>
</tr>
<tr>
<td>Termbase</td>
<td>343</td>
</tr>
<tr>
<td>Terminal</td>
<td>346</td>
</tr>
<tr>
<td>TimeDate</td>
<td>348</td>
</tr>
<tr>
<td>9.3 Library Cross Reference</td>
<td>351</td>
</tr>
</tbody>
</table>
# Table of Contents

Appendix A Modula-2 Bibliography 371

Books..................................................................................................................... 371
Magazines ............................................................................................................ 375
User Groups ......................................................................................................... 376

Appendix B Memory Organization & Run Time Description 377

B.1 Global Memory Organization........................................................................... 377
B.2 Subprograms and Resident Overlays ............................................................ 379
B.3 Program Execution ........................................................................................ 381
B.4 Processes........................................................................................................ 382
B.5 Allocation of Variables.................................................................................... 384
B.6 The Heap ........................................................................................................ 385
B.7 The Stack........................................................................................................ 385
B.8 The Procedure Activation Record .................................................................. 386
B.9 Procedure Calling Conventions .................................................................... 389
B.10 Function Results .......................................................................................... 391
B.11 Symbols in .OBJ Files ............................................................................... 392
B.12 Aborting LOGITECH Modula-2 Programs ............................................... 393
B.13 Command Line Arguments ......................................................................... 394

Appendix C Technical Tips 395

C.1 PrintTimeDate ............................................................................................... 396
C.2 Printing .......................................................................................................... 397
C.3 The Screen ..................................................................................................... 398
C.4 RedirectInput ................................................................................................. 399

Appendix D Product Support Plan 401

Glossary 403

Index 407
LOGITECH MODULA-2
Introduction

IMPORTANT

See the README file on Disk 1 for late breaking news about this version of LOGITECH Modula-2.
Introduction

How LOGITECH Modula-2 is Organized

This manual assumes that you are familiar with the basics of DOS and with basic programming concepts and terminology.

If you are a beginner, work through the tutorial to familiarize yourself with LOGITECH Modula-2. Then consult the bibliography in Appendix A, for books on Modula-2.

If you are more experienced, and perhaps familiar with Modula-2's predecessor, Pascal, read through Chapter 2, Modula-2 for the Pascal Programmer for an introduction to the implementation-specific features of LOGITECH Modula-2, and later as a reference for specific questions and problems which may arise.

This manual features:

- Introductory information which includes system requirements and installation instructions.
- A step-by-step tutorial through the system, with the POINT Editor connecting you to the various parts of the LOGITECH Modula-2 world.
- An overview of Modula-2, which explains how it differs from and is similar to Pascal, with primary features of the language.
- Complete instructions for the LOGITECH Compiler.
- Complete instructions for the LOGITECH Symbolic Post-Mortem Debugger.
- Information on version checking for orderly program development.
- How to use non-Modula-2 modules in LOGITECH Modula-2 programs.
- A description of Modula-2 implementation features.
- How to help LOGITECH Modula-2 utilities find Library information.
- A complete listing of the LOGITECH Modula-2 .DEF files, with a cross-referenced index.
- Information on memory organization and run-time organization.
- Technical tips.
- A bibliography.
- A LOGITECH Modula-2 glossary.
- An index.
How to Read This Manual

The following conventions are used in this manual:

Keys to be pressed, look like this:

[Y]    [Esc]    [↓]

Control sequences or characters entered with a **Control** or **Shift** key, look like this:

[Ctrl]-C    [Ctrl]-Break    [Shift]-F2    [Alt]-X

Keys from the **Numeric Keypad** are shown like this:

[↑]    [↓]    [←]    [→]

[PgUp]    [PgDn]    [+]    [−]

Keyboard input for the **DOS** Command line is in upper case and looks like this:

**M2L**  [→]

Mouse buttons used are based on the **LOGITECH** standard, and use three buttons, e.g,

[■]    [□]    [■]  means press the left mouse button,

[□]    [■]    [□]  means press the right mouse button, and

[□]    [■]    [■]  means press the middle mouse button.

[■]    [□]  means press both buttons on a two button mouse.

Variable names in the text are surrounded by angle brackets, as in

<Application name>  [→]

File names look like this:

**M2L.EXE**

**DOS** commands and statements look like this:

**PATH, COPY**

Product names look like this:

**MS DOS, LOGITECH Modula 2**
Reserved words, predefined functions, and user-defined functions in LOGITECH Modula-2 look like this when being discussed in text:

**PROCESS, VAL, MyFunction**

These are not emphasized in screen display or program listings.

Screen output and some listings look like this:

**Program Not Found**

Program source code looks like this:

```plaintext
IF condition THEN
    statement6;
ELSIF condition THEN
    statement7;
ELSE
    statement8;
END;
```

Sample Screens look like this:

```
Text line#  32 Demo.MOD

PROCEDURE RecursiveOne (x: CARDINAL; y:REAL; z: INTEGER
BEGIN
    WITH node[x] Do
        data1 := x;
        data2 := y;
        data3 := z;
    END; (* WITH *)
    INC(x);
    y := Y + 1.0;

Data Demo Module

| x     | CARDINAL  | +++Demo
|-------|-----------|---------
| y     | 2.0000000000E+000 | Reals
| z     | INTEGER   | Terminal
| node  | ARRAY[1..4] OF RECORD | Keyboard
```

---

```
Call breakpoint

>RecursiveOne
>RecursiveOne
>FirstOne
>Initialization
>PROCESS
```

```
Raw | Help|F1 | messages
```
LOGITECH POLICIES AND SERVICES

We know that effective communication with our customers is the key to quality service. Therefore we have set up the LMIS (LOGITECH Mouse Information Service), an electronic bulletin board where you can contact us at your convenience. To reach the LMIS, dial:

(415) 795-0408

using a 300, 1200 or 2400 baud modem.

The menu of available options is self explanatory.

LOGITECH also sponsors an electronic conference on BIX, the BYTE INFORMATION EXCHANGE system from Byte magazine. If you have access to BIX, join us in

conference LOGITECH,

and communicate with us there.

For all LOGITECH Modula-2 users, including ISVs (Independent Software Vendors), 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. If you are an ISV, we encourage you to join the impressive list of developers who use LOGITECH Modula-2 to design application software. Call us for details.

The Modula-2 User Association (MODUS) is another important source of information about the 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
Introduction

Other LOGITECH Products

At LOGITECH we pride ourselves on technical excellence and advanced engineering. In addition to LOGITECH Modula-2, we also offer these fine products which we believe to be the most advanced in their product category.

LOGITECH Modula-2 Toolkit

The LOGITECH Modula-2 Toolkit offers these Modula-2 functions:

- **M2FORMAT**, the LOGITECH Modula-2 source code formatter.
- The **LOGITECH Linker**, the optimal linker for LOGITECH Modula-2.
- The **LOGITECH RTD**, or Symbolic Run-Time Debugger.
- **M2DECOD**e dissambles .OBJ files for LOGITECH Modula-2 code.
- **M2VERS** keeps track of LOGITECH Modula-2 development versions.
- **M2XREWF** generates cross-referenced tables of Modula-2 source files.
- **M2MAKE** creates batch files with all parameters and search strategies needed to recompile and relink changes made to your source code.
- **M2CHECK** helps you streamline code by referencing unused or questionable library items.

Other LOGITECH Modula-2 Utilities

- The **LOGITECH Turbo-Pascal To Modula-2 Translator**.
- A VAX/VMS version of LOGITECH Modula-2.

Site licences are available for all LOGITECH Modula-2 products.

The LOGITECH C7 Mouse

The LOGITECH C7 Mouse connects to a serial port in your computer. It needs no pad and no external power supply.

The LOGITECH Bus Mouse

The LOGITECH Bus Mouse is equivalent to the LOGITECH C7 Mouse, except that it is connected to a Bus Board which you insert in your computer. It needs no pad and no external power supply.

For additional information, or to order these products, call the LOGITECH sales office toll-free from anywhere in the continental U.S. at (800)231-7717, or in California, call (800) 552-8885.
Installation

This chapter tells you how to install the LOGITECH Modula-2 Development System under DOS, as well as how to optimize the system for your use.

Instructions are given for installation to a set of floppy diskettes, as well as to a hard disk. In addition, this chapter tells you which floppy diskette to use while developing your programs on a floppy diskette system.

Help in the form of batch files is provided on Disk 1. You can modify these batch files, or install the system manually if your system differs from the assumed standard.

---

NOTE

Remember to read the READ.ME file on Disk #1 for late breaking information that may not have been available when this manual went to press.
**Installation**

**What do you need?**

*LOGITECH Modula-2* runs on an *IBM PC, XT, AT* or compatible computers, with:

- A floppy disk drive A, and either
  - A floppy disk drive B, or
  - A hard disk drive C.

In addition:

- To run M2C.EXE (the fully-linked version of the compiler), you must have **512 K Bytes** of RAM memory.

- To run M2COMP.EXE (the overlay version of the compiler), you must have **290 K Bytes** of RAM memory.
What have you purchased?

Manuals

If you purchased LOGITECH Modula-2 by itself, you have a copy of the POINT Editor User's Manual, in addition to the LOGITECH Modula-2 User's Manual you are now reading.

If you purchased the LOGITECH Modula-2 Development System, then you also have a copy of the LOGITECH Modula-2 Toolkit Manual.

Diskettes

If you purchased LOGITECH Modula-2 by itself, you received five diskettes

- Disk 1: POINT Editor
- Disk 2: Standard Library
- Disk 3: Fully-Linked Compiler
- Disk 4: Overlay Compiler
- Disk 5: Post-Mortem Debugger

If you purchased the LOGITECH Modula-2 Development System which includes the LOGITECH Modula-2 Toolkit, you also received four additional diskettes:

- Disk 1: Standard Library Sources I
  Utilities I
- Disk 2: Standard Library Sources II
- Disk 3: Linker
  Utilities II
- Disk 4: Run-Time Debugger
Configuration Your Operating System

In order to run LOGITECH Modula-2, you need to configure your operating system. This is done by setting up the CONFIG.SYS file on the disk from which you start your operating system. The INSTALL procedure makes the necessary additions with the proper settings to your CONFIG.SYS file, and saves the backup file as CONFIG.OLD. Then it tells you how to enlarge your environment space, according to the version of DOS you are using.

FILES=20

The number of files that can be open at the same time. A value of twenty (20) or more is needed to operate the LOGITECH Modula-2 Development System.

If the right number of files is not set in DOS, the error message file not found will appear when you try to run LOGITECH Modula-2. This means that even though the file you want may be present on the disk, it can’t be opened due to a lack of file descriptors in the operating system.

BUFFERS=20

We recommend that you set the number of buffers to twenty (20). An appropriate value will increase the performance of the LOGITECH Modula-2 system. However, this is not a requirement and you may omit this statement.

DEVICE=ANSI.SYS

This statement provides access to Extended Screen and Keyboard Control provided by DOS. Some parts of LOGITECH Modula-2 assume that this driver is used. If you omit this statement in CONFIG.SYS, certain control characters written to the display may not have the effect specified in the Terminal definition module.
Installations

**SHELL=COMMAND.COM/P/E:n** (for **DOS** 3.1 and higher)

Whenever you start your computer, **DOS** allocates a small amount of bytes (e.g., 160) for environment space. Often, this is not enough for the **LOGITECH Modula-2** settings.

An additional line in your **CONFIG.SYS** file can correct this:

**SHELL=COMMAND.COM/P/E:n**

If you are using **DOS 3.1** or **DOS 3.2**, add one of these lines:

- For **DOS 3.1**, *n* is the **number of 16 byte paragraphs** allocated for environment space. We recommend that you add this line:

  **SHELL=COMMAND.COM/P/E:31**

- For **DOS 3.2** or higher, *n* is the **number of bytes** allocated for environment space. We recommend:

  **SHELL=COMMAND.COM/P/E:500**

---

**REMEMBER**

After you create or change your **CONFIG.SYS** file, you must restart your system for the changes to take effect.
LOGITECH Modula-2 on Floppy Diskettes

---

**NOTE**

Before you install your software to either floppy drive or hard disk system, we strongly recommend that you take a minute to:

1) Put Write-Protect tabs on all your LOGITECH Modula-2 diskettes, and
2) Use the DISKCOPY and DISKCOMP commands from your DOS files to back up your diskettes. Then put your original diskettes in an archival area and use the copies for all installation.
3) Prepare formatted diskettes with readable labeling, before you copy the files in the Installation procedure which follows.

---

**Installing on Floppy Diskettes**

If your LOGITECH Modula-2 Development System is going to be installed on a dual floppy system, we recommend this organization:

**Step 1:** Prepare a POINT Editor diskette for your Modula-2 system.

Insert the LOGITECH Modula-2 POINT Editor (Disk #1) into drive B, and an empty, formatted disk into drive A. Type,

```
COPY B:PT**.* A:   
COPY B:CHECKER.* A: 
COPY B:M2ASSIST.* A: 
```

This will provide you with the LOGITECH editing environment of choice for your LOGITECH Modula-2 Development System.

**Step 2:** Prepare a Library diskette for your Modula-2 system.

Insert the LOGITECH Modula-2 Library (Disk #2) into drive B, and an empty, formatted disk into drive A. Type,

```
COPY B:.*.* A:     
```

You will use this Library Disk when linking. Some additional .LIB files are on LOGITECH Disk 5 if you want to interface C language.
Step 3: Prepare a Fully-Linked Compiler diskette for your Modula-2 system.

Insert the *LOGITECH Modula-2* Fully-Linked Compiler (Disk #3) into drive B, and an empty, formatted disk into drive A. Type,

```
COPY B:*. A: [enter]
```

You will use this disk to compile *LOGITECH Modula-2* programs. This version of the Compiler requires 512 KBytes of internal (RAM) memory.

Step 4: Prepare an Overlay Compiler diskette for your Modula-2 system.

Insert the *LOGITECH Modula-2* Overlay Compiler (Disk #4) into drive B, and an empty, formatted disk into drive A. Type,

```
COPY B:*. A: [enter]
```

You will use this disk to compile larger *LOGITECH Modula-2* programs. This version of the Compiler requires 290 KBytes of internal (RAM) memory.

Step 5: Prepare a Post-Mortem Debugger diskette for your Modula-2 system.

Insert the *LOGITECH Modula-2* Post-Mortem Debugger (Disk #5) into drive B, and an empty, formatted disk into drive A. Type,

```
COPY B:*.REF A: [enter]
COPY B:*.CFG A: [enter]
COPY B:*.EXE A: [enter]
```

This disk will help you debug crashed *LOGITECH Modula-2* programs.

Step 6: Prepare a Working diskette for your Modula-2 system.

Insert the *LOGITECH Modula-2* POINT Editor (Disk #1) into drive B, and an empty, formatted disk (which will become your Working diskette) into drive A. To copy all the files with the .SYM extension from Disk 2 onto your Working diskette, type,

```
COPY B:*.SYM A: [enter]
```

You will use this disk to hold .MOD and .DEF files to be compiled and linked.
Installation

PATH and Environment Variables

For LOGITECH Modula-2 to work properly with a floppy diskette system, you need some additional DOS commands in your AUTOEXEC.BAT file, which must be in the root directory of your boot disk. If you do not yet have such a file, create it in the root directory, using your text editor.

These commands assume that you have your Modula-2 Working Diskette in drive A, and a compiler, linker, debugger, or utilities diskette in drive B. Append these to your current AUTOEXEC.BAT file, or create an AUTOEXEC.BAT which includes the following commands:

```
SET M2SYM=A:\;
SET M2OBJ=A:\;B:\;
SET M2LIB=A:\;B:\;
SET M2REF=A:\;B:\;
SET M2MOD=A:\;B:\;
SET M2MAP=A:\;B: \;
```

These set the environment for LOGITECH Modula-2 in a dual floppy configuration. They let your LOGITECH Modula-2 system take full advantage of DOS. More on the environment variables used by LOGITECH Modula-2 can be found in the section of this manual on the library search strategy.

You must also set the DOS environment PATH statement (refer to your DOS Manual). DOS uses this statement DOS to search for .EXE files. If your Modula-2 Working diskette is in drive A and the compiler/linker/debuggers/utilities disk in drive B, the environment variable PATH should contain the following string:

```
PATH=A:\;B:\;
```

---NOTE---

Before you use LOGITECH Modula-2, be sure to re-start your system so the AUTOEXEC.BAT commands can take effect.
Running on Floppy Diskettes

When you work with *LOGITECH Modula-2* on floppy diskettes, use drive A for your *Modula-2 Working diskette*. It should contain these files:

- .SYM files from the *LOGITECH Modula-2 Standard Library*.
- .MOD and .DEF files for *Modula-2* source text you have created.
- Other files you may create with *LOGITECH Modula-2*, to use for compiling, linking or debugging.

While you are developing your programs, drive B holds a disk for what you are doing. This disk must include the appropriate *LOGITECH Modula-2* system files:

**To Edit a source file**
Insert a working copy of the *POINT Editor* disk in drive B.

**To Compile a source file**
Depending on the amount of memory in your machine, insert a working copy of one of the compiler disks in drive B.

**To Link a .OBJ file**
Insert a working copy of the Library disk in drive B;
Insert a working copy of the *LOGITECH Linker* in drive A; or
Insert a working copy of the *DOS Linker* in drive A.

**To Run a Debugger**
Insert a working copy of one of the *LOGITECH Debugger* disks in drive B.

**To Run Utilities**
Insert a working copy of a *LOGITECH* utility disk in drive B.

---

**NOTE**

Depending on the capacity of your disks, you can include two or more of the disks mentioned above onto one disk.

If you are using high density diskettes, study the following section on hard disk systems, on the environment variables used by *LOGITECH Modula-2*, and also study the section on library search strategy.
LOGITECH Modula-2 on a Hard Disk

The most convenient way to use LOGITECH Modula-2 is with a hard disk. This section tells you two ways to copy files from the distribution disks to your hard disk.

You can copy all the files into the same directory where you write your LOGITECH Modula-2 programs. However, it's better to take advantage of the structured directory system in DOS. This reduces the number of files in your directories and, at the same time, lets you use LOGITECH Modula-2 from any directory.

You can also install the LOGITECH System to a directory of your choice.

Just make sure that you are on the hard disk where you want to keep your Modula-2 files, and that Disk #1 is in Drive A.

Step 1: Create a Development Environment Directory.

To install LOGITECH Modula-2 in a directory of your choice, type,

```
MD \YOUR_DIR
```

Step 2: Run the Install Program

Insert Disk #1 in Drive A.

a) To install the LOGITECH Modula-2 files in your root directory, type

```
A: INSTALL
```

b) To install LOGITECH Modula-2 in a directory of your choice, type,

```
A: INSTALL \YOUR_DIR
```

The INSTALL program then tells you which disks to insert.
Step 3: Add Configuration Statements.

After you install the LOGITECH Modula-2 Development System, you are prompted to add some additional statements to your CONFIG.SYS and AUTOEXEC.BAT files.

An additional line in your CONFIG.SYS file can correct this:

```sh
SHELL=COMMAND.COM/P/E:n
```

If you are using DOS 3.1 or DOS 3.2, add one of these lines:

- For DOS 3.1, n is the number of 16 byte paragraphs allocated for environment space. We recommend that you add this line:
  ```sh
  SHELL=COMMAND.COM/P/E:31
  ```

- For DOS 3.2, n is the number of bytes allocated for environment space. We recommend this line:
  ```sh
  SHELL=COMMAND.COM/P/E:500
  ```

After you add these statements, restart your system for them to take effect when you work on your Modula-2 programs.

---

**NOTE**

INSTALL.BAT copies your AUTOEXEC.BAT and CONFIG.SYS files to AUTOEXEC.OLD and CONFIG.OLD, and then adds various statements to your AUTOEXEC.BAT and CONFIG.SYS files.

If you need to use the previous settings, simply rename these files with their original extensions.

If you do this, you may also want to save your Modula-2 AUTOEXEC.BAT and CONFIG.SYS files, too — under names you can use to reinstall your LOGITECH Modula-2 system, as needed.
Installation

The following procedure copies files from *LOGITECH Modula-2* diskettes to a subdirectory you have chosen.

If your system has special constraints, such as directory names that conflict with those used by *LOGITECH Modula-2*, you can install environment step-by-step. You can, of course, copy these files to any directory you choose, as long as you specify their PATH in your AUTOEXEC.BAT file.

Perform these *DOS* commands:

**Step 1:** Create a directory for your development environment.

To install *LOGITECH Modula-2* files in a directory of your choice, use the MKDIR (MD) command to create that directory.

**Step 2:** Create subdirectories for *LOGITECH Modula-2*.

From the system prompt in your chosen subdirectory (e.g., `\YOUR_DIR`), perform these *DOS* commands:

<table>
<thead>
<tr>
<th>DOS Command</th>
<th>What it does</th>
</tr>
</thead>
<tbody>
<tr>
<td>MKDIR M2EXE</td>
<td>Creates the <code>\YOUR_DIR\M2EXE</code> directory.</td>
</tr>
<tr>
<td>MKDIR M2LIB</td>
<td>Creates the <code>\YOUR_DIR\M2LIB</code> directory.</td>
</tr>
<tr>
<td>MKDIR M2TMP</td>
<td>Creates the <code>\YOUR_DIR\M2TMP</code> directory.</td>
</tr>
<tr>
<td>CD M2LIB</td>
<td>Goes to the <code>\YOUR_DIR\M2LIB</code> directory.</td>
</tr>
<tr>
<td>MKDIR DEF</td>
<td>Creates the <code>\YOUR_DIR\M2LIB\DEF</code> directory.</td>
</tr>
<tr>
<td>MKDIR LIB</td>
<td>Creates the <code>\YOUR_DIR\M2LIB\LIB</code> directory.</td>
</tr>
<tr>
<td>MKDIR MAP</td>
<td>Creates the <code>\YOUR_DIR\M2LIB\MAP</code> directory.</td>
</tr>
<tr>
<td>MKDIR MOD</td>
<td>Creates the <code>\YOUR_DIR\M2LIB\MOD</code> directory.</td>
</tr>
<tr>
<td>MKDIR OBJ</td>
<td>Creates the <code>\YOUR_DIR\M2LIB\OBJ</code> directory.</td>
</tr>
<tr>
<td>MKDIR REF</td>
<td>Creates the <code>\YOUR_DIR\M2LIB\REF</code> directory.</td>
</tr>
<tr>
<td>MKDIR SYM</td>
<td>Creates the <code>\YOUR_DIR\M2LIB\SYM</code> directory.</td>
</tr>
<tr>
<td>CD ..</td>
<td>Return to <code>\YOUR_DIR</code> subdirectory</td>
</tr>
</tbody>
</table>
Step 3: Copy the POINT Editor Files.

Insert Disk #1 with the POINT Editor into drive A and type:

COPY A:PT*. * M2EXE
COPY A:M2ASSIST.* M2EXE
COPY A:CHECKER.* M2EXE
COPY A:*.SYM M2LIB\SYM

This copies files on Disk #1 to the appropriate directories under \YOUR_DIR.

Step 4: Copy the Library Files.

Insert Disk #2 with the Standard Library files into drive A and type:

COPY A:*.LIB M2LIB\LIB
COPY A:*.OBJ M2LIB\OBJ

This copies .LIB files from Disk #2 to \YOUR_DIR\M2LIB\LIB, and .OBJ files to \YOUR_DIR\M2LIB\OBJ.

Step 5: Copy the Fully Linked Compiler Files.

Insert Disk #3 with the Fully Linked Compiler into drive A and type:

COPY A:*.* M2EXE

This copies the Fully Linked Compiler and other files from Disk #3 to \YOUR_DIR\M2EXE.

Step 6: Copy the Overlay Compiler Files.

Insert Disk #4 with the Overlay Compiler into drive A and type:

COPY A:*.* M2EXE

This copies the Overlay Compiler files from Disk #4 to \YOUR_DIR\M2EXE.

Step 7: Copy the Post-Mortem Debugger Files.

Insert Disk #5 with the Post-Mortem Debugger files into drive A and type:

COPY A:*.LIB M2LIB\LIB
COPY A:*.REF M2LIB\REF
COPY A:*.CFG M2EXE
COPY A:*.EXE M2EXE

This copies Post-Mortem Debugger files from Disk #5 to \YOUR_DIR\M2EXE.

Step 8: Adjust your PATH statement.

Read the next section carefully.
Installation

PATH and Environment Variables

For *LOGITECH Modula-2* to work properly with a hard disk system, you need some additional *DOS* commands in your *AUTOEXEC.BAT* file. *AUTOEXEC.BAT* executes various commands automatically every time you start your system. If you don’t yet have this file, create it with your *POINT* editor in the root directory.

SET Statements

Append the following to your *AUTOEXEC.BAT* file:

```
SET M2SYM=C:\M2LIB\SYM;
SET M2OBJ=C:\M2LIB\OBJ;
SET M2REF=C:\M2LIB\REF;
SET M2MOD=C:\M2LIB\MOD;
SET M2LIB=C:\M2LIB\LIB;
SET M2MAP=C:\M2LIB\MAP;
SET M2TMP=C:\M2TMP;
```

These settings are correct for a set of *LOGITECH Modula-2* directories created at the root of your hard disk. However, if you create your directories *from a subdirectory*, you must *add that subdirectory name* to the PATH statement as well as to the SET statements shown above. If your *LOGITECH Modula-2* environment is attached to the "YOUR_DIR" subdirectory, for example, then you must preface your SET specification accordingly, as in:

```
SET M2SYM=C:\YOUR_DIR\M2LIB\SYM;
```

etc., for each SET specification.

*Note:* The fictitious directory name `\YOUR_DIR` has been added to the statement in this example.
NOTE

These statements set up the environment variables for LOGITECH Modula-2, and let the LOGITECH Modula-2 system take full advantage of your hard disk. More information on the environment variables used by LOGITECH Modula-2 are found in this manual in Section 9.1, Library Search Methods.

The PATH Statement

You must also set the DOS environment statement PATH (refer to your DOS Manual). PATH is used by DOS to search for .EXE files. For example, if you keep .EXE files in drive C in a directory named COMMANDS, the PATH statement PATH will read:

PATH=C:\COMMANDS;

After you use the INSTALL program to add the M2EXE directory it should read:

PATH=C:\COMMANDS;C:\M2EXE;

Remember

Before using LOGITECH Modula-2, be sure to re-start, re-boot, your system, so the commands of the AUTOEXEC.BAT file will be executed.
Installation

Special Notes

A Word About the POINT Editor

The POINT Editor has provided the LOGITECH Modula-2 Development Group with its editing environment of choice. Apart from the installation instructions you have just read, the POINT Editor can also be placed in its own directory. In this way, you can keep just the PT.INI file of your choice in your working directory. For more information see the POINT User’s Manual.

The POINT diskette includes a file named PTM2.INI which can be copied into PT.INI, the Initialization file for the POINT Editor.

A Word About the MOD Editor

If you are more accustomed to the MOD Editor from previous releases, we have included a file named PTMOD.INI. You can copy this file to PT.INI on your POINT working disk or directory, for the accustomed MOD interface with the additional features of LOGITECH Modula-2, Version 3.0.

A Word About Mice

We recommend either the LOGITECH Bus Mouse or the LOGITECH C7 Mouse for editing source text files and for running either the LOGITECH Symbolic Post-Mortem Debugger or the LOGITECH Symbolic Run-Time Debugger. We know you will find either LOGITECH Mouse an ideal tool for the LOGITECH Modula-2 Development System, as well as for other editing and graphics tasks.

Be sure to load your mouse driver software before you attempt to edit or debug your Modula-2 work. Instructions for installing mouse software is in your mouse manual.
Chapter 1
A Tutorial

The best way to learn a new language is to use it. The same is true for programming languages.

In this chapter you will enter and run a simple Modula-2 program. It’s like getting acquainted with a city by looking at a map and a tour book: it can get you started. If you want adventures, you will then know just enough to get into trouble and also just enough to figure your way out of trouble.

This tutorial uses the POINT Editor with the m2assist extension menu, which provides a fast, flexible development environment, whether you are a beginner or an experienced programmer. POINT is easy to learn, easy to use, and is the LOGITECH Modula-2 development environment of choice.

The best environment will also include a LOGITECH Mouse, since the POINT Editor optimizes the three button standard provided by LOGITECH. If you do not yet have a mouse, or are operating temporarily on a non-mouse system, the POINT Manual has a special summary for mouse emulation with POINT on a standard keyboard.

---

NOTE

Before you begin this tutorial, be sure you have backed up your LOGITECH Modula-2 diskettes and installed the system, and that the PATH and SET statements are written as described in the preceding Installation section.
Chapter 1

Task 1: Get Ready

This tutorial assumes that you have used the default installation described in the Installation section, above. If you are already using a directory with this name for other purposes, choose another directory name, and adjust the following instructions accordingly.

Step 1: Move to Your Development Directory.

From wherever you are, on the command line, type

```
CD\YOUR_DIR
```

where \YOUR_DIR is the path and directory you have specified in the Installation.

Step 2: Move to the M2TMP directory.

Type

```
CD M2TMP
```

This takes you to the \YOUR_DIR\M2TMP directory

For this tutorial we will work from the M2TMP sub-directory. Be sure that your POINT files are in a directory that is listed in your PATH statement. Refer to the Installation section, in the front of this manual.

Step 3: Bring the PTDEMO.MOD file into the M2TMP sub-directory.

Use the DOS COPY command, or a file management utility to copy this file into the YOUR_DIR\M2TMP directory.

---

**NOTE**

Using \M2TMP to create your Modula-2 programs is a suggestion and not a requirement of LOGITECH Modula-2. You can just as easily create and/or use any other directory.
Task 2: Bring up the PTDEMO.MOD File

Again, you can create Modula-2 files with any general purpose text editor. However, to run the syntax checker you must be in the POINT environment. To load POINT, type

```
PT PTDEMO.MOD
```

Here is the POINT screen you will see, with a listing for PTDEMO.MOD as it appears in a window in the POINT environment.

```mod
open close HELP *next prev *WINDOW EDITING MOVING QUIT+ETC OPTIONS M2ASSIST

MODULE PtDemo;

FROM Terminal IMPORT writeString, WriteLn;

PROCEDURE Hello (str : ARRAY OF CHAR);

VAR i : CARDINAL;

BEGIN
  FOR i := 1 TO 10 DO
    writeString(str);
    WriteLn;
  END
END Hello

BEGIN
  Hello("Hello Everybody"," !!!")
END PtDemo.
```

For more information on POINT, see the Tutorial in the POINT Editor User's Manual.
Task 3: M2ASSIST — The Syntax Checker

M2ASSIST is composed of several functions, some of which are executed from the M2ASSIST menu at the top of the POINT screen, and some of which use the [Alt] key in combination with one of the twenty-six alphabetical keys.

The first function we will use is Check Syntax, which appears in the M2ASSIST menu.

Step 1: Check the Syntax of PTDEMO.MOD.

Press [ ] on the M2ASSIST menu and use the highlight to execute the Check Syntax. The syntax checker stops at BEGIN of the main program and the following message appears on the status line:

',; expected at 'BEGIN'

Actually, a semicolon is missing after END Hello.

Step 2: Correct the PTDEMO.MOD listing.

Place a semicolon after Hello, as in the partial listing below. The corrected listing should read:

```
... WriteIn
  END
  END Hello;
BEGIN
...
```

Now run Check Syntax again, and you will see the message:

Syntax checker: no errors found

This means you are ready now to use the LOGITECH Modula-2 Compiler which takes PTDEMO.MOD and creates PTDEMO.OBJ.
Task 4: M2ASSIST — The Compiler

The next task is to attempt to compile PTDEMO.MOD.

When compilation is done, any key returns you to the source file in the editing window.

Step 1: Run the compiler.

Press on the M2ASSIST menu and drag the cursor down the menu to select Compile. A message at the bottom of the screen asks for options. Press to accept the default options.

The screen is redrawn as follows:

```
m2comp C:\M2TMP\ptdemo.mod
LOGITECH Modula-2 Compiler, 8086, MS-DOS OBJ-file, Rel. 3.00, (C), Aug 87
Copyright (C) 1983, 1987 LOGITECH, Inc.

  source file> C:\M2TMP\ptdemo.mod

  Syntax and Declaration Analysis
  Terminal In file: \M2LIB\SYM\Terminal.SYM
  Block Analysis
  ---- error
  Listing Generation
     13  Hello("Hello Everybody","!!!")
     ****
     * 132: too many parameters
  Termination
  End Compilation

  press any key to continue
```

Step 2: Return to the editing session.

Press any key to return to the editing session. You will see the listing file you left when you selected the Compile command. In addition, at the bottom of the screen, you will see a message that reads:

```
1 COMPILER ERROR.
```
Chapter 1

Task 5: M2ASSIST — Find Next Error

Errors found during Compilation can be found quickly in your source code with Find Next Error. Find Next Error highlights the next statement tagged with a compilation error after invoking Compile or Syntax checking.

You can insert/delete lines of text in the file and then go to the next tagged error.

Corresponding error messages from the .LST file are shown in a temporary window. The error window closes at the first action.

Step 1: Look for the error.

Press (M) on the M2ASSIST menu and drag the cursor down the menu to select Find Next Error.

The screen shifts to the general area of the error which is highlighted, and a box appears at the bottom of the screen with the following message:

```
15   Hello("Hello Everybody","!!!")
*****    ^132
* 132: too many parameters
```

Step 2: Correct the source code.

Move to the area in the source code that is highlighted. As soon as you press a key or move to the highlighted text the box with the error message dissappears, leaving the highlighted text.

The source of the error is two string parameters where only one is allowed. Remove the error (","") between Everybody and !!! in the line hello("Hello Everybody","!!!").

The corrected line will read:

```
Hello("Hello Everybody!!!")
```
Step 3: Look for the next error.

Again, press \[0 0\] on the M2ASSIST menu and drag the cursor down the menu to select **Find Next Error**.

That was apparently the only error, since the message on the status line at the bottom of the screen reads:

**NO MORE ERRORS**

Step 4: Compile the source code again.

Now press \[0 0\] on the M2ASSIST menu and drag the cursor down the menu to again select **Compile**.

The screen you get looks like the following:

```plaintext
m2comp C:\M2TMP\ptdemo.mod
LOGITECH Modula-2 Compiler, 8086, MS-DOS OBJ-file, Rel. 3.00, (C), Aug 87
Copyright (C) 1983, 1987 LOGITECH, Inc.
source file> C:\M2TMP\ptdemo.mod

Syntax and Declaration Analysis
Terminal in file: \M2LIB\SYM\Terminal.SYM
Block Analysis
Code Generation
Termination
The interactive setting of the options was: S+ /R- /T+ /A- /O+ /F+
Code for 8086/8088 generated
Codesize: 155 bytes Datasize: 0 bytes
End Compilation

/> press any key to continue
```

Step 5: Return to the POINT Window.

As soon as you press a key to return to your source code in the **POINT** window, you find the following message at the bottom of the screen.

**no listing file was produced during the compile.**
Chapter 1

Task 6: M2ASSIST — The Linker

When PTDEMO.MOD is successfully compiled, an additional file named PTDEMO.OBJ is created by the compiler.

A Linker uses PTDEMO.OBJ to generate PTDEMO.EXE.

There are two ways to link an .OBJ file: use the M2ASSIST menu with the LOGITECH Linker; or choose Exit to DOS Shell from the QUIT&ETC menu, and use a DOS or other linker. If you use other than the LOGITECH Linker, include the directory with your linker in the PATH statement, as mentioned in Installation.

Link with the LOGITECH Linker

Step 1: Link the .OBJ file.

Press [ ] on the M2ASSIST menu and drag the cursor down the menu to select Link. Press [ ] to accept the default Linker options.

The screen is redrawn as follows:

```
press any key to continue
```

You are now ready to run the PTDEMO program you just linked with the LOGITECH Linker.
Link with the DOS Linker

If you have not yet purchased the LOGITECH Modula-2 Toolkit, use the linker that came with your copy of DOS.

Step 1  Select a DOS Shell.

Press [Q D E C] on the QUIT+ETC menu and drag the cursor down the menu to select Execute DOS Shell.

Step 2  Link from DOS

The screen below shows the prompts you will see as you link your copy of PTDEMO.OBJ. Information to be entered is in bold face, and finalized with [Q J], unless there is no response to be entered in which case press [Q J] to signify the default response.

```
EXIT returns you to Point

The IBM Personal Computer DOS
Version 3.nn
C:\M2TMP link ptdemo [Q J]
IBM Personal Computer Linker
Version 2.3 (C) Copyright IBM Corp. 1981, 1985
Run File [PTDEMO.EXE)
List File [NUL.MAP)
Libraries [.LIB]: \your_dir\m2lib\m2lib+s\m2lib\m2rts [Q J]
C:\M2TMP exit [Q J]
```

Enter the library names at the Libraries [.LIB]: prompt, as shown above:

\YOUR_DIR\M2LIB\M2LIB+S\M2LIB\M2RTS [Q J].

Step 3:  Return to the POINT Window.

At the final prompt, type EXIT [Q J], to return to the POINT window. Now you can run the PTDEMO program you just linked with the DOS Linker.
Task 7: M2ASSIST — Run .EXE File

Step 1: Run the .EXE file.

Press (0 0) on the M2ASSIST menu and drag the cursor down the menu to select Run. The screen is redrawn as follows:

```
C:\MZTMP\ptdemo
Hello Everybody
Hello Everybody
Hello Everybody
Hello Everybody
Hello Everybody
Hello Everybody
Hello Everybody
Hello Everybody
Hello Everybody
Hello Everybody

Press any key to continue
```

- press any key to continue
Step 2: Run the .EXE file from DOS.

Quit POINT and return to DOS, by pressing Alt-Q. Now, from the command line, type PTDEMO. The screen is redrawn as follows:

```
C: \M2TMP\PTDEMO
Hello Everybody
Hello Everybody
Hello Everybody
Hello Everybody
Hello Everybody
Hello Everybody
Hello Everybody
Hello Everybody
Hello Everybody
Hello Everybody
Hello Everybody
Hello Everybody
Hello Everybody
C: \M2TMP
```

---NOTE---
For information on other useful features of POINT, including multiple-window editing, and additional features of the M2ASSIST extension, consult the POINT Editor User's Manual.
Review: The POINT Entry/Run Cycle

You started by invoking the POINT Editor, with PTDEMO.MOD loaded as an argument. Then, from within POINT you were able to check syntax, compile, link, and run the program. Let's look at the tools you just used and the steps you just took:

```
PT.EXE
  Check Syntax
  Edit

PTDEMO.MOD

Compile
  with M2COMP
  or M2C

PTDEMO.OBJ

Link

PTDEMO.EXE

Run .EXE
  program
```

Now it's time for you to write some Modula-2 programs of your own. Refer to the bibliography in Appendix A which lists books, magazines, and user groups that are devoted to Modula-2. And remember the words of Leonardo DaVinci:

"The greatest tragedy in life is for theory to outstrip practice."

Good luck in your study of this great new programming language!
Since Modula-2 evolved from Pascal it's easy for Pascal programmers to become familiar with Modula-2.

There are two levels of difference between Modula-2 and Pascal. First, Modula-2 implements modern software engineering, such as, data abstraction, functional abstraction, concurrency and more frequent use of modular programs. All these features are not part of the standard Pascal definition, neither are they present in any implementation. The second level of difference consists of relatively minor changes in Modula-2 program syntax and constructs.

The most important difference is the introduction of the module.

2.1 Types of Modules

Modula-2 has three types of modules: program, definition, and implementation modules. Program modules contain the source code for your main program. Program libraries are created from matched pairs of definition modules and implementation modules. Source code for all modules types is stored as standard text files and may be modified by any text editor capable of working with these files. Program and implementation modules use the extension .MOD. Definition modules use the file extension .DEF.
2.1.1 Program Modules

A program module is the main module of your program. A program consists of all the modules that are referred to directly or indirectly by the main module. For program modules, the module code, which is declared following the last BEGIN, constitutes the main program. After initialization of all imported modules, the program will start there.

The examples in the following section are program modules. Program modules have the following form:

```modulename>:
  Import from the library modules to use, if any, in the form:
  FROM <modulename> IMPORT
    <list of identifiers separated by commas>;
  or:
  IMPORT <modulename>;
  Declaration of constants, types, variables and procedures.
BEGIN
  Code of the main program.
END <modulename>.
```

The list of identifiers imported may contain the names of constants, types, variables and procedures exported from a library module. These names must be separated by commas. Refer to Wirth's book *Programming in Modula-2* for a more detailed explanation of the module syntax.

2.1.2 Definition Modules

Definition modules define the interface between modules. By separating the definition of the interface between modules from the implementation of those modules, the implementations may be modified without having to recompile the entire system. As programmers involved with large systems know, recompiling the entire system can be a very time consuming process.

Definition modules have the following form:

```definition_module>:
  Import from the library modules to use, if any, in the form:
  FROM <modulename> IMPORT
    <list of names separated by commas>;
  or:
  IMPORT <modulename>;
  EXPORT QUALIFIED
    <list of names separated by commas>;
  Declaration of constants, types, variables and procedures.
  Procedure declarations consist of the procedure header only, including the parameter list.
END <modulename>.
```
2.1.3 Implementation Modules

Implementation modules contain the statements required to perform the functions defined in the definition modules. They are similar in format to program modules except their module body does not need to constitute a main program. Libraries are constructed from matching sets of definition and implementation modules.

Implementation modules have the following form:

```
IMPLEMENTATION MODULE <modulename>;
Import from the library modules to use, if any, in the form:
  FROM <modulename> IMPORT <list of names separated by commas>;
or:
  IMPORT <modulename>;
Declaration of constants, types, variables and procedures.
  Procedure declarations consist of the header and body,
  including the code of the procedure.
BEGIN
  Module initialization code.
END <modulename>.
```

The constants, types and variables declared in the corresponding definition module, must not be repeated in the implementation. These names are known implicitly. However, for every procedure specified in the definition part, a complete procedure, with matching name and parameter list, must be contained in the implementation part.

Modula-2 enhances software production by shortening development time. While the real speed of a Pascal or Modula-2 compiler is important, consider the time involved in software production cycle. Include time for software updates and alterations. Pascal often translates this into a significant additional programming effort, which offsets the benefit of short Pascal compile time. Modula-2 minimizes this wasteful "domino effect" by using highly independent module libraries.

The Modula-2 core is smaller than the Pascal core, because Modula-2 has no predefined I/O statements, math functions or string manipulation routines. Modula-2 imports them from library modules. So, Modula-2 really practices what it preaches!

This chapter has two main sections. The first shows a Pascal programmer how to write a Modula-2 program or to convert a Pascal program to Modula-2. The second explains the concept of the module.

The next section discusses differences in syntax and construct between Pascal and Modula-2. This should allow a programmer to convert Pascal programs or to write new ones in Modula-2.
2.2 First Steps From Pascal To Modula-2

To demonstrate some basic syntax differences between *Pascal* and *Modula-2*, consider the following simple number-squaring program:

```pascal
MODULE FirstDemo;

(* List of imported procedures *)
FROM InOut IMPORT WriteString, WriteInt, ReadInt, WriteLn;

VAR Number, Square : INTEGER; (* Programs' identifiers *)
BEGIN
(* ---------- Input -------*)
  WriteString("Type an integer "); ReadInt(Number);
(* ------ Processing ------*)
  Square := Number * Number;
(* ---------- Output ------*)
  WriteString("Number = "); WriteInt(Number,4); WriteLn;
  WriteString("It's square = "); WriteInt(Square,6);
  WriteLn;
END FirstDemo.
```

The following new concepts specific to *Modula-2* are in the previous example:

- *Modula-2* programs start with the reserved word `MODULE` followed by a program name. The same name appears after the very last `END` statement. The program name takes no arguments.
- All identifiers are case sensitive in *Modula-2*. Thus changing the case of one letter in an identifier's name is sufficient to create a new identifier name.
- *Modula-2* reserved words are always in upper case letters.
- Comments are enclosed in (* and *). *Modula-2* uses curly braces for sets, hence they do not enclose comments as in *Pascal*. *Modula-2* allows nested comments.
- All I/O procedures are imported from a library module, such as InOut in this case. Hence, *Pascal*’s multipurpose `WRITELN` is replaced with a series of output procedures. Each outputs only one item. If you look for the InOut module in *Programming in Modula-2*, by Niklaus Wirth, you will find that it exports many procedures. In our example we chose to "import" the four required procedures only.
2.3 More Differences

Labels and GOTO statements are no longer supported by Modula-2. To translate Pascal programs with labels or GOTO statements, you need to rewrite your Pascal program.

Constants are declared, similar to Pascal. Modula-2 allows for constant expressions to be used wherever a constant is expected. Integer constants now include hexadecimal and octal numbers. Thus: 12AFH represents a hexadecimal number, ending with an H. 27B is an octal number, ending with a B. In addition, 27C represents the same octal number, but its type is CHAR. Real constants are similar to those in Pascal. When expressed in scientific notation, only the uppercase E must be used. All real constants require a decimal point.

Character and string constants are similar to Pascal with an enhancement. Single or double quotes may be used to delimit them. Thus, "Hello" and 'Hello' are both acceptable, but 'Hello' is not. The choice of delimiter may be dictated by the presence of a quote symbol as part of the string constant. Thus "Don't" forces the use of double quotes, since a single one is part of the string.

Similarly, 'They have "some" children' must be delimited by single quotes.

Modula-2 defines these basic data types: integer, boolean, characters, real, cardinal and bitset. The first three types are used as they are in Pascal. Using reals has a restriction that prevents it from being mixed with integers and cardinals in an expression. Predefined type converter functions must be used. Cardinals are unsigned integers with values ranging from zero to an upper, machine-fixed limit. While cardinals and integers are assignment compatible, they too cannot be directly mixed in an expression. The bitset type is a predeclared set type for low level data manipulation.

Modula-2 supports sets with some syntax modifications. Set constants are enclosed in curly braces. Set types are defined with SET OF <enumerated or subrange types>:

- CONST OctalNumSet = {1,2,3,4,5,6,7,8};
- TYPE Binary = SET OF [0..1];

Modula-2 implements a generous number of set operations, including the symmetric set differences.
Chapter 2

Modula-2 supports enumerated types just like Pascal. Subranges are also similar, but Modula-2 requires them to be enclosed in square brackets. Why? This enables subrange types to be used in defining array limits, as in:

```
TYPE SmallRange  = [1..10];  
BigRange       = [1..100];  
SmallArray     = ARRAY SmallRange OF REAL;  
BigArray       = ARRAY BigRange OF REAL;  
HugeArray      = ARRAY SmallRange, BigRange OF REAL;  
Table          = ARRAY [1..10],[1..100] OF REAL;  
Matrix         = ARRAY [1..10,1..10] OF REAL;  (* WRONG! *)
```

The above list also shows the difference between the two languages in declaring arrays. Modula-2 allows subrange types to be used. Moreover, multidimensional arrays must have the range of each dimension separated by a comma. However, using multidimensional arrays in a program follows the familiar Pascal notation.

Modula-2 regards character strings as merely an array of characters. In normal practices the ASCII null (zero code) is used as a string terminator. However, it is possible to use slightly more elaborate record structures to implement strings. String manipulation depends on library modules. The standard string library offers the essential operations and treats strings as an array of characters.

Fixed records are no different in Modula-2 than in Pascal. Variant records have been extended to allow for more than one variant field. In addition, the latter may have an ELSE clause option. Consider the following example:

```
TYPE Material = (Element, Compound);  
State        = (Liquid, Gas, Solid);  
ChemicalPointer = POINTER TO Chemical;  

Chemical = RECORD  
   Name        : ARRAY [1..40] OF CHAR;  
   Formula     : ARRAY [1..20] OF CHAR;  
CASE MaterialType : Material OF  
   Element     : AtomicNumber : CARDINAL;  
   Valence     : INTEGER;  
   AtomicWeight: REAL |  
   Compound    : MolecularWeight : REAL;  
   CationCharge,  
   AnionCharge  : INTEGER;  
   CationName,  
   AnionName    : ARRAY [1..15] OF CHAR;  
END;  

CASE NormalPhysicalState:State OF  
   Liquid : LiquidDensity,BoilingPoint : REAL |  
   Gas    : VaporPressure,VaporTemp : REAL |  
   Solid  : SpecificGravity, MeltingPoint : REAL;  
END;
```
Dynamic records can be created and accessed using pointers. Their declaration is a bit more verbose, using the keywords POINTER TO, as in:

```
MODULE PointerDemo;
(* Partial listing *)
TYPE CardPtr = POINTER TO CARDINAL;
   ComplexPtr = POINTER TO Complex;
   Complex   = RECORD
        Re, Imag : REAL;
   END;

VAR CPtr : ComplexPtr;
   (* Other variables declared here *)
BEGIN
   New(CPtr); (* Create new dynamic record *)
   (* Initialize to square root of minus one *)
   CPtr^.Re := 0.0; CPtr^.Imag := 1.0;
   (* rest of the program *)
END PointerDemo.
```

As shown above, pointers are used with the carat symbol. The WITH keyword is also used in Modula-2 to reference a record’s field name without the record-name-dot notation. A mandatory END is required. Modula-2 allows one record identifier per WITH statement. There is no need for the BEGIN keyword after the DO reserved word.
Chapter 2

Creating dynamic variables with variant record structures uses the predefined NEW procedure and includes variant tags as additional arguments. Recalling the variant record Chemical, and its related pointer type ChemicalPointer, we proceed to define the following pointer-typed variables:

FROM Storage IMPORT ALLOCATE
VAR Iron, Water, Oxygen : ChemicalPointer;

and create dynamic variables using the above pointers:

(* Iron is an element and a solid at normal conditions *)
NEW(Iron,Solid);

(* Water is a compound and a liquid at normal conditions *)
NEW(Water,Liquid);

(* Oxygen is an element and a gas at normal conditions *)
NEW(Oxygen,Gas);

Since I/O operations are no longer in the Modula-2 core, the predefined Pascal FILE OF <type> has no equivalent. Instead, File structures depend on the I/O library module.

Modula-2 variable declaration is identical to that in Pascal with one additional feature. An absolute address in square brackets, may follow the variable name. For example:

VAR Screen[0B800H:0H] : ARRAY [1..MAXCOL],[1..MAXROW] OF CHAR;

Modula-2 has simplified the syntax of statement blocks. In Pascal loops or WITH statements, if the DO reserved word was followed by BEGIN, there is a compound statement. Modula-2 has dropped the BEGIN keyword after DO and replaced it with a mandatory END to close the loop or WITH body.
The IF statement is very similar to that in *Pascal* with the following changes:

- No need for BEGIN-END in THEN or ELSE clauses with more than one statement.
- The IF construct must close with the END statement.
- The *Pascal* ELSEIF is now ELSIF, one letter shorter.

The above changes in the following function calculate your checking account balance. For less than $500, a local bank charges you with a $5.00 service charge. For under $1500, you get 5.4% interest rate. Beyond that amount, you get a 9.4% rate.

```modula
PROCEDURE BankOnIt(Savings : REAL) : REAL;
VAR
    BankCharges, Interest : REAL;
    BEGIN
        BankCharges := 0.0;
        IF Savings < 500. THEN (* Apply a five dollar service charge *)
            BankCharges := 5.00;
            Interest := 5.4 (* percent *)
        ELSIF Savings < 1500. THEN (* same rate as above, but no charges *)
            Interest := 5.4
        ELSE (* Very nice account *)
            Interest := 9.4;
        END; (* IF statement *)
        RETURN (Savings * (1. + Interest/100.) - BankCharges);
    END BankOnIt;
```

The CASE statement has also been enhanced in *Modula-2*. A much needed catch-all ELSE clause is recognized. Statement sequences for each case are simply separated by vertical bars. The BEGIN-END keywords are no longer needed for compound statements. Here, the CASE statement translates numeric school grades into letters:

```modula
CASE NumericGrade OF
    90..100 : Grade := 'A';
    Message := 'Very Nice work champ!';
    80..89 : Grade := 'B';
    Message := 'Nice work';
    70..79 : Grade := 'C';
    Message := 'OK, but you can do better'
    ELSE
    Grade := 'F';
    Message := 'Sorry, you failed';
END; (* CASE NumericGrade *)
```
Modula-2 has improved on loops where needed. The REPEAT-UNTIL loop is identical to its implementation in Pascal. The WHILE-DO loop now requires a mandatory END statement to bracket the loop. This is regardless of the number of statements inside the loop. No BEGIN keyword is required after the DO. The FOR-DO loop has undergone even more changes. Like the above WHILE loop, it must have an END statement. Modula-2 allows the loop counter to increment/decrement by more than one, using the BY clause. These "steps" can be positive or negative. Appropriately, Modula-2 no longer supports the Pascal DOWNTO keyword. Here is a short program demonstrating the FOR-DO loop in its new construct.

```
MODULE AreaUnderCurve;
   (* Program to calculate area under curve Y = X^2 between *)
   (* zero and one. Simpson's rule is used. *)
FROM InOut IMPORT WriteString, WriteLn;
FROM RealInOut IMPORT WriteReal;
VAR Y : ARRAY [1..11] OF REAL;
    Increment, Area, SumEven, SumOdd : REAL;
    i : CARDINAL
BEGIN
    Increment := 0.1;
    i := 1
   WHILE i <= 11 DO (* Initialize Y array *)
      Y[i] := (Increment * FLOAT(i)) * (Increment * FLOAT(i))
         INC(i) (* Increment i by one *);
   END:
   (* Initialize summations *)
   SumEven := 0.0; SumOdd := 0.0;
   (* Start loop for summing even terms *)
   FOR i := 2 TO 10 BY 2 DO
      SumEven := SumEven + Y[i];
   END;
   (* Start loop for summing odd terms, counting back *)
   FOR i := 9 TO 1 BY -2 DO
      SumOdd := SumOdd + Y[i];
   END;
   Area := Increment / 3.0 * (Y[1] + 4.0*SumEven + 2.0*SumOdd + Y[11]);
   WriteString("Area for X^2 between 0 and 1 = ");
   WriteReal(Area,14); WriteLn;
END AreaUnderCurve.
```
Modula-2 introduces a new loop construct, the open loop. The keywords LOOP and END define the open loop body. To exit such a loop an EXIT statement is used. The open loop is very flexible. An if statement with EXIT placed right after the LOOP keyword gives the effect of a WHILE loop, as in:

```pascal
i := 0; j := 0;
WHILE i < 10 DO
  INC(i);
  INC(j, i);
END;
i := 0; j := 0;
LOOP
  IF i >= 10 THEN EXIT; END;
  INC(i);
  INC(j, i) (* j := j + i *);
END;
```

Similarly, placing the loop exit test just before the end of the loop simulates the REPEAT-UNTIL loop.

```pascal
i := 0; j := 0;
REPEAT
  INC(i);
  INC(j, i);
UNTIL i = 10;
i := 0; j := 0;
LOOP
  INC(i);
  INC(j, i);
  IF i = 10 THEN EXIT; END;
END;
```
Chapter 2

The loop exit test can be anywhere inside the loop body, as shown in the following example. The program calculates the Bessel function of the first kind.

```pascal
MODULE BesselFunction;

FROM InOut IMPORT WriteString, WriteLn, ReadCard;
FROM RealInOut IMPORT WriteReal, ReadReal;

CONST Epsilon = 1.0E-08;

VAR Sum, Term, X, Y,
    Factor1, Factor2, Factor3, PowerTerm : REAL;
    Order, i : CARDINAL;

BEGIN
    WriteString("Enter order of Bessel function ");
    ReadCard(Order); WriteLn;
    WriteString("Enter argument ");
    ReadReal(X); WriteLn;
    Sum := 0.0; i := 0;
    Y := -0.25 * X * X;
    Factor1 := 1.0;
    Factor2 := 1.0;
    Factor3 := 1.0; PowerTerm := 1.0;
    IF Order > 0 THEN
        FOR i := 1 TO Order DO
            Factor3 := Factor3 * FLOAT(i);
            PowerTerm := PowerTerm * X / 2.0;
        END;
    END;
    i := 0; (* Initialize counter *)
    LOOP
        Term := Factor1 / Factor2 / Factor3;
        Sum := Sum + Term;
        (* Is added term insignificant ? *)
        IF ABS(Term) < Epsilon THEN EXIT; END;
        INC(i);
        Factor1 := Factor1 * Y;
        Factor2 := Factor2 * FLOAT(i);
        Factor3 := Factor3 * FLOAT(i);
    END;
    (* Program flow resumes here after EXIT *)
    Sum := Sum * PowerTerm; (* Last calculation *)
    WriteString("Bessel Function = ");
    WriteReal(Sum,14); WriteLn; WriteLn;
END BesselFunction.
```

Each EXIT statement resumes program flow after the exited loop body. Therefore, to exit nested open loops one needs as many exit statements as there are loops.
2.4 Functions and Procedures

Modula-2 considers a function as merely a procedure returning a value. Thus, the keyword FUNCTION has been dropped and replaced with PROCEDURE. The other change implements a RETURN statement that exits the function and returns the sought value. If there are any statements after the RETURN, they will not be executed. LOGITECH Modula-2, Version 3.0, functions only return basic types and pointers.

Consider the following examples of a function to calculate the square root of a real number, using Newton’s iterative method.

```cpp
PROCEDURE SquareRoot(X : REAL) : REAL;

CONST Epsilon = 1.0E-08; (* Tolerance factor *)

VAR Y : REAL; (* Local storage for square root *)

BEGIN
  Y := X / 2.; (* Initial guess for square root *)
  REPEAT (* Improve guess by Newton's iterations *)
    Y = (Y + X / Y) / 2.;
  UNTIL ABS(Y*Y - X) < Epsilon;
  RETURN Y (* back to caller *);
END SquareRoot;
```

An additional difference between the two languages, is that Modula-2 requires all procedures and functions to include their names after the last END in the subprogram.

Like Pascal, Modula-2 allows parameter passing by value or by reference (using VAR declaration). The latter makes it possible to simulate a function that returns structured data types. Modula-2 has implemented an important new feature in parameter passing -- open arrays. This enables procedures (and functions) to tackle arrays of consistent type, but varying in size. This makes it easier to write general purpose routines in Modula-2. Open arrays are limited to one dimensional arrays. They are declared in an argument list as ARRAY OF <type>, with no dimension limits. Inside the procedure body the dimension bounds are mapped onto [0..<array size - 1>]. Modula-2 provides the predefined HIGH() function to return the upper bound value for an open array. Thus an open array is mapped onto [0..HIGH(<Open array name>)].
Here is an example for a routine to calculate the mean value of an array of reals.

```pascal
PROCEDURE Mean(X : ARRAY OF REAL) : REAL;
VAR i : CARDINAL;
    Sum : REAL;
BEGIN
    Sum := 0.; (* Initialize sum *)
    FOR i := 0 TO HIGH(X) DO
        Sum := Sum + X[i];
    END;
    RETURN Sum / FLOAT(HIGH(X) + 1);
END Mean;
```

Function `Mean` is able to handle arrays of varying sizes. The number of elements in the passed array `X` is `(HIGH(X) + 1)`. This assumes that the entire array `X` is filled with data.

Procedural and functional types are supported in Modula-2. The following example demonstrates the first type. The program below reads an array of cardinals from a file and sorts them. The sorting routines are imported. The program examines the list size and depending on its value employs the appropriate sorting method. For small arrays, the bubble sort is used. For medium arrays the Shell sort is called upon. QuickSort is reserved for large arrays. To demonstrate the procedural type, the program defines `SortProc`. It is a procedure taking two arguments: an array of cardinal and a scalar type cardinal. The variable `SortMethod` is of `SortProc` type. In the `IF` statement we assign either imported sorting procedure to `SortMethod`. Notice that the assignment does not involve any procedural arguments. Following the `IF` statement is a call to `SortMethod` with a complete argument list. The call will execute the assigned procedure (BubbleSort, ShellSort or QuickSort).
Here is the program:

```modula-2
MODULE Sort;
FROM InOut IMPORT WriteString, ReadString, WriteCard, WriteLn;
(* Modules FileIO and CardinalSortLib are fictitious *)
FROM MyFileIO IMPORT TextFile, EOF, Assign, Reset, ReadCardinal, Close;
FROM CardinalSortLib IMPORT ShellSort, QuickSort, BubbleSort;

(* Define a procedure type with an array of cardinals *)
(* and a scalar cardinal as arguments. *)
(* Imported sorting procedures must have same arguments *)

TYPE SortProc = PROCEDURE (VAR ARRAY OF CARDINAL; CARDINAL);

VAR CardinalList : ARRAY [1..5000] OF CARDINAL;
Num, i : CARDINAL;
Filename : ARRAY [1..14] OF CHAR;
F : TextFile
SortMethod : SortProc;

BEGIN
WriteString("Enter data filename ");
ReadString(Filename)
Assign(TextFile,Filename);
Reset(F)
Num := 0;
WHILE NOT EOF(F) DO
INC (Num);
ReadCardinal(F,CardinalList[Num]);
END;
Close(F);

IF Num <= 30 THEN (* Small list, use bubble sort *)
SortMethod := BubbleSort (* No arguments *)
ELSIF Num <= 150 THEN (* Medium list => Shell sort *)
SortMethod := ShellSort (* No arguments *)
ELSE (* QuickSort used for large array *)
SortMethod := QuickSort (* No arguments *);
END;

SortMethod(CardinalList, Num); (* Sort list *)

FOR i := 1 TO Num DO (* Display sorted list *)
  WriteCard (i,5);
  WriteString (', ');
  WriteCard (CardinalList[i],6);
  WriteLn;
END;

END Sort.
```

*Modula-2* provides PROC, a predefined parameterless procedure type. This is useful in creating coroutines, which are discussed later.

Predefined functions and procedures are listed in Wirth’s *Programming in Modula-2.*
Chapter 2

2.5 Use of Modules

2.5.1 User Definable Modules

*Modula-2* implements library modules to benefit software productivity. This affects both the individual programmer and a team of programmers working on a big project. One of the advantages of library modules is that they minimize side effects between modules written by different programmers or written at different times. This reduces debugging greatly and significantly improves on software maintainability.

The virtue of modules stems from the fact that inter-module communication is spelled out and is not ambiguous. This is done by specifying the objects exported and imported. As we have seen in previous *Modula-2* programs, there are invariably lists of imports. Each specifies the module from which to import and the specific procedures imported. If the reader looks at the definition module of, for example, module InOut, he will find all the imported items defined in that module (i.e. marked for export). *Modula-2* works on the principle that you can obtain an item only if it is made available to you.

In *Modula-2* a library module is made up of two parts: the definition and the implementation modules. The definition module is regarded as the interface with client modules. All exported objects are catalogued there. This includes constants, data types, variables and procedures. The implementation module has all the detailed exported procedure code and additional local constants, data types, variables and procedures. Optional module initialization code lines may be included. Each of the definition and implementation modules are compiled separately. The programmer may "improve" on the implementation module by replacing old algorithms with more efficient ones. As long as the exported objects are not altered, we only need to recompile the implementation module.
Consider the following example to demonstrate some of the above points. We present a small library module to create and add complex numbers. The definition module is:

```
DEFINITION MODULE ComplexOps;

EXPORT QUALIFIED
  Complex, (* Type *)
  MakeComplex, AddComplex; (* Procedures *)

  TYPE Complex = RECORD Real, Imaginary : REAL; END;
  (* Only procedure headings are needed *)

  PROCEDURE MakeComplex(X, Y : REAL; (* Input *)
                       VAR C : Complex (* Output *))
    (* Procedure to create a complex number from X & Y components. *)

  PROCEDURE AddComplex (A, B : Complex; (* Input *)
                        VAR C : Complex (* Output *))
    (* Procedure to add complex numbers A & B to give C *)

END ComplexOps.
```

The definition module ComplexOps exports the "transparent" type Complex. The terminology refers to types whose definition is made available to client modules. *Modula-2* also allows the export of "opaque" types, where the data type definition is not revealed. We will discuss this in more detail later. For now, it is enough to say that there is the following difference between transparent and opaque types: the ability of the client modules to have their own procedures (possibly available for export) to manipulate the transparent types only. This privilege is denied with opaque types. Thus clients modules of ComplexOps can develop and export procedure to subtract, divide and multiply complex numbers. Their access to the components of type Complex makes it possible.

In general, definition modules need only the heading of the exported procedures, and the definition module ComplexOps is no exception. In practice, the definition module should be the first one written to set the module specification. In large software projects this is the appropriate thing to do.
Chapter 2

The implementation module is:

```pascal
IMPLEMENTATION MODULE ComplexOps;

(* Type Complex has been defined in the definition module *)

PROCEDURE MakeComplex ( X, Y : REAL; (* Input *)
    VAR C : Complex (* Output *) )

(* Procedure to create a complex number from X & Y. *)
(* components. *)
BEGIN
    C.Real := X;
    C.Imaginary := Y;
END MakeComplex;

PROCEDURE AddComplex( A, B : Complex; (* Input *)
    VAR C : Complex (* Output *) );

(* Procedure to add complex numbers A & B to give C. *)
BEGIN
    C.Real := A.Real + B.Real;
    C.Imaginary := A.Imaginary + B.Imaginary;
END AddComplex;

END ComplexOps.
```

The implementation module does not contain the definition of type Complex. Since it is exported, the compiler is already aware of it through the definition module. The exported procedures are listed in the module. All the imported objects needed in each module need to be explicitly imported regardless, if it is a definition module, an implementation module, or a program module. Otherwise, import lists are located in the implementation module. Local constants, data types, variables and procedures are of course included in the implementation module.

We spoke earlier of the ability to change and improve the code in the implementation module. In certain cases this may require that exported transparent data types be modified. This poses a problem since transparent types give library module developers little or no control over how client modules use them. Most likely the sought improvement may be hindered because of potential data type incompatibility between the old and new structures. A new sister module is created. However this solution is not always a sound way to go.
While the above discussion refers to a rather specific case, it also points to a broader programming aspect: full control over exported data types. *Modula-2* has met this need by allowing opaque exported types. In this case the definition module lists the name of the opaque type only. No type structure is defined there. Instead, it is located in the implementation module. With the details about the structure denied to client modules, the exporting module has the monopoly on procedures that manipulate opaque types. Thus, with full control over opaque types comes the responsibility to export every procedure needed to process the data types in question. Care in planning ahead must be exercised.

Return to the complex number addition module. Complex numbers may be represented by two dimensional rectangular coordinates \((X,Y)\). Alternatively, the same \((X,Y)\) point can be replaced by polar coordinates: a modulus and an angle. While the two systems are equivalent, their components represent different physical entities. It is possible to develop a library of complex operations using rectangular coordinates and later change the implementation module to use polar ones. An opaque complex type makes the smooth transition.

Below is the new definition module for `ComplexOps`. Notice that the exported type `Complex` has no structure definition associated with it.

```modula-2
definition module ComplexOps;
  from Storage import allocate;

  export qualified
    Complex, (* Type *)
    MakeComplex, AddComplex; (* Procedures *)

  type Complex; (* Is now opaque *)
  (* Only procedure headings are needed *)

  procedure MakeComplex( X, Y : REAL; (* Input *)
    var C : Complex (* Output *));
  (* Procedure to create a complex number from \(X & Y\) components. *)

  procedure AddComplex( A, B : Complex; (* Input *)
    var C : Complex (* Output *));
  (* Procedure to add complex numbers \(A & B\) to give \(C\). *)

end ComplexOps.
```
Chapter 2

The implementation module is similar to the previous version. Within it the Complex type is now fully defined. The fields of the type Complex have been renamed to remind the reader that rectangular coordinates are used to represent complex numbers. Notice that opaque types must be pointer to other structures. This is mandatory in Modula-2.

IMPLEMENTATION MODULE ComplexOps;
FROM Storage IMPORT ALLOCATE;

(* Type Complex uses rectangular coordinates *)
TYPE Complex = POINTER TO RECORD
   XCoord, YCoord : REAL;
END;

PROCEDURE MakeComplex( X, Y : REAL; (* Input *)
   VAR C : Complex (* Output *))
(* Procedure to create a complex number from X & Y components.*)
BEGIN
   NEW(C);
   C.XCoord := X;
   C.YCoord := Y;
END MakeComplex;

PROCEDURE AddComplex (A, B : Complex; (* Input *)
   VAR C : Complex (* Output *))
(* Procedure to add complex numbers A & B to give C *)
BEGIN
   C.XCoord := A.XCoord + B.XCoord;
   C.YCoord := A.YCoord + B.YCoord;
END AddComplex;

END ComplexOps.
Here is the implementation module version that uses polar coordinates:

IMPLEMENTATION MODULE ComplexOps;
FROM Storage IMPORT ALLOCATE;
FROM MathLibO IMPORT sqrt, arctan, sin, cos;
(* Type Complex uses polar coordinates *)
TYPE Complex = POINTER TO RECORD
  Modulus, Angle : REAL;
END;
PROCEDURE MakeComplex ( X, Y : REAL; (* Input *)
  VAR C : Complex (* Output *));
(* Procedure to create a complex number from X & Y components. *)
BEGIN
  NEW(C);
  C^.Modulus := sqrt(X * X + Y * Y);
  C^.Angle := arctan(Y / X);
END MakeComplex;
PROCEDURE AddComplex ( A, B : Complex; (* Input *)
  VAR C : Complex (* Output *));
(* Procedure to add complex numbers A & B to give C *)
VAR X, Y : REAL;
BEGIN
  X := A^.Modulus * cos(A^.Angle) + B^.Modulus * cos(B^.Angle);
  Y := A^.Modulus * sin(A^.Angle) + B^.Modulus * sin(B^.Angle);
  MakeComplex(X, Y, C);
END AddComplex;
END ComplexOps.

The following changes took place:

- Four required mathematical functions are imported from MathLibO.
- The Complex type is defined using the Modulus and Angle fields.
- The body of the two module procedures has been significantly changed.
- Procedure AddComplex now calls procedure MakeComplex.

Note: the MakeComplex procedure in the very first implementation seemed an extravagant export: given the definition of Complex, client programs can assign values to the record fields effortlessly. The situation is reversed with the opaque Complex: client modules now really need procedure MakeComplex, since they have no idea about its internal structure. The rectangular and polar versions demonstrate this point.
2.5.2 Importing Procedures with Identical Names

With the incentive to develop library modules it is inevitable that the same procedure names appear in more than one module. How do we resolve the conflict due to importing two identically named routines? It is possible to omit the import list, thus importing the entire library. To use the imported routine we use the same notation as with referencing fields of record structures. With the module name constantly referenced, the compiler is able to distinguish which procedure we are calling. Moreover, the program readability will enjoy the clarity too. Consider the following example.

```pascal
MODULE ImportAllDemo

IMPORT InOut;
IMPORT MyFileIO;

VAR Filename : ARRAY [1..14] OF CHAR;
   F      : MyFileIO.TextFile; (* Imported type *)
   Message : ARRAY [1..80] OF CHAR;
   NumLines : CARDINAL;

BEGIN;
   InOut.WriteString("Enter file name ");
   InOut.ReadString(Filename);
   InOut.WriteLine;
   REPEAT
      InOut.WriteString("Enter number of lines ");
      InOut.ReadCard(NumLines);
      InOut.WriteLine;
   UNTIL NumLines > 0;

   MyFileIO.Assign(F,Filename);
   MyFileIO.Reset(F);
   InOut.WriteString("Enter text ");
   InOut.WriteLine;
   REPEAT
      InOut.ReadString(Message);
      MyFileIO.WriteString(F,Message);
      DEC(NumLines);
   UNTIL NumLines = 0;
   MyFileIO.Close(F);

END ImportAllDemo.
```

In the above example we can distinguish for each call to procedure WriteString whether it is imported from modules InOut or MyFileIO.
2.5.3 Standard Library Modules

*L*OGITECH *M*oduла-2 comes with a variety of versatile library modules. They supply your programs with a wide gamut of capabilities. This includes string manipulation, file I/O, disk directory access, data conversions, mathematical functions, *D*OS and low level access, coroutines, just to name a few. Many of the above routines are part of *P*ascal, but not *M*oduла-2. Thus *M*oduла-2 depends heavily on a core or fundamental library modules. The reader is referred to other parts of the manual where the definition modules are discussed.

There are three small but very important modules -- SYSTEM, Storage and Processes. We will discuss these modules because they export low level and process management routines.

The module Storage tackles the allocation and deallocation of dynamic variables. ALLOCATE and DEALLOCATE procedures are defined as:

```
PROCEDURE ALLOCATE (VAR a : ADDRESS; size : CARDINAL)
PROCEDURE DEALLOCATE (VAR a : ADDRESS; size : CARDINAL)
```

where ADDRESS is a pointer to a memory location imported from module SYSTEM. These are equivalent to calling the NEW and DISPOSE procedures used for the same purpose. The following demonstrates how the two sets of procedures work identically. Let us define the following data types and variable.

```
TYPE Ptr    = POINTER TO Element;
    Element = RECORD
        Volume, Weight : REAL;
        Name : ARRAY [1..80] OF CHAR;
    END;

VAR Indicator : Ptr;
```

Calling NEW(Indicator) and ALLOCATE(Indicator, TSIZE(Element)) yield the same result: creating a dynamic variable accessed through the pointer Indicator. TSIZE() is a function imported from module SYSTEM that returns the size of any data type. Similarly, DISPOSE(Indicator) and DEALLOCATE(Indicator, TSIZE(Element)) both undo the effect of the above procedures.
Let us demonstrate the use of the ALLOCATE and DEALLOCATE procedures in developing a short dynamic string library module.

DEFINITION MODULE DynamicString;

EXPORT QUALIFIED STRING, NewString, RemoveString, AssignString, Length;

TYPE STRING; (* Opaque type *)

PROCEDURE NewString ( VAR S : STRING; (* Output *)
  MaxLength : CARDINAL (* Input *));
(* Create a dynamic string *)

PROCEDURE RemoveString ( VAR S : STRING; (* Input *));
(* Remove a dynamic string *)

PROCEDURE AssignString ( VAR S : STRING; (* Output *)
  A : ARRAY OF CHAR (* Input *)
(* assign an array of characters to a STRING *)

PROCEDURE Length ( S : STRING) : CARDINAL;
(* Function to return string length *);

END DynamicString.

The implementation module is:

IMPLEMENTATION MODULE DynamicString

FROM SYSTEM IMPORT ADDRESS, TSIZE;
FROM Storage IMPORT ALLOCATE, DEALLOCATE;

TYPE STRING = POINTER TO RECORD
  Long,
  MaxLong : CARDINAL;
  Element : ADDRESS;
END;

PROCEDURE NewString ( VAR S : STRING; (* Output *)
  MaxLength : CARDINAL (* Input *));
(* Create a dynamic string *)

BEGIN
  NEW(S);
  WITH S DO
    Long := 0;
    MaxLong := MaxLength;
    ALLOCATE ( Element, + MaxLong);
  END;
END NewString;
PROCEDURE RemoveString(VAR S : STRING; (* Input *));
(* Remove a dynamic string *)
BEGIN
  WITH S DO
    DEALLOCATE(Element, + MaxLong);
  END;
  DISPOSE(S);
END RemoveString;

PROCEDURE AssignString ( VAR S : STRING; (* Output *)
                         A : ARRAY OF CHAR (* Input *)
) (* assign an array of characters to a STRING *)
VAR Ptr : POINTER TO CHAR;
BEGIN
  IF A[0] <> OC
  THEN i := 0;
      WHILE (i <= HIGH(A) AND A[i] <> OC)
         AND (S^.MaxLong>= (i*TSIZE(CHAR)))
      DO
         Ptr := S^.Element + i * TSIZE(CHAR);
         A[i] := Pt^;
         INC(i);
      END;
      S^.Long := i + 1;
  ELSE
    S^.Long := 0 (* Empty string *);
  END;
END AssignString;

PROCEDURE Length(S : STRING) : CARDINAL;
(* Function to return string length *)
BEGIN
  RETURN S^.Long;
END Length;

END DynamicString.
Chapter 2

The next module we examine is SYSTEM. It exports four data types: BYTE, WORD, ADDRESS and PROCESS. The type WORD corresponds to one hardware storage unit. For example, the types CARDINAL and INTEGER use one WORD of storage. The type ADDRESS is defined as POINTER TO WORD. The type PROCESS is used in declaring coroutines.

The type WORD opens the door for some data conversion and the creation of general purpose (generic) routines. Recall that Modula-2 routines accept open arrays of any type in their argument lists. The ARRAY OF WORD is no exception and is compatible with any type, scalar or otherwise.

In the first example of using WORD we demonstrate the compatibility between CARDINAL and INTEGER. Each type occupies one WORD of storage, a key feature in the example. The following procedure searches an array of either types. The index of the matched element is returned. A boolean flag is used to indicate whether the returned value reflects a successful search. We assume that the array values are in the range [0..32767], the common value range for integers and cardinals.

```pascal
PROCEDURE SearchArray ( A : ARRAY OF WORD; (* Input *)
S : WORD (* Input *)
VAR Found : BOOLEAN (* Output *) ) : CARDINAL;

VAR i, SoughtValue : CARDINAL;

BEGIN
  Found := FALSE; (* Default outcome *)
i := 0; (* Zero search index *)
  SoughtValue := CARDINAL(S);
  WHILE (i <= HIGH(A)) AND (NOT Found) DO
    IF CARDINAL(A[i]) = SoughtValue THEN (* Found it! *)
      Found := TRUE
    ELSE (* Next element? *)
      INC(i);
    END;
  END; (* WHILE *)
  RETURN i;
END SearchArray;
```

Using ARRAY OF WORD to create generic modules is more elaborate when handling multi-word data structures. Since the processed object size varies, we must supply a single "sample" type in the generic procedure argument list. The above sample type is used as a template to determine the type size and provide local scalar variables. Another set of needed parameters is the user-supplied operations, such as comparisons, performed on the data objects. This is supplied in the form of procedural or functional parameters.
Here is a simple procedure to perform a generic linear list search on an array:

```pascal
PROCEDURE GenericLookUp ( VAR SearchArray : ARRAY OF WORD;
                           SampleScalarType, SearchValue : ARRAY OF WORD;
                           IsItEqual : SuppliedPROC;
                           VAR Found : BOOLEAN;
                           VAR Num, TypeSize, SearchIndex : CARDINAL;

PROCEDURE GetElement ( Index : CARDINAL;
                       VAR Object : ARRAY OF WORD);
     (* Procedure to extract one object *)
     VAR i : CARDINAL;

BEGIN
   FOR i := 0 TO TypeSize-1 DO
      Object[i] := SearchArray[(Index*TypeSize + i);
   END;
END GetElement;

BEGIN
   TypeSize := HIGH(SampleScalarType) + 1;
   Num := (HIGH(SearchArray) + 1) DIV TypeSize;
   SearchIndex := 0;
   Found := FALSE;
   WHILE (SearchIndex < Num) AND (NOT Found)
      DO GetElement(SearchIndex, SampleScalarType);
      IF IsItEqual(SampleScalarType, SearchValue)
         THEN Found := TRUE
      ELSE
         INC(SearchIndex);
      END;
   END;
END GenericLookUp;
```

In the above example we supply an array of objects via SearchArray. The SearchValue and SampleScalarType variables supply the searched value and an additional internally needed copy of the single object. The local procedure GetElement is used to extract a member of the search array and save it into SampleScalarType. The user-supplied function IsItEqual is used in comparing the search value with an array element.
Chapter 2

Below is a sample for the IsItEqual function dealing with date records. Local pointers are used to access the record structure in question. Once the pointer addresses are assigned, the RETURN statement supplies the logical result for the two-field test.

PROCEDURE IsItEqual(Element1, Element2 : ARRAY OF WORD) : BOOLEAN;

VAR Ptrl, Ptr2 : POINTER TO RECORD
    DayNumber, MonthNumber : CARDINAL;
END;

(* Get pointers addresses *)
Ptrl := ADR(Element1);
Ptr2 := ADR(Element2);
RETURN ((Ptrl^.DayNumber = Ptr2^.DayNumber) AND
            (Ptrl^.MonthNumber = Ptr2^.MonthNumber));
END IsItEqual;

Module SYSTEM has three address related functions. ADR(Z) returns the address of identifier Z. Functions SIZE and TSIZE return the sizes of a variable and data type, respectively. The rest of the exported routines tackle concurrency.
Consider the following simple example. It continuously displays the messages "In Coroutine <n>", where <n> follows the sequence [1,2,3].

MODULE ConcurrentDemo;

FROM InOut IMPORT WriteString, WriteLn;
FROM SYSTEM IMPORT WORD, PROCESS, ADR, SIZE, NEWPROCESS, TRANSFER;

VAR main, Coroutinel, Coroutine2, Coroutine3 : PROCESS;
    WorkSpace1, WorkSpace2, WorkSpace3 : ARRAY [1..200] OF WORD;
(* Workspace *)

PROCEDURE Message1;
BEGIN
    LOOP
        WriteString("In Coroutine # 1");
        WriteLn;
        TRANSFER(Coroutinel, Coroutine2);
    END;
END Message1;

PROCEDURE Message2;
BEGIN
    LOOP
        WriteString("In Coroutine # 2");
        WriteLn;
        TRANSFER(Coroutine2, Coroutine3);
    END;
END Message2;

PROCEDURE Message3;
BEGIN
    LOOP
        WriteString("In Coroutine # 3");
        WriteLn;
        TRANSFER(Coroutine3, Coroutinel);
    END;
END Message3;

BEGIN (* main *)
    (* Create the new Coroutines *)
    NEWPROCESS(Message1, ADR(WorkSpace1), SIZE(WorkSpace1), Coroutinel);
    NEWPROCESS(Message2, ADR(WorkSpace2), SIZE(WorkSpace2), Coroutine2);
    NEWPROCESS(Message3, ADR(WorkSpace3), SIZE(WorkSpace3), Coroutine3);
    TRANSFER(main, Coroutinel);
END ConcurrentDemo.
Chapter 2

The previous program shows that each co-routine is created using the NEWPROCESS procedure taking the following arguments:

- Parameterless procedure name. The procedure must be at the top level in the module and not nested within another routine.
- The address of a coroutine workspace (for stacks and other items).
- Size of the workspace.
- A PROCESS typed variable to be associated with the coroutine name.

Our example uses three variables, each 200 WORDS long, to reserve the needed workspaces. An alternate route is to dynamically allocate the workspace sizes.

The coroutine example also shows how coroutines are activated. The TRANSFER procedure is used to request the activation of a new coroutine while suspending the old one. When ConcurrentDemo first runs, the three coroutines are created. Next, the main section transfers the attention of the CPU to the first coroutine and suspends itself. An infinite sequence of tasks begins. Each coroutine displays a message and transfer CPU control to another coroutine, and so on. Coroutine procedures are parameterless, as stated earlier, and contain their code inside an infinite open loop.

IOTRANSFER is another procedure exported by SYSTEM and works similar to TRANSFER. It is oriented towards tackling device interrupts. Since these interrupts take place beyond the program’s control, there must be an automatic way to handle them. IOTRANSFER shifts the control from a first process to a second one, but is able to resume the first when an interrupt occurs. IOTRANSFER takes a third parameter, the interrupt vector number. Module SYSTEM also exports procedure LISTEN. This causes the coroutine to wait for an IOTRANSFER to take place.

Synchronization between coroutines is vital in keeping their liveliness. This assures that every process maintains its vitality. Using a single CPU, each process must run for a short period of time and then be suspended to allow others to resume likewise. The above coordination becomes more critical when the same data is accessed by more than one coroutine. In this case it is imperative to ensure that only one process manipulate data. This means that other coroutines must wait for their turn to access the same data. The overall picture depicts two waiting queues: one for processes simply waiting their turn to use the CPU, the other for processes waiting to access a critical data item.
The Processes module exports items needed to accomplish the above sought synchronization. The definition module is:

```
DEFINITION MODULE Processes;

EXPORT QUALIFIED SIGNAL, init, SEND, WAIT, Awaited, StartProcess;

TYPE SIGNAL; (* Opaque type used by processes to communicate with each other *)

PROCEDURE Init(VAR S : SIGNAL); (* Initialize signal *)

PROCEDURE SEND(VAR S : SIGNAL); (* Send signal *)

PROCEDURE WAIT(VAR S : SIGNAL); (* Wait for signal *)

PROCEDURE Awaited(S : SIGNAL) : BOOLEAN;
(* Function to return if a signal is awaited *)

PROCEDURE StartProcess(P : PROC; WorkSpace : CARDINAL);
(* Start process P with WorkSpace bytes *)

END Processes.
```

In the above module, type SIGNAL is used for process inter-communication. Procedure Init is used to initialize a signal. Procedure WAIT suspends a process while waiting for a particular signal to be sent (using SEND) by another process.
Chapter 2

To demonstrate how the above data type and procedures are used, consider the following program, which takes an array of reals from the keyboard and calculates the corresponding average and standard deviation values. These statistics are evaluated using the sums of the observations and their squared values which can be evaluated concurrently. A coroutine is employed to update the sum of squares, while the main program calculates the sum of observations. Since each array element is accessed by the main program and the coroutine, we need to synchronize their access.

Let’s look at the listing and then resume our discussion.

```
MODULE Synchroniciry;

FROM Processes IMPORT SIGNAL, Init, SEND, WAIT, StartProcess;
FROM InOut IMPORT WriteString, WriteLn, ReadCard, WriteCard;
FROM RealInOut IMPORT ReadReal, WriteReal;
FROM MathLibO IMPORT sqrt;

CONST MAX = 100;

VAR Count, NumData : CARDINAL;
   GoAheadMakeMyDay : SIGNAL;
   SumX, SumXX, Average, StdDeviation : REAL;
   X : ARRAY [1..MAX] OF REAL;

PROCEDURE GetSumSquare;
(* Process to calculate sum of data squares *)

BEGIN
   (* Message displayed first time process is invoked *)
   WriteString('Start squaring');
   WriteLn;
   LOOP
      WAIT(GoAheadMakeMyDay); (* wait for a go signal *)
      WriteString('Squaring observation # ');
      WriteCard(Count,4);
      WriteLn;
      SumXX := SumXX + X[Count] * X[Count];
   END;
END GetSumSquare;
```
PROCEDURE GetData;
VAR i : CARDINAL;
BEGIN
  REPEAT
    WriteString('Enter number of data (<100) ');
    ReadCard(Numdata)
    UNTIL (NumData <= MAX) AND (NumData > 2);
  FOR i := 1 TO NumData DO
    WriteString('Enter observation # ');
    WriteCard(i,4);
    WriteString(' ');
    ReadReal(X[i]);
    WriteLn;
  END;
END GetData;

BEGIN
  GetData;
  WriteString('All data entered');
  WriteLn;
  Init(GoAheadMakeMyDay);
  (* Initialize counter and statistical summations *)
  Count := 1;
  SumX := 0.0;
  SumXX := 0.0;
  SendProcess(GetSumSquare,600);
END(GoAheadMakeMyDay);
LOOP
  WriteString('Surnrning observation # ');
  WriteCard(Count,4);
  WriteLn;
  SumX := SumX + X[Count];
  INC(Count); (* Increment global data counter *)
  (* Are all the observations processed? *)
  IF Count > NumData THEN EXIT END;
  (* Signal for GetSumSquare process to resume *)
  Send(GoAheadMakeMyDay);
END;
  Avg := SumX / FLOAT(NumData);
  StdDeviation := sqrt((SumXX - SumX * SumX /
                        FLOAT(NumData)) / (FLOAT(NumData - 1)));
  WriteString('Average = ');
  WriteReal(Avg,14);
  WriteLn;
  WriteString('Standard Deviation = ');
  WriteReal(StdDeviation,14);
  WriteLn;
END Synchronicity.
When the program starts it first prompts for keyboard data entry, performed by procedure GetData. The main program proceeds with a confirmation message followed by initializing the process signal. After the appropriate variable initializations process GetSumSquare is triggered. In turn it displays the "Start squaring" message once and waits for a signal. The control is briefly transferred back to the main program only to execute the SEND procedure and return us to the coroutine. The open loop is resumed and the message "Squaring observation 1" is displayed followed by the first update of the sum of squares. Having accessed the first array member, X[1], the coroutine signals to the main program for it to resume. The message "Summing observation 1" is displayed, sum updated and counter incremented. This is followed by a test to determine if all the data have been processed and the loop is accordingly exited. Otherwise the main program signals the coroutine to perform its task and another cycle of calculations is executed.
Chapter 3
The Compiler

The LOGITECH Modula-2 Compiler translates an ASCII text file with an extension of either .DEF or .MOD written in high level Modula-2 code — into a linkable object file with an extension of .OBJ containing low level machine code.

There are two versions of the LOGITECH compiler — a fully linked version, and an overlay version.

M2C.EXE is the fully linked version.

M2COMP.EXE is the overlay version and consists of a base file and six overlay files. This version uses less memory and so takes longer to compile a given module than M2C.EXE, but may be useful when memory is limited due to hardware or due to other applications present in memory at the same time.
3.1 How to Use the Compiler

To run the fully-linked version of the compiler, type:

\texttt{M2C} \texttt{[Enter]}

To run the overlay version of the compiler, type:

\texttt{M2COMP} \texttt{[Enter]}

If you are compiling with \texttt{M2C.EXE} from the \texttt{DOS} command line, you will see the banner with the version number and a prompt for the name of the module to be compiled:

\begin{verbatim}
C:\\TEMP> m2c
LOGITECH MODULA-2 Compiler, Rel. m.n
Copyright (C) 1983, 1984, 1985 LOGITECH
source file>_
\end{verbatim}

Enter the filename and any options (see \textbf{Section 3.8} on compiler options). The default drive is the current disk and the default extension for program and implementation modules is .MOD.

When this module is compiled, the compiler asks for another module. At this point, you can enter another filename or terminate the compiler by pressing \texttt{[Esc]}

You can also run the compiler by typing the filename(s) on the command line, thus:

\texttt{M2COMP (or M2C) <File 1> <File 2> ... <File X> [Enter]}

In this case the compiler will compile all the files in the specified order. At the end it returns automatically to \texttt{DOS} without any further request for source files.
NOTE

If you use the M2ASSIST environment in POINT, you will see a compile options prompt on the bottom line of the screen.

As soon as you enter the necessary options, you will see a new screen with information similar to that on the previous page.

To compile more than one file at a time with POINT:

Step 1: Save the file(s) to be compiled;
Step 2: Type the complete compile command on a scratch screen;
Step 3: Select Execute Selected Command from the QUIT+ETC menu.
3.2 Compiler Organization

The overlay version of the compiler is organized as a base part and several passes or 'overlays'. The base part remains in memory during the entire compilation and calls the passes sequentially. When loading these passes, the compiler assumes they are on the same drive as M2COMP.EXE, the compiler base.

The necessary overlay files are:

- M2COMP.EXE: compiler base
- M2OVLINI.OVL: initialization
- M2OVL1.OVL: syntax analysis and declaration
- M2OVL2.OVL: block analysis
- M2OVL3.OVL: code generation
- M2OVLSYM.OVL: symbol file generation
- M2OVLLLIS.OVL: lister

The fully linked version of the compiler is the single file M2C.EXE.

---NOTE---

During compilation, temporary work files are created on the current drive and directory, or where the environment variable M2TMP specifies. (You may specify a RAM disk to increase compilation speed.) These files are deleted before compilation ends.
3.3 Compiler Output Files

Several files are generated by the compiler. They are created in the current directory and are given the same file name as the source file, but with the appropriate extension:

.SYM Symbol file
Compiler output file with symbol table information. This information is generated during compilation of a definition module.

.REF Reference file
Compiler output file with debugger information, generated during compilation of an implementation or a program module.

.OBJ Object file
Compiler output file with generated 8086 object code in linker format, generated during compilation of implementation or program module.

.LST Listing file
Normally generated only if errors occur.
Chapter 3

3.4 Compilation of a Program Module

Compilation of a program module in which there are no errors generates a linkable object (.OBJ) file, and a debug reference (.REF) file.

If there are errors, the link and reference files are not produced, but a listing (.LST) file is produced.

The L option (see Section 3.8 on compiler options, below) tells the compiler to generate a listing file even if there are no errors.

Thus, if you type

M2COMP [J]

you get a screen that looks like this:

```
C:\TEMP > m2comp
LOGITECH MODULA-2 Compiler, Rel. m.n
Copyright (C) 1983, 1984, 1985 LOGITECH
source file> examp1 [J]
Syntax and Declaration Analysis
Terminal in file: B:Terminal.SYM
Block Analysis
Code Generation
Termination
The interactive setting of the options was: S+ /R+ /T+ /A- /O- /F+
code for 8086/8088 generated
Codesize: 90 bytes Datasize: 1 bytes
End Compilation

LOGITECH MODULA-2 Compiler, Rel. m.n
Copyright (C) 1983, 1984, 1985 LOGITECH
source file> [Esc]
---- no compilation
Termination
End Compilation
C:\\TEMP >
```

In the above screen, the file EXAMPLE1 is entered at the first source file prompt. [Esc] is pressed at the second source file prompt, in order to return to DOS.
If, from your TEMP directory, you enter

**M2COMP**

you will see a screen that resembles the one below. As with the screen just discussed, if you press **Esc** at the **source file** prompt, you will be returned to the **DOS** prompt.

```
C:\TEMP > m2comp
LOGITECH Modula-2 Compiler, Rel. m.n
   Copyright (C) 1983, 1984, 1985, 1987 LOGITECH
source file> ([ESC])
    --- no compilation
Termination
End Compilation
C:\TEMP >
```
Chapter 3

Syntax and Declaration Analysis, Block Analysis, and Code Generation denote the succession of activated compiler passes. If errors are detected by the compiler, compilation stops after the pass that finds the error. Errors are displayed on the screen, and a listing file is generated with error messages similar to those below.

When the error display is more than one page, the compiler will ask if you want to see more errors after each page.

Esc stops the error listing display.

Pressing any other key continues the error listing.

This is true unless the /Batch option is used, in which case the compiler will not ask for more errors.

In both cases, the compiler generates the .LST error listing.
3.5 Compilation of a Definition Module

Compilation of a definition (.DEF) module is similar to the compilation of a program module. However, when a definition module is successfully compiled, a symbol (.SYM) file is the result; when a program or implementation module is compiled the result is a linkable object (.OBJ) file.

The symbol file contains the declarations of the definition part in symbolic, compiler-readable format. It also contains a unique module key which is used to check consistency. If errors are detected by the compiler, then a listing file is generated instead of the symbol file.

---NOTE---
A Definition Module must be compiled before its Implementation Module.
A Definition Module must be compiled before any module that imports it.

Example:

```
C:\TEMP > m2comp find.def
LOGITECH MODULA-2 Compiler, Rel. m.n
Copyright (C) 1983, 1984, 1985 LOGITECH
source file> find.def
Syntax and Declaration Analysis
Symfile
Termination
End compilation
C:\TEMP >
```
3.6 Compilation of an Implementation Module

Compiling an implementation module is similar to compiling a program module.

When an implementation module is compiled, a symbol file for this module is needed. This symbol file is produced before compiling the implementation module, by compiling the corresponding definition module.

Compiler output files for implementation modules are the same as those generated when compiling a program module. A linkable object (.OBJ) file and a debug reference (.REF) file are generated as the result of a successful compilation. A listing file is produced only if there are errors.

```
C:\TEMP > m2comp find
LOGITECH MODULA-2 Compiler, Rel. m.n
Copyright (C) 1983, 1984, 1985 LOGITECH
source file> find.mod
Syntax and Declaration Analysis
Examp3 in file: A:Examp3.SYM
Storage in file: B:Storage.SYM
Block Analysis
Code Generation
Termination
The interactive setting of the options was:S+/R+/T+/A-/O-/F+
code for 8086/8088 generated
Codesize: 234 bytes Datasize: 56 bytes
End Compilation
C:\TEMP >
```
3.7 Symbol Files Needed for Compilation

Symbol files are used by the compiler for full inter-module checking. When a definition module is compiled, it generates a symbol file containing symbol table information. When the corresponding implementation part is compiled (or when a "client" module is compiled which imports it) the appropriate symbol file is read.

By default, the compiler first searches for symbol files on the disk/directory containing the source file. It uses the module name (truncated if necessary) as the filename, and a extension of .SYM. If a symbol file is not found on the first search, additional searches on other drives or directories are done automatically. (See the section on library search strategy for a complete description).

If a symbol file is not found, the compiler issues a message and asks for the file. This can be prevented, using the Autoquery option (see compiler options described below). If the Query option is turned on, the compiler will not perform any automatic searches. It will display the module name and let you enter the file name for every symbol file needed.

When the compiler asks for a symbol file, the request is repeated until an appropriate file is found or \( \text{Esc} \) is pressed. \( \text{Esc} \) tells the compiler that the file is not available. The compiler then stops at the end of the first pass, after listing all the required symbol files. This helps you detect any other missing files.
3.8 Compiler Options

When it reads the source file name, the compiler can also accept some options. Options are entered just after the filename, are preceded by a forward slash ( / ), and may use additional arguments ( + ) and ( - ). An option value is a predefined string that defines the state of the corresponding option. Possible values for the compiler options are listed below, followed by an explanation of their effects.

The default values in the following table are good unless you specify otherwise in a file which you can create named M2C.CFG. Keep this file in the M2EXE directory (where M2C.EXE resides) so it can be read by M2C.EXE or M2COMP.EXE. The default values M2C.CFG specifies will be applied to the filenames as if had been appended on the command line.
### 3.8.1 Table of Available Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Value for ON</th>
<th>Value for OFF</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>query</td>
<td>Query</td>
<td>NOQuery</td>
<td>NOQ</td>
</tr>
<tr>
<td>autoquery</td>
<td>Aquery</td>
<td>NOAquery</td>
<td>A</td>
</tr>
<tr>
<td>interactive</td>
<td>Interactive</td>
<td>Batch</td>
<td>I</td>
</tr>
<tr>
<td>listing</td>
<td>Listing</td>
<td>NOListing</td>
<td>NOL</td>
</tr>
<tr>
<td>error listing</td>
<td>EListing</td>
<td>NOEListing</td>
<td>EL</td>
</tr>
<tr>
<td>emulator/coprocessor</td>
<td>Emulator</td>
<td>Coprocessor</td>
<td>E</td>
</tr>
<tr>
<td>8086/80286</td>
<td>2</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>version</td>
<td>Version</td>
<td>NOVersion</td>
<td>NOV</td>
</tr>
<tr>
<td>statistics</td>
<td>STATistics</td>
<td>NOSTATistics</td>
<td>STAT</td>
</tr>
<tr>
<td>stacktest</td>
<td>S+</td>
<td>S-</td>
<td>S+</td>
</tr>
<tr>
<td>float test</td>
<td>F+</td>
<td>F-</td>
<td>F+</td>
</tr>
<tr>
<td>rangetest</td>
<td>R+</td>
<td>R-</td>
<td>R+</td>
</tr>
<tr>
<td>indextest</td>
<td>T+</td>
<td>T-</td>
<td>T+</td>
</tr>
<tr>
<td>alignment</td>
<td>A+</td>
<td>A-</td>
<td>A-</td>
</tr>
<tr>
<td>optimize</td>
<td>O+</td>
<td>O-</td>
<td>O-</td>
</tr>
<tr>
<td>headerinlisting</td>
<td>Header</td>
<td>NOHeader</td>
<td>H</td>
</tr>
<tr>
<td>footerinlisting</td>
<td>Footer</td>
<td>NOFooter</td>
<td>NOF</td>
</tr>
<tr>
<td>dateinlisting</td>
<td>DATE</td>
<td>NODATE</td>
<td>NODA</td>
</tr>
<tr>
<td>debug</td>
<td>Debug</td>
<td>NODEbug</td>
<td>D</td>
</tr>
<tr>
<td>symbol</td>
<td>SYmbol</td>
<td>NOSYmbol</td>
<td>SY</td>
</tr>
<tr>
<td>m2linker</td>
<td>M2L</td>
<td>NOM2L</td>
<td>M2L</td>
</tr>
</tbody>
</table>

To change the compile option defaults, put the desired settings (in the same syntax of the command line) into a file you create named M2C.CFG. In the above list, use either **Upper Case** letters (as in the bold letters in the table, above), or the complete name to specify an option. Optionally, **R, S, A, O, and T** may use the + argument.
3.8.2 Description of the Options

/Q Query
/NOQ Autoquery
/A
/NOA

 Defines the search mechanism for the symbol files of imported modules. The following table shows the possible combinations of option settings and the corresponding behavior of the compiler:

<table>
<thead>
<tr>
<th>Query</th>
<th>Autoquery</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query</td>
<td>Aquery</td>
<td>Ask for filenames</td>
</tr>
<tr>
<td>Query</td>
<td>NOAquery</td>
<td>Ask for filenames</td>
</tr>
<tr>
<td>NOQuery</td>
<td>Aquery</td>
<td>Search for file by default strategy. If not found, ask for filename.</td>
</tr>
<tr>
<td>NOQuery</td>
<td>NOAquery</td>
<td>If not found, compile ends.</td>
</tr>
</tbody>
</table>

The default setting for these two options is /NOQuery and /Aquery.

/I Interactive
/B Batch

 Tells the compiler whether to run interactive or as a batch job. In interactive mode, display of error messages is stopped after a screen page and is resumed by hitting a key. This facility is turned off in the batch mode. Note: Autoquery is not affected by this option.
/L
/NOL
/EL
/NOEL

Listing

Error Listing
Says whether or not to generate a listing. This table shows how option combinations create corresponding behavior by the compiler.

<table>
<thead>
<tr>
<th>Listing</th>
<th>EListing</th>
<th>NOEListing</th>
<th>NOEListing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Listing</td>
<td>EListing</td>
<td>NOEListing</td>
<td>NOEListing</td>
</tr>
<tr>
<td>NOListing</td>
<td>EListing</td>
<td>NOEListing</td>
<td>NOEListing</td>
</tr>
<tr>
<td>NOListing</td>
<td>NOEListing</td>
<td>NOEListing</td>
<td>NOEListing</td>
</tr>
</tbody>
</table>

A listing file is always generated. Same as above. Detected errors generate a short error listing file with only the erroneous lines with error messages. No listing generated.

In all cases the compiler writes the lines with errors and the error message on the screen. Before each compilation, the compiler deletes the corresponding listing file ( <filename>.LST ).

/E
/C

Emulator
Coprocessor
This affects code generation for floating point arithmetic. If set to coprocessor, the compiler generates inline code for the Intel 8087 numeric processor. Otherwise, it generates code for the LOGITECH REAL ARITHMETIC EMULATOR.

/V
/NOV

Version
The compiler displays information about the running version, for example, processor and operating system flags.

/STAT
/NOSTAT

Statistics
At the end of a compilation the compiler displays statistics on the generated code.
Chapter 3

/S+  Stack test
/S-  

/R+  Range and Overflow test
/R-  

/F+  Float Arithmetic Error Test
/F-  

/T+  Index and NIL pointer test
/T-  

/A+  Alignment
/A-  
Affects the variable and record field allocation. If set to A+, all
variables except single bytes are allocated on even boundaries. If set to
A-, no special effort is made to allocate variables on even boundaries.
Choose either to save memory space A-, or increase program execution
speed A+.

/O+  Register Trace Optimization
/O-  
Reduces code size and increases execution speed. (See the following
subsection for more information.)

/8  8086
/2  80286
Affects code generation. If set to 8086, the compiler generates code for
the 8086/8088. If set for 80286, the compiler generates code for the
80186/80286. The advanced instructions used are ENTER, LEAVE,
PUSH Immediate, Shift/Rotate Immediate, and Integer Immediate
Multiply.
The Compiler

/H
/NOH
/F
/NOF
/DA
/NODA

Header in Listing
Footer in Listing
Date in Listing

These define the format of the generated listing file. The header option says whether a page header line is generated or not. The footer option defines whether a page footer line is generated or not. The date option says whether the date information is generated within the header line. The format of a page header line is:

MODULA-2 <filename.ext> <date> <page #>

Footer line text can only be defined in the compiler parameter module.

/D
/NOD

Debug

This tells whether or not to generate the reference (.REF) file. This file contains the necessary information for the symbolic debugger.

/M2L
/NOM2L

LOGITECH Linker Information

Instructs the compiler to produce extra information in the .OBJ file to allow the LOGITECH Linker to perform some improvements. The extra information lets the linker remove unreferenced procedure calls and constants from the .EXE file, and also allows the linker to search automatically for the files to be linked.

/SY
/NOSY

Symbol

Produces symbol that will be used by debuggers. Refer to Appendix E for naming conventions if you don't use the LOGITECH Debugger.
3.8.2.1 The Optimize Option

This option can be used to tighten code. This is done by tracking register content. Our test numbers show a reduction of up to 10% in size, and about the same increment in execution speed. The gain in speed and reduction in size may vary substantially from program to program. Compilation time is longer when optimization is ON ( /O+ ).

To perform optimization on specific areas of source code (See Section 3.9), you can use (*$0+*), (*$0-*), and (*$0=*).

When the compiler is instructed to use the optimize option, it considers that the program being compiled satisfies some requirements. The average Modula-2 application usually meets these requirements but the programmer does have the ability to cheat the compiler.

In the following cases the requirement outline is sketched and some examples of bad and good program behaviour is given.
Case 1: Using ADR ()

A memory update is not recognized to kill the previous value when the following conditions are true:

- Location type is any scalar type, and is designated either by a variable name or a record field name;
- A pointer is initialized with the address of that variable (via the ADR function);
- The location is updated before use by its name and then by the de-referenced pointer;
- This happens within an EBB* boundary.

This may lead to generation of bad code. To be safe, use in-line option (*$O+/-*).

Example 1 shows where failing to put in-line options is an improper use of optimization.

```
VAR p: POINTER TO INTEGER;
  i, j: INTEGER;
  . .
----------------------------> (*$O-*)
i := something;
p := ADR(i);
p^ := somethingElse;
j := i;
----------------------------> (*$O+*)
  . .
```

* EBB or Extended Base Block. A code section that must be completely executed before the final statement in that section can be executed.
Chapter 3

Case 2: Calling with VAR Parameter

The compiler doesn't perform inter-procedural alias analysis which sometimes leads to bad code, as follows:

**Example 2**

```
PROCEDURE P
VAR i: INTEGER;
BEGIN
  ...
  Q(i, i);
  ...
END P;

PROCEDURE Q (VAR x, y: INTEGER);
VAR a, b: INTEGER;
BEGIN
  ...
  ------------------> (*$0-*)
  x := a;
  y := a + 1;
  b := x;
  ------------------> (*$0+*)
  ...
END Q.
```

In procedure Q, variable b gets a bad value, since x and y are aliased to the same memory location. One way around this might be to use (*$0-/+*) to encapsulate the three statements in procedure Q.

Case 3: Safe Cases of Aliases

Cases of aliases other than those in Case 1 and Case 2, such as fields in a variant record (e.g., p^ and q^ when p=q, and a[i] and a[j] when i=j) are recognized by the compiler without your intervention.
Case 4: Using CODE Statement

Another situation that unsafely relates to the optimize option is playing tricks with the CODE statement. CODE statements are well treated by the compiler if they are "self-contained"—i.e., if they don't jump into the middle of another statement.

When the compiler encounters a CODE call, it resets all information about the content of the registers and resumes collecting at the next statement. This information can be faked by a misplaced jump. The examples illustrate the possible situations.

Example 3

```
CODE (073H, 01H); (* JNB @FOO *)
CODE (040H); (* INC AX *)
GETREG(AX, error); (* @FOO: MOV error, AX *)
IF error THEN ...
```

The code in Example 3 above, is acceptable because after compilation of these statements, the registers descriptor will be properly initialized with the information that AX contains the variable "error".

Example 4

```
CODE (073H, 04H); (* JNB @FOO *)
CODE (040H); (* INC AX *)
GETREG(AX, error); (* @FOO: MOV error, AX *)
IF error THEN ...
```

The code in Example 4 above, is not acceptable because, after compilation, the register descriptor will be wrongly initialized with the information that AX contains the variable "error". This is not true if the JNB is taken.

This code style, even if accepted by the compiler, is not clean. It relies, for example, on the fact that the compiler will generate three bytes of code to translate the GETREG statement.

To repair the code in Example 4, either rearrange the sequence, or use (*$0-/*+*).
3.9 Compiler Directives in Modules

Certain compiler directives may be specified in the source text of a module. These directives must appear immediately at the beginning of a comment and consist of $<\text{letter}>$<setting>, without any intervening or preceding spaces.

<table>
<thead>
<tr>
<th>Letter</th>
<th>Definition</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Stack overflow test</td>
<td>S+</td>
</tr>
<tr>
<td>R</td>
<td>Subrange and arithmetic overflow test</td>
<td>R+</td>
</tr>
<tr>
<td>F</td>
<td>Real arithmetic error test</td>
<td>F+</td>
</tr>
<tr>
<td>T</td>
<td>Index test (arrays, case) and NIL pointer test</td>
<td>T+</td>
</tr>
<tr>
<td>A</td>
<td>Word alignment for variables and record fields</td>
<td>A-</td>
</tr>
<tr>
<td>O</td>
<td>Register Trace Alignment</td>
<td>O-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Setting</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Set the option ON</td>
</tr>
<tr>
<td>-</td>
<td>Set the option OFF</td>
</tr>
<tr>
<td>=</td>
<td>Revert to setting before last</td>
</tr>
</tbody>
</table>

**Example**

```
MODULE x; (*$T+$)  
  ...
  (*$T$)
  CASE i OF
  ...
  END
  (*$T$)
  ...
  END x
```

test code is generated
no test code is generated
test code is generated (i.e. the prior value is restored)
3.10 Compiler Messages

There are two types of compilation errors:

- Errors detected in the source text printed on the listing and displayed on the screen.
- Operational errors displayed on the screen.

3.10.1 Source Text Errors

These errors appear in the listing file, marked under the offending line by a ^ and the error number. The source line and error message are also displayed on the screen as they are written to the listing. Compiler error messages are also listed in an appendix.

3.10.2 Compiler Operational Messages and Errors

Upon termination, the compiler sets the MS-DOS errorlevel system variable. This variable can be checked in a batch file. The generated values are:

- 0 successful compilation
- 1 abnormal termination due to internal errors
- 2 incomplete compilation due to missing files
- 3 source program error

During operation of the compiler the following messages and errors may be displayed:

Assertion of compiler
  internal reference: xx
  at source line : nn
Please send a bug-report to LOGITECH

We hope you never get this message. It is displayed when an internal consistency check of the compiler fails. If you get this message, please contact LOGITECH with a copy of the program which caused the error. The source line information may help you find a work-around. The internal reference is an indicator for the kind of problem that occurred and will help LOGITECH locate it.
cannot load ...

There is not enough memory to allocate all data areas the compiler needs or to load a compiler overlay. See the chapter on Program Execution for details.

EOF on Control

We hope this message never occurs. You see it when an internal consistency check of the lister fails. If you get this message, contact LOGITECH with a copy of the program that caused the error.

error

The compiler detected errors in your program. These errors will appear on the screen and in the listing file.

error message first element on control

See EOF on Control.

file creation failed

Your disk directory is probably full. When running under DOS, this message may also appear if you did not boot your operating system from a disk that contains a CONFIG.SYS file, as described in the installation section of this manual.

file not found

A source or symbol file was not found. The compiler will repeatedly request the filename. You should either type the correct filename or press [Esc] if the required file is missing. When running under DOS, this message may also appear if you did not boot your operating system from a disk that contains a CONFIG.SYS file, as described in the installation section of this manual.

<file name> halted

An overlay of the compiler terminated with an unexpected status. This might happen if you stop the compiler with [Ctrl]-[Break] or [Ctrl]-[C].
heap overflow
There is not enough memory to allocate all data areas the compiler needs, or to load a compiler overlay. See the chapter on Program Execution for details.

illegal option: <option typed>
Please refer to the list of valid compiler options.

Incorrect line number on Control
See EOF on Control.

incorrect module name
The module name found in the symbol file does not correspond to the name of the module for which a symbol file is needed. Make sure you enter the right filename.

NControl too small
See EOF on Control.

no compilation
No compilation takes place because no source file was specified.

no file
You typed \texttt{Esc} when asked to enter a filename, and did not supply any file.

not catalogued: <extension>
The compiler had problems closing the file with the given extension. Make sure that the disks are in the drives.

not deleted: <extension>
The compiler had problems deleting the file with the given extension. Make sure that the disks are in the drives.
output disk full

Because of insufficient space on your disk, the compiler has stopped. You should delete superfluous files.

<program name> program not found

A compiler overlay was not found on the disk where it is expected to be. Please check whether you installed Modula-2 properly. When running under DOS, this message may also appear if you did not boot your operating system from a disk that contains a CONFIG.SYS file, as described in the installation section of this manual.

stack overflow

Not enough memory to allocate all data areas the compiler needs or to load a compiler overlay.

symbol files missing

The compiler could not find all the symbol files for the imported modules. Therefore type checking is impossible and compilation stops. Check that the corresponding definition modules have been compiled and that all necessary symbol files have been specified correctly.

<file name> warned

An overlay of the compiler terminated with an unexpected status. This might happen if you stop the compiler with Ctrl-Break or Ctrl-C.

wrong symbol file

The file found is not a correct symbol file. Most likely, the file isn’t a symbol file at all, or it was not generated by the same compiler.
3.11 Compiler Table Limits

The following error messages depend on some internal compiler table sizes. The following list gives the description of the errors and the actual table size of the compiler:

7 too many identifiers (identifier table full)

Identifier table holds 30,000 characters. This is the limit on the total number of characters of all distinct identifiers in one module and the exported identifiers of its imported modules.

Contrary to readable, self-documenting programming style, shorter identifiers and use of the same identifiers in different scopes helps avoid this message.

8 too many identifiers (hash table full)

Hash table holds 3571 identifiers. This limits the number of distinct identifiers in one module, including all the identifiers exported by the imported modules.

The same identifier names in different scopes helps avoid this message.

205 implementation restriction: procedure too long

Code size per procedure limited to 5000. Split the procedure into smaller entities.

206 implementation restriction: statement table overflow

Number of statements per procedure limited to 1000. Split the procedure into smaller entities.

209 expression too complicated: jump table overflow

Jump table size is 50 entries. This determines the number of possible short circuit jumps in a boolean expression.

Try breaking the expression into several temporary expressions.

210 too many globals, externals, and calls (linker table overflow)

The linker table holds fixup information for the linker. The size of this table is 850 entries per procedure. Access to an imported variable generates one entry; a call to an imported procedure generates two entries. A forward call to a local procedure generates one entry.

Split the procedure into smaller sections, or reduce frequency of access to imported variables and calls to imported procedures.
3.12 Compiler Error Messages

0: illegal character in source file
1: 
2: constant out of range
3: open comment at end of file
4: string terminator not on this line
5: too many errors
6: string too long
7: too many identifiers (identifier table full)
8: too many identifiers (hash table full)

20: identifier expected
21: integer constant expected
22: ] expected
23: ; expected
24: block name at the END does not match
25: error in block
26: := expected
27: error in expression
28: THEN expected
29: error in LOOP statement

30: constant must not be CARDINAL
31: error in REPEAT statement
32: UNTIL expected
33: error in WHILE statement
34: DO expected
35: error in CASE statement
36: OF expected
37: : expected
38: BEGIN expected
39: error in WITH statement

40: END expected
41: ) expected
42: error in constant
43: := expected
44: error in TYPE declaration
45:  '(', expected
46:  MODULE expected
47:  QUALIFIED expected
48:  error in factor
49:  error in simple type

50:  '...', expected
51:  error in formal type
52:  error in statement sequence
53:  '.', expected
54:  export at global level not allowed
55:  body in definition module not allowed
56:  TO expected
57:  nested module in definition module not allowed
58:  ').' expected
59:  '..' expected

60:  error in FOR statement
61:  IMPORT expected

70:  identifier specified twice in importlist
71:  identifier not exported from qualifying module
72:  identifier declared twice or illegal forward reference to this identifier
73:  identifier not declared
74:  type not declared
75:  identifier already declared in module environment
76:  too many nesting levels
78:  value of absolute address must be of type CARDINAL
79:  scope table overflow in compiler

80:  illegal priority
81:  definition module belonging to implementation not found
82:  structure not allowed for implementation of hidden type
83:  procedure implementation different from definition
84:  not all defined procedures or hidden types implemented
85:  name conflict of exported object or enumeration constant in environment
86:  incompatible versions of symbolic modules
87:  

97
Chapter 3

88: function type is not scalar or basic type
89:
90: pointer-referenced type not declared
91: tagfieldtype expected
92: incompatible type of variant-constant
93: constant used twice
94: arithmetic error in evaluation of constant expression
95: incorrect range
96: range only with scalar type
97: type-incompatible constructor element
98: element value out of bounds
99: set-type identifier expected
100: structured type too large
101: undeclared identifier in export list of the module
102: range not belonging to base type
103: wrong class of identifier
104: no such module name found
105: module name expected
106:
107: set too large
108:
109: scalar or subrange type expected

110: case label out of bounds
111: illegal export from program module
112: code block for modules not allowed

119: illegal variable as FOR loop counter

120: incompatible types in conversion
121: this type is not expected
122: variable expected
123: incorrect constant
124: no procedure found for substitution
125: unsatisfying parameters of substituted procedure
126: set constant out of range
127: error in standard procedure parameters
128: type incompatibility
129: type identifier expected
130: type impossible to index
131: field not belonging to a record variable
132: too many parameters
133: function parenthesis missing
134: reference not to a variable
135: illegal parameter substitution
136: constant expected
137: expected parameters
138: BOOLEAN type expected
139: scalar types expected
140: operation with incompatible type
141: only global procedure or function allowed in expression
142: incompatible element type
143: type incompatible operands
144: no selectors allowed for procedures
145: only function call allowed in expression
146: arrow not belonging to a pointer variable
147: standard function or procedure must not be assigned
148: constant not allowed as variant
149: SET type expected
150: illegal substitution to WORD or BYTE parameter
151: EXIT only in LOOP
152: RETURN only in PROCEDURE
153: expression expected
154: expression not allowed
155: type of function expected
156: integer constant expected
157: procedure call expected
158: identifier not exported from qualifying module
159: code buffer overflow
160: illegal value for code
161: call of procedure with lower priority not allowed
170: global data too large (more than 64K bytes)
171: local data too large (more than 32K bytes)
172: parameter data too large (more than 32K bytes)
Chapter 3

200: compiler error
201: implementation restriction
202: implementation restriction: FOR step too large
203: implementation restriction: boolean expression too long
204: implementation restriction: expression too complicated
205: implementation restriction: procedure too long
206: implementation restriction: statement table overflow
207: implementation restriction: illegal type conversion
208: expression too complicated: jump table overflow
209: too many globals, externals and calls (linker table overflow)
210: implementation restriction: code >= 64K bytes
211: not further specified error
212: division by zero
213: index out of range or conversion error
214: case label defined twice
215: DEFINITION expected
Chapter 4
Linking Modula-2 Files

The .OBJ files you obtain after the compilation can be linked by the LOGITECH Linker or with the linker that comes with your version of DOS.

The LOGITECH Linker is part of the LOGITECH Modula-2 Toolkit and runs either from the DOS command line or from the M2ASSIST extension of the POINT Environment. The LOGITECH Linker supports a multi-layer overlay scheme, is able to link by procedure rather than by module (this will make your executable files smaller), and is able to find all the needed files automatically.

Please refer to the manual for the linker you will be using for details on its use.

Caution

The DOS Linker is not case-sensitive.

Thus: if you have two symbols that differ only in case, the DOS Linker displays

Symbol defined twice

One solution to this problem is to use the /NOSYMBO compiler option, which defines fewer symbols.

Another solution is to use a linker that recognizes the /NOIGNORECASE option

Note: your object files may not link with DOS Linkers older than version 2.30.
Chapter 5
Version Checking

All modules in a program must be compiled with consistent versions of module definitions.

When you change a module definition .DEF file, you must also recompile all program and implementation modules using that module before you can create a new executable program. When you compile a changed .DEF file, this creates a new .SYM file which is incompatible with any other version of that module. Even if you don't change anything in the definition part, recompiling it creates a new version of the .SYM file.
Chapter 5

5.1 Module Keys and Version Checking

LOGITECH Modula-2 checks for version consistency and keeps inconsistent versions from being compiled together.

Version checking is simple in concept, but can be complex in application. Each time you recompile a .DEF file it creates a different module key for the resulting .SYM file.

Once you compile a .DEF definition file, you can compile its "client" implementation module which uses the definition part. These other modules will import the module, and the compiler will find the compiled version of the definition part, and use it to fully check the module being compiled. The module key of the referenced definition parts are in the compiled output.

At compile, link and load time, LOGITECH Modula-2 verifies that all the keys included for a given definition module are the same. This guarantees that all modules which share an interface are compiled with the same version of the interface. This ensures the consistency of the program, as if there was only one source file, compiled all at once.

5.2 Version Errors and How to Fix Them

If the version consistency rule is broken, you will get a version error during either compilation, linking, or (sub)program loading. The following sections describe the typical cause and some possible corrections for version errors.

\[NOTE\]

The \texttt{M2MAKE} utility from the \textit{LOGITECH Modula-2} Toolkit is the easiest way to resolve version errors.
5.3 Version Errors During Compilation

A version error while compiling module A.MOD can only arise if there is some definition module X.DEF that is imported by two different paths into A.MOD, and the version imported by one path is not the same as the version imported on the other path.

Look for a moment at the following example:

- A.MOD imports B.DEF and C.DEF
- B.DEF imports X.DEF
- C.DEF imports X.DEF

Suppose that we compile as follows:

- X.DEF becomes X.SYM (version 1)
- B.DEF becomes B.SYM (uses version 1 of X)
- X.DEF becomes X.SYM (version 2)
- C.DEF becomes C.SYM (version 2 of X)
- A.MOD becomes A.OBJ

There will be a version error when A.MOD is compiled, because the version of X.DEF imported through B.DEF is not the same as the version imported through C.DEF. The recompilation of X.DEF is the source of the version conflict. Before A.MOD can be compiled, B.DEF must be recompiled with the newer version of X.DEF.
Chapter 5

5.4 Version Errors During Linking

When two or more modules are linked together, a version error can occur if some definition module has been used in two different versions by the linked modules.

Example:

<table>
<thead>
<tr>
<th>Module</th>
<th>Action</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAIN.MOD</td>
<td>imports</td>
<td>Terminal, InOut</td>
</tr>
<tr>
<td>INOUT.DEF</td>
<td>defines</td>
<td>InOut</td>
</tr>
<tr>
<td></td>
<td>imports</td>
<td>nothing</td>
</tr>
<tr>
<td>INOUT.MOD</td>
<td>implements</td>
<td>InOut</td>
</tr>
<tr>
<td></td>
<td>imports</td>
<td>Terminal</td>
</tr>
<tr>
<td>TERMINAL.DEF</td>
<td>defines</td>
<td>Terminal</td>
</tr>
<tr>
<td></td>
<td>imports</td>
<td>nothing</td>
</tr>
<tr>
<td>TERMINAL.MOD</td>
<td>implements</td>
<td>Terminal</td>
</tr>
<tr>
<td></td>
<td>imports</td>
<td>nothing</td>
</tr>
</tbody>
</table>

Then suppose these compilations are done:

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>TERMINAL.DEF</td>
<td>TERMINAL.SYM</td>
</tr>
<tr>
<td>INOUT.DEF</td>
<td>INOUT.SYM</td>
</tr>
<tr>
<td>INOUT.MOD</td>
<td>INOUT.OBJ</td>
</tr>
<tr>
<td>TERMINAL.MOD</td>
<td>TERMINAL.OBJ</td>
</tr>
<tr>
<td>TERMINAL.DEF</td>
<td>TERMINAL.SYM</td>
</tr>
<tr>
<td>MAIN.MOD</td>
<td>MAIN.OBJ</td>
</tr>
</tbody>
</table>

Under these conditions, linking MAIN.MOD generates a version conflict between the version of TERMINAL.SYM used by MAIN.MOD, and the version used by TERMINAL.MOD and INOUT.MOD. One solution is to recompile INOUT.MOD and TERMINAL.MOD with the new TERMINAL.SYM and link again.
The version conflict will be shown by the linker as an unresolved public symbol with the following format:

```
KEY__DDMMYY_HHMM_OF_<moduleName>
```

where YY is for the last two digits of the year, MMM for the first three letters of the month, DD for the two digits of the day of the month, HH for the two digits for the hour, and MM, the two digits for the minutes.

Figure 5-1: Sample of Version conflict at link time
Chapter 5

5.5 Version Errors During Loading

When the LOGITECH Modula-2 Overlay schema (which is only accessible through the LOGITECH Linker) is used to build an application program, an additional check is made by the LOGITECH Modula-2 system. The system can detect inconsistency at load time between the base and overlay layers. This error will happen when an incorrect version of the .MAP file is used when the overlay is linked. Here is a typical case:

A base layer is compiled and linked to produce MYPROG.EXE and MYPROG.MAP. Next, a version of MYOVL1.OVL is produced, using MYPROG.MAP as the base .MAP file.

Later, some modules of MYPROG.EXE are changed and the base layer is recompiled and relinked. This generates a second version of MYPROG.MAP that is different from the first version of MYPROG.MAP that was used to generate MYOVL1.OVL.

The difference in .MAP versions will be flagged at run time as inconsistent.

Remedy this situation by relinking the overlay programs with the new .MAP file version.
Chapter 6
Interfacing Other Languages

LOGITECH Modula-2 uses the standard .OBJ object file format of DOS. This means you can link LOGITECH Modula-2 Compiler output files with .OBJ object files produced by other compilers or by the assembler.

You need three pieces of information for an interface between different languages:

- Precise symbol names in the .OBJ file as procedure entrypoint and variables: C puts an _ (underscore) before symbols; other languages put symbols in upper case.
- Calling conventions:
  When one procedure calls another, parameter information is passed between them. Each type of information is put on the stack in an order that is dependent on the language and its implementation. A function also returns information into registers.
- How the run-time supports of the different languages interact.

LOGITECH Modula-2 gives you three ways to handle this interface.

- You can follow the LOGITECH Modula-2 conventions described in Appendix B.9, B.10, and B.11. Section 6.1 shows you how to write such an Assembly program. If the procedure to be called is in a high-level language, create an Assembly interface.
- You can use Modula-2 low-level features like EXTCELL, CODE, SEGREG, GETREG in order to follow the convention of the external procedure that you call. See Section 6.2.
- LOGITECH Modula-2 has an extension which helps you interface other languages by defining foreign definition modules. This is described in Section 6.3.
6.1 Assembly Implementation with Modula-2 Conventions

This section describes the Assembly language routine in Figure 6-1, which can be called by a LOGITECH Modula-2 program. This routine in turn calls the Write procedure from the Terminal module of the standard Modula-2 library.

Follow these steps:

Step 1: Write a Modula-2 Definition File.
This .DEF file will reference the names of your Assembly language procedures, variables, and their structures for the Modula-2 program listed in Figure 6-2.

Step 2: Compile the .DEF file.
This will create a .SYM file.

Step 3: Write an Assembly Language Program.
See the sample listing in Figure 6-1. You need to define an initialization routine which will be executed before the main part of your program is run. In Modula-2, each module which is imported must be initialised before being used.

Since Modula-2 performs version checking, your Assembly program must define a public symbol with information about your .DEF file. This is called the key; it contains the date of the .SYM file you created at Step 2.

The key is defined as:

```
KEY__ <date of SYMfile> _OF_ <module name>
```

KEY__ uses two underline characters with no break;
<date of SYMfile> uses the case-sensitive format ddmmyy_hhmm where;
- dd stands for the day;
- mmm, the month;
- yy, the year;
- "_" is a separator;
- hh stands for the hour; and
- mm stands for the minute.

When compiling a Modula-2 implementation module, the compiler generates this key, and at the same time, makes sure that your implementation corresponds to your definition. As you write in Assembly, remember: an automatic check cannot be performed to insure that your Assembly code follows the definition; that is your responsibility!
Step 4: Assemble the .ASM File.

Use an assembler to produce a .OBJ file. Remember that *Modula-2* is case sensitive, so use this option in your assembler.

Step 5: Link the different .OBJ Files.

If you use a *DOS* linker, include the names of all the necessary .OBJ files on the command line, including the file you created with the assembler.

With the *LOGITECH Linker*, you don’t need to specify all the filenames: the name of the main program file is enough. If your *Assembly* program imports a file not in the standard library, then you must specify that file on the command line when you link.

Step 6: Run the .EXE file.

The .EXE file produced by the linking can be run like any other .EXE file.
Chapter 6

This sample program shows the interface between Assembly and LOGITECH Modula-2

- The initialization procedure writes "hello"
- The main module calls the AsmWrite procedure to write a string; this procedure uses the procedure Write in the Terminal module of the standard library.

Figure 6-1: EXASM Program Listing.

```
TITLE ExAsm

; The symbol KEY_<date SYM file>_OF_<modulename> is needed for version checking of Modula-2. Its value has no special meaning, usually 0.

PUBLIC KEY_<19jun87_2007_OF_ExAsm
KEY_<19jun87_2007_OF_ExAsm EQU 0

; Exported Procedures
PUBLIC L_AsmpWrite_ExAsm

; Initialization entrypoint
PUBLIC $INIT__ExAsm

; Exported Variables
PUBLIC text__ExAsm

; Used procedures
EXTRN L_Write__Terminal: FAR
EXTRN $INIT__Terminal: FAR

SEGMENT BYTE PUBLIC 'CODE'
ASSUME CS: ExAsm_TEXT

;------------------------------------------------------------------------------------
;------------------------------------------------------------------------------------
Asmwrite (VAR str: ARRAY OF CHAR)
PUSH BP  ; save BP and SP
MOV BP, SP ; BP should not be changed -
           ; It will be used to access the parameters

MOV AX, SEG ExAsm_DATA
MOV DS, AX

ASSUME DS: ExAsm_DATA

; i:=0;
MOV AX, 0
MOV i, AX

; WHILE (i<=HIGH(str)) AND (str[i]<>0c)
while:
  MOV AX, i
  CMP AX, 10[BP] ; HIGH(str) is offset at 10
  JNBE end
  JNBE end
  LES BX, 6[BP] ; ADR(str) is offset at 6
```

112
Figure 6-1: EXASM Program Listing (cont’d)

```assembly
  MOV SI, AX
  MOV AL, ES: [BX+SI]
  CMP AL, 0
  JE end

; Write (str[i]);
  PUSH AX
  CALL L_Write_Terminal

; INC (i);
  MOV CX, SEG ExAsm_DATA
  MOV DS, CX
  INC i
  JMP while

; end:
  MOV SP, BP
  POP BP
  RET 6
  ; restore SP and BP
  ; release the 6 bytes
  ; of the parameters

L_AsmWrite_ExAsm ENDP

;---------------------------------------------------------------------------------------------------------------------
$INIT_ExAsm PROC FAR
;---------------------------------------------------------------------------------------------------------------------
  MOV AX, INIT_FLAG_DATA ; Test if already
  MOV DS, AX ; initialized
  ASSUME DS: INIT_FLAG_DATA
  MOV AL, 1 ; TRUE
  XCHG AL, BYTE PTR FLAG_ExAsm
  OR AL, AL ; is it FALSE or TRUE ?
  JNE End_init ; skip if TRUE

; Execution of the init code -- here, for example, we write 'Hello'
; after initialization of all needed (imported) modules

  CALL $INIT_Terminal
  MOV AX, 7
  PUSH AX
  MOV AX, SEG ExAsm_DATA
  PUSH AX
  MOV AX, OFFSET text__ExAsm
  PUSH AX
  CALL L_AsmWrite__ExAsm

End_init:
  RET
$INIT_ExAsm ENDP

ExAsm_TEXT ENDS

INIT_FLAG_DATA SEGMENT WORD PUBLIC 'FAR_DATA'
  FLAG_ExAsm DB 0 ; initialization flag
INIT_FLAG_DATA ENDS

ExAsm_DATA SEGMENT WORD PUBLIC 'FAR_DATA'
text__ExAsm DB 'Hello'
i DW 0
ExAsm_DATA ENDS
END
```

Figure 6-1: EXASM Program Listing (End)
DEFINITION MODULE ExAsm;
PROCEDURE AsmWrite (VAR str : ARRAY OF CHAR);
END ExAsm.

**Figure 6-2: Modula-2 listing for EXASM.DEF**

MODULE CallAsm;
FROM ExAsm IMPORT AsmWrite;
Var str : ARRAY [0..5];
BEGIN
  str := "folks!";
  AsmWrite(str);
END CallAsm;

**Figure 6-3: Modula-2 listing for CALLASM.MOD**
6.2 Use of Low-Level Features

The listing in Figure 6.4 tells how to call other high level languages, in this case a C library output routine.

*LOGITECH Modula-2* low level EXT CALL, CODE, SETREG, and GETREG features.

```plaintext
MODULE Example;
FROM SYSTEM IMPORT EXT CALL, CODE, AX;
PROCEDURE CWrite(text: ARRAY OF CHAR);
BEGIN
  FOR i := 0 TO HIGH(text) DO
    SETREG(AX, text[i]);
    CODE(050H); (* PUSH AX *)
    (* we call the C library routine putchar *)
    EXT CALL ("_putchar");
    CODE(058H); (* POP AX *)
    (* caller must POP parameters after the call in C *)
  END;
END CWrite;
END Example.
```

**Figure 6.4:** *LOGITECH Modula-2* low level EXT CALL, CODE, SETREG, and GETREG

**WARNING**

In this method, nothing is done to initialize C run-time support. This means that you cannot use the C routines which depend on run-time support.
6.3 Foreign Definition

The *LOGITECH Modula-2 Development System* lets you declare foreign definition modules that interface other languages. These modules have the same syntax than the standard Modula-2 definition modules except that they should begin with the keyword **FOREIGN** which may be followed by a qualifier indicating the language to interface:

\[
\text{ForeignDefinitionModule} = \text{FOREIGN [LanguageQualifier] DefinitionModule}\]

LanguageQualifier = C

A foreign definition module could look like this:

```plaintext
FOREIGN DEFINITION MODULE ForeignModule;
EXPORT QUALIFIED ForeignProc;
PROCEDURE ForeignProc ( num : INTEGER );
END ForeignModule.
```

**Figure 6-5 Foreign Definition Module with no Qualifier**

or like this:

```plaintext
FOREIGN C DEFINITION MODULE ForeignModule;
EXPORT QUALIFIED ForeignProc;
PROCEDURE ForeignProc ( num : INTEGER );
END ForeignModule.
```

**Figure 6-6 Foreign Definition Module with C Qualifier**

6.3.1 Conventions Used

The implementation of the foreign definition module can be in any language whose compiler generates the standard .OBJ file format, and uses both the symbol name and the parameter passing conventions explained below.
6.3.1.1 Symbol Name Convention

If you don’t use any language qualifier, the compiler will assume that the symbol names generated by the foreign compiler are the very same symbol names as those declared in the foreign definition module. This is convenient when interfacing Assembly language because the symbol names generated by the assembler in the .OBJ file are the same as those in the source code. Pascal compilers usually generate the same symbol names too in the .OBJ file, but all in upper case. (This is not a problem when linking with the DOS linker, which is not case-sensitive.)

- In Assembly language, declare the symbols as described below:

```
ForeignProc PROC FAR           ; for a routine named ForeignProc in
...                               ; the foreign definition module.
ForeignProc ENDP
```

If you use the language qualifier "C" to indicate that the foreign language is C language, the compiler will assume that the routine symbol names are the very same as those declared in the foreign definition module, but that the data symbol names are the symbol names with a _ (underscore) at the very beginning of the name. Microsoft C, version 4.0 automatically adds the underscore at the beginning of data names. For routine names, the keyword pascal should be used to indicate to the C compiler to generate the symbol name only.

- Microsoft C, version 4.0, declares the routine as described below:

```
far pascal ForeignProc ( num )
int num;
{
...
}
```

Data symbol names are automatically added with an underscore with Microsoft C.

You can implement the foreign module in Modula-2 itself; the compiler assumes that this Modula-2 module will be called by another language with the conventions described above. If you already have software written in another language, you can use Modula-2 to develop additional, complementary code. The declaration of the routine should be as usual:

```
PROCEDURE ForeignProc ( num : INTEGER );
BEGIN
  ...
END ForeignProc;
```
6.3.1.2 Parameter Passing Convention

The parameters are passed with the *Pascal* convention: i.e., the parameters are pushed from left to right by the caller and the stack is cleared by the called routine before returning to the caller. Here are some examples:

- For the assembly language the parameters should be popped in the reverse order than they were pushed, and the stack should be cleared before the return. Please see the example below for interfacing with assembly language.

- When using *Version 4.0* of *Microsoft C*, the keyword `pascal` will indicate to the compiler to use the *Pascal* convention for passing the parameters.

- When using *LOGITECH Modula-2* to implement the foreign routine, the compiler will automatically use the *Pascal* convention for passing the parameters.

6.3.1.3 Default Data Segment When Using "C" Qualifier

To be compatible with *Version 4.0* of *Microsoft C*, the data segment should contain by default the segment of a group named DGROUP. Therefore before any call to a foreign module routine, the *LOGITECH Modula-2* compiler generates the following code sequence:

```assembly
MOV AX, SEG DGROUP
MOV DS, AX
CALLF ForeignProc
```
6.3.2 Examples of Foreign Definition Modules

In the examples below the foreign definition module is one of the two above. The differences are whether the main program is in *Modula-2* or in another language. The details are discussed below.

6.3.2.1 "main" in Modula-2, and "foreign module" in C

The foreign definition module is the one in Figure 6-6, above.

Here is the C implementation of ForeignProc:

```c
far pascal ForeignProc ( num )
int num;
int i;
printf("This is C Language\n");
for (i=0;i<num;i++)
    printf("Hello Folks ...\n");
```

The command for compiling this file is:

**MSC /AL FOREIGNM.C**

Here is the main program in *Modula-2*:

```modula2
MODULE MainMod;
FROM ForeignModule IMPORT ForeignProc;
FROM Terminal IMPORT WriteString, WriteLn;
BEGIN
    WriteString("This is Modula-2");WriteLn;
    ForeignProc(10);
    WriteString("This is Modula-2");WriteLn;
END MainMod.
```

The command for compiling this file is:

**M2C FOREIGNM.DEF MAINMOD.MOD**

The command for linking these files is:

**LINK MAINMOD.OBJ+FOREIGNM.OBJ/MAP**

The screen interaction you see while linking looks like this:

Run File [MainMod.EXE]:
List File [MainMod.MAP]:
Libraries [.LIB]:\M2LIB\LIB\MCRTS.LIB+\M2LIB\LIB\M2LIB.LIB
Cannot find library: M2USER.LIB
Enter new file spec:
To initialize C run time support, MCRTS.LIB should be used when linking. This library file does not ask for the entry point of the .EXE file. Therefore the entry point will be given to the C run time support which will, after its initialization, call _main — which is declared at the entry point of RTSMAIN.OBJ. After initialization of the Modula-2 RTS the main part of MainMod is called.

To run the program, simply type:

```
MAINMOD
```

6.3.2.2 "main" in C and "foreign module" in Modula-2

The foreign definition module is the one in Figure 6-6 above, with the "C" qualifier.

Here is the Modula-2 implementation of ForeignProc:

```
IMPLEMENTATION MODULE ForeignModule;
FROM Terminal IMPORT WriteString, WriteLn;
PROCEDURE ForeignProc ( num : INTEGER);
VAR i : INTEGER;
BEGIN
  WriteString ("This is Modula-2"); WriteLn;
  FOR i := 1 TO num DO
    WriteString ("Hello Folks ••• "); WriteLn
END
END ForeignProc;
END ForeignModule.
```

The command for compiling this file is:

```
M2C FOREIGNM.DEF FOREIGNM
```

Here is the main program in C.

```
far pascal ForeignProc ( int );

main ()
{
    printf("This is C Language\n");
    ForeignProc(10);
    printf("This is C Language\n");
}
```

The command for compiling this file is:

```
MSC /AL MAINC.C
```
Link these files with the following command:

```
LINK MAINC.OBJ+FOREIGNM.OBJ/MAP
```

You should see the following output:

- **Run File** [MainC.EXE]:
- **List File** [MainC.MAP]:
- **Libraries** [.LIB]: \M2LIB\LIB\CMRTS.LIB+\M2LIB\LIB\M2LIB.LIB
- **Cannot find library**: M2USER.LIB
- **Enter new file spec**:

To initialize **C run time support** and **Module-2 RTS**, use the CMRTS.LIB file when linking. CMRTS.LIB asks for the entry point of the EXE file. Therefore the **Module-2 RTS** is initialized first, and then calls `__astart`, which is the entry point for **C run time support**. After the **C run time support** is initialized, the call is done to `main`.

To execute the program just type:

```
MAINC
```

### 6.3.2.3 "main" in Modula-2 and "foreign module" in Assembly

The foreign definition module is shown in Figure 6-5 above, with no language qualifier.

Here is the **Assembly** implementation of `ForeignProc`:

```assembly
TITLE ForeignModule
PUBLIC ForeignProc

ForeignModule_TEXT SEGMENT BYTE PUBLIC 'CODE'
ForeignModule_TEXT ENDS
ForeignModule_DATA SEGMENT WORD PUBLIC 'FAR_DATA'
ForeignModule_DATA ENDS

ASSUME CS: ForeignModule_TEXT, DS: ForeignModule_DATA

ForeignModule_DATA SEGMENT
    str1 DB 'This is assembly language', OAH, ODH, 'S'
    str2 DB 'Hello Folks ...', OAH, ODH, 'S'
ForeignModule_DATA ENDS

ForeignModule_TEXT SEGMENT

ForeignProc PROC FAR
    PUSH BP
    MOV BP, SP
    SUB SP, 2
    MOV AX, SEG ForeignModule_DATA
    MOV DS, AX
    MOV DX, OFFSET ForeignModule_DATA:str1
    MOV AH, 9
    INT 021H
    ; writes str1 to screen

121
Chapter 6

forblock: MOV DX, OFFSET ForeignModule_DATA: str2
MOV AH, 9
INT 021H
INC WORD PTR [BP-2]

forloop: MOV AX, [BP+6]
CMP [BP-2], AX
jl forblock

MOV SP, BP
POP BP
RET 2

ForeignProc ENDP

ForeignModule_TEXT ENDS
END

The main program in Modula-2 is the same as that in Section 6.3.2.1.

The command for compiling this file should be:
M2C FOREIGNM.DEF MAINMOD.MOD

The command for compiling this file should be:
MASM FOREIGNM.ASM

The commands for compiling and linking these files should be:
LINK MAINMOD.OBJ+FOREIGNM.OBJ/MAP

You should see the following output:

Run File [MainMod.EXE]:
List File [MainMod.MAP]:
Libraries [.LIB]:\M2LIB\LIB\M2RTS.LIB+\M2LIB\LIB\M2LIB.LIB
Cannot find library: M2USER.LIB
Enter new file spec:

Warning with Foreign Modules

When using foreign definition modules, take low level considerations of different compilers into account. Some run time supports reroute some system calls and trap some interrupts. Be aware of the potential for this kind of problem.

LOGITECH Modula-2 utilities are tailored to work with Modula-2 code generated by the LOGITECH Modula-2 Compiler, and may not work perfectly with code generated by other compilers.
Chapter 7
The Symbolic Post-Mortem Debugger

The LOGITECH Symbolic Post-Mortem Debugger, known as the LOGITECH PMD, or simply the PMD, lets you inspect a crashed program to find out what went wrong with the program and where the problem occurred.

LOGITECH Modula-2 works with two complementary debuggers: the Symbolic Post-Mortem Debugger (referred to as the LOGITECH PMD, or simply the PMD) is described in this chapter. The Symbolic Run-Time Debugger which is called the LOGITECH RTD, or simply the RTD, is part of the LOGITECH Modula-2 Toolkit and is described there.

The PMD lets you symbolically analyze a program that terminated abnormally. Thus, you can determine what went wrong, and where the problem occurred. To do this, a memory dump must have been created and written onto a file by LOGITECH Modula-2 Run-Time Support. This memory dump, together with the .REF reference files generated by the compiler, are taken by the PMD, and you are given symbolic information on the state of the program: the procedure call chain with the procedure names, the modules they come from, and the values of all data according to their types and structures.
A memory dump is created only if the module DebugPMD has been imported into at least one of the program or sub-program (overlay) modules. The memory dump is written into the file <filename>.PMD, where <filename> is the name of the program or overlay file that contains the process that has terminated abnormally. If a program or overlay file terminates abnormally, without the DebugPMD module linked to it, the program is merely terminated with an appropriate message.

If a program terminates abnormally and unpredictably, insert calls to the DebugPMD library module and recompile your source code. Then, after linking, run the program again. Then, when your program crashes, the memory image is saved on disk in a dump file. The PMD lets you inspect this dump file symbolically: it displays the state of the program, using the corresponding module and procedure names. It also shows the names and values of variables according to their type structure.

LOGITECH Modula-2 creates <program_name>.PMD, a memory dump file, when:

- A run-time error occurs.
- A program calls the standard procedure HALT.
- A program calls the Terminate procedure, exported by the RTSMain module, with a parameter other than Normal or Warning.
- \[Ctrl]+[Break] or \[Ctrl]+[C] is pressed while a program that imports the Break module is running.

The structure and user interface of the RTD are the same as that of the PMD. The RTD uses the same windows and screen layout as the PMD. The RTD commands are a superset of the PMD commands. All commands of the PMD are also valid in the RTD.
7.1 LOGITECH PMD Files

The following files are on the diskette:

MDA.CFG
CGA.CFG
EGA.CFG
DB.CFG
DB.HLP
PMD.EXE

The file extensions stand for:

.CFG  Configuration file
.HLP  Help file
.EXE  Executable file

7.2 The PMD and Your Hardware

The LOGITECH PMD works on IBM PC and compatible computers. We recommend that you run the PMD with a mouse, although it is possible to work without one. If a mouse is used, it is important to have recent mouse drivers: LOGITECH, Release 3.20 or higher, or Microsoft, Release 6.00 or higher.

The PMD requires approximately 260K bytes of memory, not counting DOS or any other drivers you may have installed.

The PMD can work with a MDA, CGA or EGA video controller. With a CGA and EGA the windows can have colors. With an EGA controller, the PMD can have a screen with 43 lines.
7.3 How to Run the Post-Mortem Debugger

To run the LOGITECH PMD, type:

\texttt{PMD \ <dump\_file\_name\_if\_known> \ <\text{command line parameter}>}

The debugger responds with a sign-on message:

\texttt{LOGITECH MODULA-2 Post-Mortem Debugger}

followed by the version number and a copyright notice.

If you have not specified a dump file name, you will see the following screen. If you specified the name of the dump file on the command line you will not see the prompt.

![Sample Screen](image)

**Figure 7-1 Sample Screen**

If you did not enter the dump file name on the command line, enter it here.
When the **PMD** is loaded, it executes some internal initialization, and then displays appropriate information in each window on the screen.

---

**NOTE**

If you use the dump file name on the command line specify it as an argument just after the debugger.

The **PMD** from *LOGITECH Modula-2, Version 3.0* uses the .MAP file of the program.

---

**IMPORTANT !!!**

!!! NO DEBUGGING CAN BE DONE WITHOUT A .MAP FILE !!!

---

The application program must be compiled with the SYMBOL option (which is set by default) and the link must be made with a .MAP file (default of the linker).
7.4 PMD Configuration

The PMD reads certain files during its initialization phase. One contains the layout of the screen, and one has information for the help window.

7.4.1 Screen configuration

The screen configuration is in the file DB.CFG. This file is a binary file which cannot be edited. If the file is not found, the debugger prompts for it. If you press \texttt{Esc}, you get the default setting for the screen. You can then modify the setting and either save it with the \texttt{save config} command, or be queried when you leave the debugger.

On the distribution disk, four screen configuration files are provided:

- **MDA.CFG**: Monochrome. Fits all controllers. MDA.CFG is used when the configuration file is not found and you press \texttt{Esc}.
- **CGA.CFG**: Color. Works with CGA controller or with EGA in CGA mode.
- **EGA.CFG**: EGA. Works in 43 lines mode
- **DB.CFG**: Same as MDA.CFG.

If the computer has an exotic display (e.g. Olivetti, ATT, or COMPAQ), start with MDA configuration (MDA is less critical).

7.4.2 On-line Help

On-line help is in a text file named DB.HLP. If DB.HLP is not found, you are not prompted for it.

---

Search strategy is the same for both the PMD and RTD: the debugger first looks into the current directory and then in the directory from where it was loaded.
7.5 Post-Mortem Debugger Options

When you start the `PMD`, you may also specify various options on the command line. Options are denoted by a `/` (forward slash), followed by the first character of the option name. For example, to activate the Query option, enter:

```
PMD/Q
```

when starting the `PMD`.

```
/Q
```

(default : `/Q-`)

**Query**

Tells the `PMD` to search for reference and source files according to the query search strategy. You will be prompted to enter the reference and source file names. If the Query option is not specified, the `PMD` automatically searches for these files according to the default search strategy.
7.6 User interface

7.6.1 Windows

The *PMD* uses windows for optimal viewing of executed code.

A window can be *open* or it can be an *icon*. Open windows show their contents. Icons appear as labels on the last line of the screen. Throughout this chapter, we will often referred to a "window", meaning an open window, or to an "icon", which means a window label.

These windows cannot be overlapped and they always share the entire screen. A window is always displayed beside another window. For example, if the screen is divided vertically in two windows, a third window can be opened only at the border of one of the already opened windows.

There is always one active window. Therefore, if a menu is called it will referenc the functions in the active window. It is also possible to activate an icon so that its menu is available. The activation of the icon does not open it as a window.

The menus and the messages are displayed with pop-up windows.
Window functions are:

- Activate a window
- Scroll through the contents

- Change color screen colors:
  - Borders
  - Window Contents
  - Menus

- Change Size and Position:
  - Move window borders (because they share the screen, the motion of a border modifies the size of adjacent windows)
    - Fill the whole screen (zoom window)
    - Shrink to become an icon
    - Swap position and/or size with another window. Can be swap an icon and a window, but not two icons.

All commands can be done with the mouse and/or with the keyboard. With the mouse, use the double click which calls the most probable command, or use the menu. With the keyboard, use the menu or the short cuts. If the mouse is not connected, the mouse cursor is not displayed.
7.6.2 Mouse Functions

The mouse button is context sensitive. The table below describes these meanings.

<table>
<thead>
<tr>
<th>Cursor position</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Left Window Border</td>
<td>Scroll Up</td>
<td>Select Absolute Vertical Position</td>
<td>Scroll Down</td>
</tr>
<tr>
<td>Bottom Window Border</td>
<td>Scroll Left</td>
<td>Select Absolute Horizontal Position</td>
<td>Scroll Right</td>
</tr>
<tr>
<td>Bottom Left Corner of Window</td>
<td>Move</td>
<td>Left Bottom Border</td>
<td></td>
</tr>
<tr>
<td>Inside Window</td>
<td>One Click: Select</td>
<td>Call Menu To Manipulate Window</td>
<td>Call Menu For Specific Window Actions</td>
</tr>
<tr>
<td></td>
<td>Double Click: Carry Out Most Probable Action On The Selection</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prompt</td>
<td>Terminate User Entry</td>
<td>Escape from Prompt</td>
<td>Escape from Prompt</td>
</tr>
<tr>
<td>Menu</td>
<td>Execute Highlighted Action</td>
<td>Execute Highlighted Action</td>
<td>Execute Highlighted Action</td>
</tr>
</tbody>
</table>

Figure 7-2 Mouse Function Table
Note:

Scroll functions: Similar to those in the POINT Editor or Microsoft Word. Attesting to scroll beyond the ends causes a beep.

If two-button mouse is used, use (••) for (0 • 0).

Click: (■■■) anywhere inside the window or on an icon you wish to select. (■■■) doesn’t expand the icon, but gives you its local menu.

Move window borders: Select the lower left window corner (the other part of the border is used as scroll bar). If there is ambiguity, the PMD prompts you for a menu.

eXchange windows: Select one window. Then point at the other window with the mouse and select eXchange in that window’s menu. This moves the active window into the new window position.

The other window commands are available via the menu.
7.6.3 Keyboard Functions

7.6.3.1 How to scroll

The active window can be scrolled horizontally and vertically. The cursor keypad is mapped as follows for scrolling:

```
+-------+-------+-------+
|       | HOME   | ^     | PG UP  | ---- 1 page up
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1 line up</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>------</td>
<td>--------</td>
</tr>
<tr>
<td>beginning of text</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>------</td>
<td>--------</td>
</tr>
</tbody>
</table>
+-------+-------+-------+
|       | <      | >    | ---- 1 column to the right to the left
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1 column to the left</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>------</td>
<td>--------</td>
</tr>
</tbody>
</table>
+-------+-------+-------+
|       | END    | V    | PG DN  | ---- 1 page down
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>end of text</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>------</td>
<td>--------</td>
</tr>
</tbody>
</table>
+-------+-------+-------+
|       |        |      |        |
| 1 line down                 |
|-------|--------|------|--------|
+-------+-------+-------+
```

Figure 7-3 Cursor Key Scrolling

7.6.3.2 Select a window object

To select a window object, move cursor above the object and press `Spacebar` or `.]`. To activate a window, use a window activation command.
7.6.3.3 Call the menu

- **F10**
  - Displays or erases the menu.

- **↓**
  - Validates the selected item in the menu.

- **Spacebar**
  - Leaves the menu without performing the action.

**Menu bar**
- Menu bar can be moved up or down. The cursor keypad is mapped as follows for bar moving:

```
+-------+-------+-------+
1 1 1 1
first column
PG UP
+-------+-------+-------+
1 1 1 1
```

**Figure 7-4 Select The Menu Bar**

However, it is faster to use keystroke commands. The menu beside the corresponding item displays the appropriate keystroke sequence.

- * means **Alt**
- ^ means **Ctrl**

The musical note sign means **double click**.

Off-menu, **Esc** purges the keyboard input buffer.

Wrong keystrokes are beeped, if beep is ON.

**↓** and **Spacebar** activate the selected command.
Chapter 7

7.6.3.4 Respond to a prompt

| -   | Validates the characters entered to the debugger. |
|     | or Spacebar                                     |
| Esc | Aborts the input processing.                    |
| Ctrl-X | Erases all the input characters.               |
| ← Back | Erases the last character input.               |

**NOTE**

- Boxed characters act as `-`.
- Boxed characters and Boxed characters act as `Esc`.

7.6.3.5 Move the mouse cursor with the keyboard

These functions are for the numeric keypad when a mouse is connected to the computer. You can move the mouse cursor by using `†Shift` with the keypad. The cursor keypad is mapped as follows:

```
+-------+-------+-------+
1 1 1 1
--- 1
1 line up
+-------+-------+-------+
few columns to the left  ---  HOME  ^  PG UP  ---- few lines up
| |
1 column to the left  ------  <  |  >  |  ---- 1 column to the right
| |
few columns to the right  ---  END  V  PG DN  ---- few lines down
| |
+-------+-------+-------+
1 line down
```

*Figure 7-5  Keyboard and Mouse*
7.7 Windows and Commands

The LOGITECH PMD displays these windows:

<table>
<thead>
<tr>
<th>Type</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call:</td>
<td>Show the calling chain of your program at crash time</td>
</tr>
<tr>
<td>Module:</td>
<td>Show list of the modules in memory</td>
</tr>
<tr>
<td>Text:</td>
<td>Modula-2 source file</td>
</tr>
<tr>
<td>Data:</td>
<td>Data defined in a module or procedure</td>
</tr>
<tr>
<td>Raw:</td>
<td>Direct access to the memory</td>
</tr>
<tr>
<td>Help:</td>
<td>Displays the contents of DB.HLP (help file)</td>
</tr>
<tr>
<td>Message:</td>
<td>Displays messages of the PMD (e.g., which file is accessed)</td>
</tr>
</tbody>
</table>

The PMD has two types of commands - global and local. Local commands are only applicable to the particular window in which they appear and are shown.

Quit command

This command lets you quit the PMD. If the screen layout was changed during the session and not saved, you can save the configuration at that time.
Window activation commands

You can switch from one window to another one by typing:

<table>
<thead>
<tr>
<th>Keystrokes</th>
<th>Window Selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>~@</td>
<td>Call</td>
</tr>
<tr>
<td>~C</td>
<td>Module</td>
</tr>
<tr>
<td>~D</td>
<td>Data</td>
</tr>
<tr>
<td>~T</td>
<td>Text</td>
</tr>
<tr>
<td>~R</td>
<td>Raw</td>
</tr>
<tr>
<td>~H</td>
<td>Help</td>
</tr>
<tr>
<td>~S</td>
<td>meSsage</td>
</tr>
</tbody>
</table>

You can also click into the window or go through menu. When you activate an icon, it does not open it lets you use its menu. Open an icon with a window manipulation command.

F1 opens the help window (full screen). F1 exits the help window.
Window manipulation commands

You can move the window borders. With a mouse, the border can be selected by picking the lower left corner of a window (the other part of the border is used as scroll bar). If an ambiguity exists, the PMD prompts you for a menu to select. If no mouse is used, use Alt-M and the PMD prompts with a menu for the window to modify.

Alt-Z (Zoom) toggles a window between full and partial screen.

A window can be open or it can be an icon. A window is iconized (or shrunk Alt-S) when only its label is visible on the last line of the screen. You can open an icon with the vertical or the horizontal expand, Alt-V or Alt-H.

Alt-X, lets you exchange two windows. This command moves the active window in the selected window. Without a mouse, a menu lets you select the target window.

Configuration commands:

Alt-T Bring up a menu to change screen colors:

Alt-B, Alt-G Border colors.
Alt-L, Alt-O Menu colors.
Alt-C, Alt-D Window content colors.

Alt-N Turns a bell on or off.

Alt-F Lets you save the configuration. If you do not save after a change, you will be prompted when you will leave the debugger.

Low level commands:

Alt-R Redraws all windows.
Alt-P Centers the current selection.

Menu commands:

F10 Invokes and exits the menu (toggle).
7.7.1 Call Window

The Call window displays the chain of procedure calls of the crashed process.

Local commands in the Call window:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>Gives the address and line number of the executed statement.</td>
</tr>
<tr>
<td>Examine break process</td>
<td>Updates all windows with the information related to the process running when the program stopped.</td>
</tr>
<tr>
<td>Data</td>
<td>Updates the data window with the data of the selected element (PROCEDURE or PROCESS).</td>
</tr>
<tr>
<td>Text</td>
<td>Updates the text window with the text of the selected element (PROCEDURE or PROCESS).</td>
</tr>
<tr>
<td>Both</td>
<td>Executes Data and Text commands or is equivalent to the double click on the selected item.</td>
</tr>
</tbody>
</table>

**NOTE**

You can see the contents of the process only if RTSMAIN.REF is available.
The following screen shows the default setup of the windows.

![Sample Screen](image)

**Figure 7-6** Sample Screen
7.7.2 Module Window

The Module window displays the list of modules of the program being debugged.

Local Commands in the Module Window

**Find:** Lets you search for a module. Wildcard * and ? characters use the same syntax as DOS. To select the next module name matching the input pattern use Find again and press ↵.

**Address:** Gives data and code addresses of the module, and updates the raw window.

**Data:** Updates the data window with data of the selected element.

**Text:** Updates the text window with the text of the selected element.

**Both:** Executes Data and Text commands or is equivalent to the double click on the selected item.
7.7.3 Data Window

Data window displays variables and/or parameters of the selected procedure or module.

Local Commands in the Data Window

Son and Father: Displays the data structure beneath the current level for the selected item. If the selected item is an array, the Son command displays the values of the elements of the array. If the selected variable is a record, the Son command displays the names and values of the record fields. Likewise, local modules are shown as data of the embedding module. You can also examine the content of the process descriptor by entering the Son command when a variable of type "PROCESS" is selected. In addition, this command can be used to follow linked lists when you select a variable which is a pointer or is of type "ADDRESS". The double click applies these functions. The command Son is applied when an element is "double clicked". The command Father is applied when the path on the top of the window is "double-clicked". The command Son can be used on a variable of type PROCESS only if the file RTSMAIN.REF is available.

eXchange: Lets you switch from procedure local data to module global data and vice versa.

Right/Left: These commands are only applicable when the selected data item is an element of an array, or part of an element of an array. The Right and Left commands select the element with the next higher or lower index in the array. The current level is not changed by these commands. If the array elements are records, the record field selected is not affected.

Index: Lets you select randomly an element of an array by giving the value of its index.
**Chapter 7**

**Type transfer:**
Lets you change the type of a displayed variable. You can use a predefined or user-defined type. If no type is given, the variable is displayed with its original type. The Type transfer is allowed only if the type of the variable and the new type are of the same size. If you use a type of your own, the debugger prompts you for the module defining it. A type-changed variable is marked by a "T".

**Variable:**
Returns to the first level of the selected procedure or module. The first level shows the variables of the procedure or module. The Variables command can be used after you have repeatedly entered the Son command and wish to return to the first level directly, without repeatedly entering the Father command.

**Examine PROCESS:**
This command can be used when you select a variable of type "PROCESS". Otherwise, the PMD prompts you to introduce the address of the process descriptor - the content of a variable of type PROCESS. The Examine command displays the call chain of the process to be examined. Enter the call window command Examine break process to show the Call window of the process that was running when the program stopped. Checks to see if the process is initialized (a word with a special pattern is in all process descriptor).

**Address:**
Displays the address of the selected data item and updates the Raw window.
The Symbolic Post Mortem Debugger

The following sample screens show the path you follow to modify the content of an array element with a record structure. First, invoke the Son command to view the elements of the variable "node" of the module "Demo" (Figures 7-7 and 7-8. Again, invoke the Son command to display the fields of the record "node[1]", and the value and type of each field (Figures 7-9).

```
PROCEDURE RecursiveOne (x: CARDINAL; y:REAL; z: INTEGER
BEGIN
  WITH node[x] Do
    data1 := x;
    data2 := y;
    data3 := z;
  END; (* WITH *)
  INC(x);
  y:=y+1.0;
END; (* WITH *)
```

Figure 7-7 Sample Screen
Chapter 7

Figure 7-8 Sample Screen

Figure 7-9 Sample Screen
The Symbolic Post Mortem Debugger

Figure 7-10 Sample Screen

Figure 7-11 Sample Screen
7.7.4 Text Window

The Text window displays the text of a module or procedure. The (>) greater-than sign indicates the line in which the program, the next procedure, or where the last process transfer or interrupt occurred.

Local Commands in the Text Window

Find: Prompts for a PROCEDURE or local module name (case sensitive), or a line number and, if found, the PMD sets the selected position to this PROCEDURE, module, or line number. This command allows you to enter either a line number or a procedure name. Wild characters are accepted (" * ", " ? ").

eXchange: Lets you switch from MOD to DEF or DEF to MOD.

Address: Show the code address of the selected source line and updates the Raw window.
7.7.5 Raw Window

The Raw window displays the memory contents around a given address. The initial address of the selected memory location depends on the window from which you invoke the Raw window. The values are set the same way as in the PMD.

Local Commands in the Raw Window

Address: Lets you enter the address of data to be displayed.

Son: Takes the contents of the selected memory location as the new selected address. You typically enter this command to follow a linked list. (dereferencing)

Examine PROCESS: Assumes the memory contents at the selected address is of type "PROCESS" that is a pointer to a process descriptor. The Examine command displays the Call window of the process. The call window command Examine break process can be used to show the Call window of the process that was running when the program crashed. It also checks the check word of the process descriptor to check if a valid process is selected.

Hexadecimal: Decimal to hexadecimal conversion.

Decimal: Hexadecimal to decimal conversion.

Various display modes: address, byte, cardinal, char, integer, longint, real, text, word

#Address: ADDRESS (hexadecimal) format.

#Byte: BYTE (hexadecimal) format.

#Unsigned: Unsigned CARDINAL format

#Char: CHAR (octal) format. Non-printable characters are displayed as octal numbers.

#Integer: INTEGER.

#Text: TEXT. Non-printable characters are displayed as IBM PC extended characters.

#Real: REAL.

#Word: WORD (hexadecimal) format. (default)
7.7.6 Message Window

Version: Lets you display the version of the debugger.

7.7.7 Markers

> (greater-than) is used in the PMD as an execution marker to indicate active code when
the program crashed. It appears in the Call, Module and Text windows.

7.7.8 Selecting an Item for Display

The PMD displays the position of the selected item highlighting the proper line. You
may select a different item using the cursor keys or the mouse.

7.7.9 Relation between Windows

The PMD displays different windows at the same time. This impacts what is shown and
how selections are made. The Call and Module windows are mostly used for selecting
text and/or data. You can select an element and use the double click (or the menu) to
update the other windows.

7.7.9.1 Update made from the Call window

When a windows update is requested from the Call window, the PMD shows the data
and/or the text of the selected procedure. The Raw window shows the contents of the
stack.

7.7.9.2 Update made from the Module window

When a windows update is requested from the Module window, the PMD displays the
data and/or the text of the current module. The Raw window shows the global data area.

7.7.9.3 Update from the Data and Text windows

You can modify the contents of the Raw window by using the command Address of the
Data or the Text window.
The Symbolic Post Mortem Debugger

7.8 Consistency Checks

The PMD does three consistency checks:

- Between the code in memory (.PMD file or .EXE file) and the .MAP file. This check is made by using keys stored in the code and referenced by a $OK label in the map.

- Between the code in memory and the .REF file. This test is made by using the keys stored in the code and a key stored in the .REF.

- Between the .REF file and the .MOD file. This check is made by using the date of the .MOD (i.e. the date of the source file when it was compiled) stored in the .REF file. If this date is not the same as the date of the .MOD file read by the debugger, an inconsistency is signaled.

The inconsistency between the .MAP and the code in memory is very dangerous. It is extremely probable that all the information known from the .MAP is wrong. For this reason, the debugger does not display any symbolic information. It is strongly advised to relink your application.

An inconsistency between the .REF and the code in memory will make trouble only for the corresponding module. Depending on the changes made, only the data can be wrong or only the position displayed in the Text window or both can be wrong. It is advised to recompile this module and to relink the application (with a .MAP !)

An inconsistency between the .MOD and the .REF will make trouble when displaying the statement executed in the Text window. It is also advised to recompile this module and to relink the application.
NOTE

An additional check is made when a process descriptor or when an overlay descriptor is accessed (an overlay descriptor is also allocated for the main program). A field is initialized in both descriptors with a specific value. If this value is not found, an error is signaled. This error means that you tried to analyze an incorrect part of the memory or that the memory was destroyed.

A test is also made when a .EXE is loaded to check if it is a Modula-2 program. The debugger can debug programs only if the main is in Modula-2.
7.9 Messages

Already as an icon
Already at top level
ASSERT: message
  An internal error is detected into the debugger. The execution stops.

Beginning of this ARRAY

Call list incomplete (BP chain invalid)
  A problem was encountered while reading the stack (memory destroyed?)

Call list too long (>32)
  This is a warning message which tells you that not all procedures on
  the stack are visible in the call list.

Can't be iconized
Can't expand in an icon
Cmd not allowed (use zoom)
Cmd not allowed on a window
Cmd not allowed on an icon
Cmd not valid in DEF MODULE
Color changed
Config changes NOT saved
  You did not save the last screen configuration.

DEF file not found
End of this ARRAY
Help file not found
Chapter 7

Incorrect MAP file
An inconsistency is detected between the .MAP file and the code. This message appears and remains in the windows to warn you.

Incorrect REF file
An inconsistency was detected with the .REF file. A pop up window displays the reason the first time the inconsistency is detected. This message remains in the window to remind you.

Invalid call list
This message is displayed in a pop-up window when a problem is encountered while reading the call list (memory destroyed ?)

Invalid descriptor
The overlay descriptor has no valid check word (memory destroyed ?)

Invalid PROCESS descriptor
The process used as a parameter of the last command has no valid check word (memory destroyed ?)

Invalid process
The dump cannot be analyzed, the process descriptor of the process which crashed is invalid (memory destroyed ?)

MOD file not found

MODULE not found in list

No call list

No data (unknown PROCEDURE)

No data in this element

No data in this local MODULE

No data in this PROCEDURE

No global data in this MODULE

No global or local data

No selected PROCEDURE

No text associated

No text for a PROCESS
The Symbolic Post Mortem Debugger

Not enough memory
    Memory problem. Use Small or Big swap option.

PROC/MODULE isn’t in this text

REF file can’t be re-opened

REF file not found

Swapping of icons not allowed

This border can’t be moved

This data can’t be modified
    [Data window, Modify command]
    This data cannot be modified. Usually this is for hidden types. In this case you should use the Son command to see the effective type and then you can modify the variable.

This data isn’t an ARRAY

This data isn’t structured

This data is of its original TYPE

To modify data use Son cmd

Too many MODULEs (> 256)
    The debugger cannot debug programs with more than 256 modules.

Too small for expanding

TYPE not found in given MODULE

TYPE sizes differ

Wrong version of REF file: Bad structure
    The REF file does not have the correct version (recompile the module and relink the application) or is too big.

Wrong version of REF file: Different from OBJ
    An inconsistency was detected between the .REF and the code in memory.

Wrong version of text file
    An inconsistency was detected between the .MOD and the .REF file.
Out of dump
Out of dump : = NIL
Out of dump : > 1 MB
Chapter 8
Implementation Features

WARNING

Use the features in this chapter with care, since they can conflict with the basic software of the operating system, and with the LOGITECH Modula-2 system.
Chapter 8

8.1 System Dependent Facilities

This section gives an overview of the LOGITECH Modula-2 specific low-level features. Section 8.2, Priorities and Interrupts gives additional information on hardware dependencies.

The differences in programming for various implementations can be attributed to:

- Changes to the language itself.
- Differences in the set of available procedures and data types which reflect the structure of the machine used.
- Differences in the internal representation of data.
- Differences in the set of available modules, in particular those for handling files and peripheral devices.

The last item reflects the environmental aspects of Modula-2 — such as the set of standard library modules that give access to the file system, the keyboard and the screen.

A listing of the .DEF files for the LOGITECH Modula-2 library is in Section 9.2.
8.1.1 Language Extensions

Constants of type ADDRESS may be declared as <segment:offset>, where the segment and offset are CARDINAL numbers. The segment and offset must be constant numbers.

Example:

```modula2
CONST int3Addr = OH:12H;

TYPE ScreenType = 
  ARRAY [0..24] OF (* rows *)
  ARRAY [0..79] OF (* columns *)
  RECORD char, attr: CHAR END; (* content *)

VAR screen [0B000H:0H] : ScreenType;

a := 1234:5678; (* assume 'a' of type ADDRESS *)
```

LOGITECH Modula-2 also provides for the declaration of absolute variables. Absolute variables are variables for which the programmer, rather than the LOGITECH Modula-2 compiler, defines the memory address at which the variable will be located. This feature is intended to be used for memory-mapped input and output.

When declaring an absolute variable, the identifier denoting it must be followed by an address constant in brackets. The address constant defines the absolute address of the variable in memory. The variable `screen` in the above example is declared as an absolute variable.

8.1.2 Address Arithmetic

In Modula-2, the standard module SYSTEM provides the type ADDRESS. The use of the type ADDRESS and the operations on objects of type ADDRESS, must be considered non-portable. The implementation of ADDRESS operations is very dependent on the architecture of the target system. The structure of a computer may restrict the operations that are possible on objects of type ADDRESS.
8.1.2.1 Interpretation of Objects of Type ADDRESS

Objects of type ADDRESS denote a particular location in memory. Type ADDRESS is compatible with any pointer type. Objects of type ADDRESS can be used as if there were two different type definitions for type ADDRESS:

\[
\text{TYPE ADDRESS} = \text{POINTER TO WORD;}
\]

\[
\text{TYPE ADDRESS} = \text{RECORD}
\]
\[
\begin{align*}
\text{OFFSET} & : \text{CARDINAL;} \\
\text{SEGMENT} & : \text{CARDINAL;} \\
\text{END;}
\end{align*}
\]

Example:

If we assume the declarations:

\[
\text{VAR}
\]
\[
\begin{align*}
a & : \text{ADDRESS;} \\
w & : \text{WORD;} \\
\text{off, seg} & : \text{CARDINAL;}
\end{align*}
\]

then the following statements are legal:

\[
\begin{align*}
a^ \oplus & := w; \ w := a^ \oplus; \\
a.\text{OFFSET} & := \text{off}; \ \text{off} := a.\text{OFFSET}; \\
a.\text{SEGMENT} & := \text{seg}; \ \text{seg} := a.\text{SEGMENT} \\
\text{WITH a DO} \ \text{SEGMENT} & := \text{seg END;}
\end{align*}
\]

8.1.2.2 Operations Involving Objects of Type ADDRESS

A restricted set of arithmetic operations on objects of type ADDRESS is possible. The switch and compiler option T determines whether or not test code is generated for ADDRESS operations.
Addition and Subtraction

Addition and subtraction are allowed in expressions of type ADDRESS. An ADDRESS expression contains exactly one operand of type ADDRESS. All other operands must be of type CARDINAL. The operation is only performed with the OFFSET value of the ADDRESS - the SEGMENT value is never modified. If test code is on, the run-time error \textit{address overflow} will occur upon an overflow of the OFFSET value on an addition or subtraction operation.

The standard procedures \textsc{INC} and \textsc{DEC} can also be used with variables of type ADDRESS. The following declarations are assumed:

\begin{verbatim}
PROCEDURE INC (VAR a:ADDRESS; k:CARDINAL);
PROCEDURE DEC (VAR a:ADDRESS; k:CARDINAL);
\end{verbatim}

\begin{center}
\textbf{NOTE}
\end{center}

The second parameter must be assignment compatible with type CARDINAL. If test code is on, negative values will generate a run-time error. Using \textsc{INC} and \textsc{DEC} is the preferred way to do ADDRESS arithmetic.

The following list shows the kinds of expressions involving operands of type ADDRESS which are valid. Each operand itself may be an expression of the corresponding type.

\begin{center}
\begin{tabular}{|c|c|c|c|}
\hline
\textbf{Operation} & \textbf{1st Operand} & \textbf{2nd Operand} & \textbf{Result Type} \\
\hline
Addition & ADDRESS & CARDINAL & ADDRESS \\
Addition & CARDINAL & ADDRESS & ADDRESS \\
Subtraction & ADDRESS & CARDINAL & ADDRESS \\
\hline
\end{tabular}
\end{center}
Chapter 8

Multiplication and Division

Multiplication and division operations are not allowed with operands of type ADDRESS. This includes:

\[
\begin{array}{c}
* \\
/ \\
MOD \\
DIV
\end{array}
\]

Comparison

The comparison of two operands or expressions of type ADDRESS is allowed. Here are the restrictions and how the comparison is implemented:

equal
not-equal

A check on (non-) identity is generated:

\[ a1 = a2 \iff (a1.\text{SEGMENT} = a2.\text{SEGMENT}) \land (a1.\text{OFFSET} = a2.\text{OFFSET}) \]

greater-than
greater-equal
less-than
less-equal

These are allowed only between addresses within the same segment — if the SEGMENT values of both operands are identical. They compare the OFFSET values only.

The result of these operations is undefined if the two ADDRESS operands compared have different SEGMENT values.

If test code is ON, the run-time error address overflow will occur when the SEGMENT values are not equal.
8.1.2.3 Dereferencing Pointers

The switch and compiler option "T" determines whether or not test code is generated for accessing data through pointers.

If test code is on, any pointer with an offset equal to $0xFFFF$ is considered to be NIL. Therefore, any access through such a pointer is illegal and will result in a run-time error.

The predefined constant NIL, which is compatible with all pointer types, has the internal representation $0H:0FFFFH$. It is strongly recommended that no program makes use of this information. The representation of NIL is implementation dependent and subject to change without notice.
8.1.3 The Module SYSTEM

The module SYSTEM offers additional facilities to programs written in the Modula-2 language. Most of them are dependent upon the implementation or are specific to the target processor. Module SYSTEM also contains types and procedures which allow very basic coroutine handling.

Module SYSTEM is directly known to the compiler because its exported objects obey special rules that must be checked by the compiler. If a compilation unit imports objects from module SYSTEM, no symbol file need be supplied for this module. However, the declaration of these objects in the import list is required.

Furthermore, no link file exists for this module. The implementation of the pseudo module SYSTEM is realized by inline code or by calls to the LOGITECH Modula-2 runtime support, generated by the compiler.

The interface of the pseudo module SYSTEM cannot be described completely with a regular Modula-2 definition module. Module SYSTEM offers some features which expand the language itself. However, for easy reference, a description of module SYSTEM in a form similar to a definition module has been included in the library section of this manual.

For additional information please refer to Section 12 of Report on the Programming Language in Programming in Modula-2.
8.1.3.1 Constants Exported from Module SYSTEM

AX, BX, CX, DX, SI, DI, ES, DS, CS, SS, SP, BP

These constants denote the processor’s registers. They are defined for use with the procedures GETREG and SETREG which are also provided by module SYSTEM.

8.1.3.2 Types Exported from Module SYSTEM

BYTE

An individually accessible storage unit (one byte). No operations except assignments and type conversions are allowed for variables of type BYTE. An actual parameter of any type that uses one byte of storage may be passed to a formal BYTE parameter. For convenience, small CARDINAL constants ( ≤ 255 ) are also allowed as parameters.

WORD

One word of memory (two bytes). No operations except assignments and type conversions are allowed for variables of type WORD. An actual parameter of any type that uses one word of storage may be passed to a formal WORD parameter.

PROCESS

A type used for process handling.

ADDRESS

The address of any location in storage. The type ADDRESS is compatible with all pointer types and is itself defined as POINTER TO WORD. Section 8.1.2, Address Arithmetic, explains more on the properties and the use of type ADDRESS.
8.1.3.3 Functions Exported from Module SYSTEM

ADR(variable): ADDRESS

Storage address of the parameter variable.

SIZE(variable): CARDINAL

Returns the number of bytes used in storage by the parameter variable. If the variable is of type RECORD with variants, then a variant of maximal size is assumed.

TSIZE(type): CARDINAL

Yields the number of bytes used in storage by a variable of the substituted type. If the type is a record with variants, then tag constants of the last FieldList (see syntax in Programming in Modula-2) may be substituted in their nesting order. If some or all tag constants are omitted, then the remaining variant with maximal size is assumed.
8.1.3.4 Procedures Exported from Module SYSTEM

NEWPROCESS (processBody : PROC;
    workspaceAddress : ADDRESS;
    workspaceSize : CARDINAL;
    VAR process : PROCESS)

Create a new process.

processBody is the procedure to execute.
workspaceAddress is the address of the data area for the process (the workspace).
workspaceSize is the size of the workspace in bytes.

The variable process receives the created PROCESS object. Allow 400 bytes for system overhead in each workspace.

NOTE

If the workspace of the new process is too small and does not allow a reasonable initialization, the process that calls NEWPROCESS is terminated with a stack overflow.

TRANSFER (VAR fromProcess, toProcess : PROCESS)

Save the current process state in fromProcess, and resume the execution of the process in toProcess.

IOTRANSFER (VAR interruptHandler: PROCESS;
    interruptedProcess: PROCESS;
    interruptVectorNumber: CARDINAL)

Save the current process state in interruptHandler, and resume the execution of the process in interruptedProcess. The occurrence of the designated interrupt has the effect of TRANSFER (interruptedProcess, interruptHandler).
LISTEN

Temporarily lower the priority of the calling process and allow pending interrupts to come through.

GETREG (register: CARDINAL; VAR value: BYTEorWORD)
SETREG (register: CARDINAL; value: BYTEorWORD);

These two procedures are used to set and to retrieve the contents of machine registers. They generate in-line code, and are particularly useful in conjunction with the special procedures CODE and SWI (software interrupt) described below. The registers AX, BX, CX, DX, SP, BP, SI, DI, ES, CS, SS, and DS are accessible where SP, BP, SS and CS cannot be used with SETREG. For register only the register constants provided by module SYSTEM should be used.

If the actual argument for value is a variable in one byte, only the lower half of the register is affected. For example, in SETREG (AX, ch), where ch is declared to be a CHAR, only the AL register is modified.

--- WARNING ---

Utmost care must be exercised when using GETREG and SETREG. It must be kept in mind that expression evaluation and address computation use registers and therefore might destroy the value of a register already set by SETREG or to be read by GETREG. It is impossible for the compiler to recognize such a situation and the programmer must take full responsibility.

Only constants, or variables and value parameters which are declared local to the procedure calling GETREG or SETREG, should be used for the second argument. This argument should be of a simple type. It should neither be an expression, contain a function call, index an array, nor be a global (module) variable or a VAR parameter. If necessary, input parameter values should be copied to local variables of simple types which can be used when calling SETREG. Only local variables of simple types should be used with GETREG. If necessary, their values should be copied to the real output parameters. If there are sequences of calls to SETREG or GETREG, no other statements should break such a sequence. All local copies of input values should be made before the first call to SETREG, and the values of the local variables should be copied back after the last call to GETREG.

Unpredictable effects may result from failure to heed this warning.
Implementation Features

CODE (code1Const, code2Const, ... : BYTE)

Insert binary machine instructions into the code. A call to CODE inserts the constant values, code1Const, code2Const, etc., in-line as executable code.

SWI(InterruptVectorNumber: CARDINAL)

This procedure is used to generate a software interrupt. It compiles into an INT instruction. The parameter must be a constant.

If you are using the procedure SWI to call the IBM-PC ROM BIOS or to call any other assembly routines, we strongly recommend that you save and restore the base pointer register BP. The value of the BP register is essential to LOGITECH Modula-2 because it is used to access local variables and procedure parameters.

To save and restore the BP register, use procedure CODE, which is also provided by module SYSTEM. Insert CODE(55H); right before, and CODE(5DH); right after the call to SWI. This pushes and pops the BP register to/from the stack, so that its value will be preserved.

ENABLE
DISABLE

Calls to the procedures ENABLE and DISABLE compile into STI and CLI instructions, which enables or disables interrupts.

Note: Any call to the operating system, or any input or output by means of the LOGITECH Modula-2 library may have the effect of enabling interrupts, thus undoing a previous call to DISABLE.

INBYTE (port: CARDINAL; VAR value: BYTEorWORD)
OUTBYTE (port: CARDINAL; value: BYTEorWORD)

Get or put a byte value from or to the specified I/O port.

INWORD (port: CARDINAL; VAR value: WORD)
OUTWORD (port: CARDINAL; value: WORD)

Get or put a word value from or to the specified I/O port.
Chapter 8

DOSCALL (functionNumber: CARDINAL; ...)

It generates a DOS function call via software interrupt 21H. The parameter list is variable, depending on the first parameter, which must be a constant and indicates the number of the DOS function. The appendix contains a detailed description of the available DOSCALLs.

Because the parameters of DOSCALL must be given to DOS in registers, no complicated expressions should be used. The compiler might easily run out of registers, resulting in compiler error 204.

EXTCALL (Procname : ARRAY of CHAR);

Procname must be a constant string. It tells the compiler to call a non Modula-2 procedure. It is up to you to take care of the parameter passing and calling conventions, perhaps in CODE statements. (See Chapter 6, Interfacing Other Languages).
8.1.4 Data Representation

The data types have the following internal representation in LOGITECH Modula-2:

- **BYTE**: One byte.
- **BOOLEAN**: One byte, TRUE=1, FALSE=0.
- **CHAR**: One byte, ASCII character set.
- **Enumeration Types**: One byte, elements are numbered 0..255.
- **WORD**: Two bytes.
- **INTEGER**: Two bytes, -32768..32767, two’s complement notation, least significant byte first.
- **LONGINT**: Four bytes, -2147483648..2147483648, twos complement notation, least significant byte first.
- **CARDINAL**: Two bytes, 0..65535, least significant byte first.
- **Subrange Types**: Same representation as the base type.
- **REAL**: Eight bytes, Intel 8087 double precision format (IEEE Floating Point standard).
- **SET**: The size of a SET is the number of bytes obtained by dividing the number of elements in the SET (up to 256) by 8 and rounded to the next higher integer. The elements are associated to the bits consistently with the above pictures.
- **BITSET**: Two bytes. If we number the elements of a set from 0 to 15, the representation in a memory word is:

<table>
<thead>
<tr>
<th>7 6 5 4 3 2 1 0</th>
<th>15 14 13 12 11 10 9 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>low byte</td>
<td>high byte</td>
</tr>
<tr>
<td><strong>POINTER</strong></td>
<td><strong>Proc</strong></td>
</tr>
<tr>
<td>-------------</td>
<td>-----------</td>
</tr>
<tr>
<td>Four 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).</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Array</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>An array is stored as a contiguous sequence of elements, with the indices in ascending order, the right-most index varying most quickly. If the base type fits in one byte (CHAR, BOOLEAN, enumeration) the elements are stored in sequential bytes. Otherwise, each element is stored on a word boundary (at an even address).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Record</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>The fields of a record are allocated in the order in which they are declared. The first field has the lowest address. If you select the Alignment option, fields with a size other than one byte are allocated on even addresses. Therefore, dummy bytes are included after odd sized elements.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Opaque Types</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Opaque Types are always allocated four bytes, regardless of their actual implementation.</td>
</tr>
</tbody>
</table>
8.1.5 Type Conversion and Type Transfer

There are two ways to deal with the "strong typing" of Modula-2: type conversion and type transfer.

Type Conversion

Type conversion provides the means to convert data from one type into another one, regardless of the internal representation. This is the system independent and portable way to convert data from one type to another. Therefore, whenever possible, type conversion should be used rather than type transfer. The procedures to make the conversion are built-in, standard procedures, or part of the LOGITECH Modula-2 library. They are as follows:

Standard Functions

- CHR
- ORD
- VAL
- FLOAT
- TRUNC

Library Functions

- MathLib0.real
- MathLib0.entier

Type conversion works by calculating a new value of a new type which corresponds to the value to be converted. Code may be executed to perform the conversion, and range checks are done for the resulting values.
Type Transfer

The second way is referred to as type transfer, or sometimes, as type coercion. This method is system dependent — it depends on the internal representation of data. Therefore, use type transfers with utmost care, and avoid them whenever possible.

With a type transfer, no conversion of data takes place. The data is simply interpreted in a different way, according to the new type structure.

*Modula-2* lets you use an identifier of a type, either a standard type like CHAR, or any user-defined type, as if it were a function procedure. The compiler does not produce any code for type transfers. A type transfer simply indicates that a value shall be interpreted in a different way.

A type transfer is only allowed if the object is of the same size as objects of the new type. When transferring a variable of type T1 into type T2, the following relation must be true:

\[
\text{SYSTEM.TSIZE (T1)} = \text{SYSTEM.TSIZE (T2)}
\]

When using type transfer instead of type conversion, be aware of the internal representation of data. Your programs may not run on other machines or with other implementations of *Modula-2*.

**Example 1:** Interpret the value of a SET as a CARDINAL

```plaintext
VAR  b : BITSET; (* 'b' and 'c' are both *)
     c : CARDINAL; (* represented as 2 bytes *)
     b := {0,5,15};
     c := CARDINAL(b); (* c = 2**0 + 2**5 + 2**15 *)
```

**Example 2:** Use a CARDINAL value in an INTEGER expression

```plaintext
VAR  i : INTEGER;
     c : CARDINAL;
     i := 2*i + INTEGER (c);
```

This second example illustrates an abuse of the type transfer. What was really meant was a conversion to integer. The correct solution is:

```plaintext
i := 2*i + VAL (INTEGER, c);
```
8.2 Priorities and Interrupts

8.2.1 Use of Priorities

Priorities can be specified in the header of a module. They are allowed in program and implementation modules, as well as in local modules declared inside of another module. Priorities are used to control the occurrence of interrupts. When no priority is specified, all interrupts may occur.

However, when a program is running at a certain priority, only interrupts of a higher priority will be accepted. At the highest priority all interrupts are disabled. Note also that running at the lowest, specified priority is very different from running without any priority.

A priority module is entered upon the execution of its module initialization code or upon a call of an exported procedure. A priority module is left upon return from its initialization code or upon return from an exported procedure. When entering a priority module, the interrupt control system (hardware and software) is set such that interrupts of a priority lower or equal to the one specified in that module are not passed to the processor. When leaving a priority module, the interrupt control system is reset to the state it was prior to entering that module. The procedure LISTEN, from module SYSTEM, allows a process to lower its priority temporarily. During the execution of the procedure LISTEN the interrupt control system is set such that all pending interrupts are accepted.

Inside a priority module, calls to procedures of other priority modules with a lower priority than that of the module with the call statement are not allowed. When this situation is detected by the compiler, an appropriate error message is produced (error 161). If a procedure of a module with no specified priority is called, the current priority remains unchanged. If a procedure of a module with higher priority is called, that higher priority becomes effective during execution of the called procedure. The old priority is restored upon return from that procedure.

Priorities are attached to processes. Upon a TRANSFER or IOTRANSFER to a process running at another priority, the interrupt control system is switched to the priority of the process which will be activated. The same holds true for the implicit coroutine transfer which occurs upon an interrupt.

When a subprogram terminates, the priority is set back to the value which was effective when the subprogram was loaded.
8.2.2 Priority Levels

Both the *LOGITECH Modula-2* compiler and the *LOGITECH Modula-2* run-time support only allow for a fixed range of priority levels. Eight priority levels are supported with values ranging from 0 (lowest level) to 7 (highest level). If a module priority is specified with a value out of this range, the compiler produces an appropriate error message (error 80).

**NOTE**

Do not confuse software priority level with hardware priority mask. The mapping between the two is explained in Section 8.2.4.2.
8.2.3 Interrupt Handling

There are three main ways to handle interrupts in LOGITECH Modula-2:

8.2.3.1 Standard Method with IOTRANSFER

In "standard" Modula-2, the procedure IOTRANSFER from module SYSTEM allows for the implementation of interrupt handlers. After the call to IOTRANSFER the interrupt handler is installed and is waiting for the specified interrupt. However, on 8086 based systems, the system needs to be notified that interrupts from the corresponding device may now occur. In LOGITECH Modula-2, the module Devices provides the capability to enable and disable interrupts from a device. After an interrupt handler has been installed, module 'Devices' should be used to enable interrupts from the corresponding device.

An interrupt handler should not call the operating system, for instance to write to the terminal or to a file. If the operating system is not reentrant, such a call may crash the whole system. In general, operating systems that do not support multi-tasking, for instance DOS 2.0, are not reentrant.

Please refer also to the definition module Devices and to the sample device driver InputDevice at the end of this section. Module InputDevice illustrates how an interrupt handler should be programmed with LOGITECH Modula-2.

8.2.3.2 Faster Method with IOTRANSFER

The standard method using IOTRANSFER as described in the previous section, associates a process with the next occurrence of the specified interrupt only. The procedure InstallHandler provided by module Devices allows you to install an interrupt handler that will be removed upon the proper termination. It associates a process, the interrupt handler, permanently with a particular interrupt. While it is not required to install an interrupt handler in this way, it may be useful for handling time critical interrupts. Installing an interrupt handler permanently improves the performance, by about 20 percent, of IOTRANSFER and of the implicit coroutine transfer that takes place when the interrupt occurs. InstallHandler must only be called after the process has been created (by means of NEWPROCESS) and before the process has called IOTRANSFER. For instance, it may be called at the beginning of the code of the process.
8.2.3.3 Low Level Interrupt Handling

This is the fastest method to handle interrupts, but also the least portable. Unlike the previous two methods, this implementation doesn’t perform a context switch on interrupts, but uses the current stack to handle the interrupt. This provides a great improvement in speed because the overhead of two transfers is removed.

One disadvantage of this method is that the debug utilities do not support the debugging of interrupt service routines, because the state of the stack does not have the usual form.

The RTSIntPROC module lets you install such Interrupt Service Routines. The procedure installed as ISR must be a PROC, but does not have to save registers or similar functions, because the module RTSIntPROC saves all registers and sends the End-Of-Interrupt command to the interrupt controller before the call to the Modula-2 procedure.

The procedure is executed "Interrupts Disabled". Thus, the procedure does not need to be in a priority module (and as the stack and context is unpredictable, it is not allowed in a priority module). As the procedure is executed "Interrupts Disabled", it must be fast to allow other interrupts to be serviced.

The procedure must not call DOS directly or indirectly, because DOS is not re-entrant, and the interrupt PROC may interrupt any running code, including DOS, depending when the interrupt occurs. For example, it shall not use library modules making Input/Output like FileSystem, Terminal, InOut, etc.
The following code shows a module which allows the installation of a Modula-2 procedure as an interrupt service routine:

```
MODULE ModulaISR;
FROM RTSIntPROC IMPORT SetIntPROC, FreeIntPROC;
FROM SimpleTerm IMPORT WriteString, WriteLn, KeyPressed;
VAR
  BreakPressed : BOOLEAN;
PROCEDURE ISR;
BEGIN
  BreakPressed := True; (* signal the event to the outside *)
END ISR;
BEGIN
  BreakPressed := FALSE;
  SetIntPROC ( ISR, 27 );
  LOOP
    IF BreakPressed THEN
      BreakPressed := FALSE; (* reset the event *)
      WriteString ( "Break !" ); (* make the time-consuming work *)
      WriteLn;
    END;
    IF KeyPressed() THEN EXIT END;
  END;
END;
FreeIntProc; (* releases all installed ISRs *)
END ModulaISR
```
8.2.3.4 How to Cope With Non-Reentrancy of MS-DOS

The non-reentrancy of MS-DOS appears to be a problem when writing a real-time kernel in Modula2 (as in other languages). The basic principle is to avoid task switching while DOS is in a 'critical section'. There is an undocumented DOS call (34H) which can be used to determine whether DOS is in such a critical section. Since DOSCALL 34H is not documented, it is not supported by the LOGITECH Modula-2 Compiler. The following program extract shows you how to get access to this information:

```modula2
MODULE scheduler;
   FROM SYSTEM IMPORT
      ADR, ADDRESS, SETREG, GETREG, SWI, AX, ES, BX;
   TYPE
      BooleanPtr = POINTER TO BOOLEAN;
   VAR
      criticalSectionPtr : BooleanPtr;
      aux              : ADDRESS;
   BEGIN
      SETREG (AX, 3400H);
      SWI (21H);
      GETREG (ES, aux.SEGMENT);
      GETREG (BX, aux.OFFSET);
      criticalSectionPtr := BooleanPtr(aux);
   END scheduler.
```

In the scheduler routine which actually performs the task switching, one must test the critical section flag in DOS:

```modula2
IF criticalSectionPtr A THEN
   (*
      don’t do the transfer to the waiting process,
      but let the interrupted process continue
   *)
   TRANSFER(currentProcess,interruptedProcess);
ELSE
   (* transfers to waiting process *)
   TRANSFER(currentProcess, waitingProcess);
END;
```

A similar check can be done to avoid DOS function calls in an interrupt handler routine, while DOS is in a critical section.
8.2.4 Implementation Notes

*LOGITECH Modula-2* implements priorities and device handling through the mask register of the interrupt controller in the *8086* system. The corresponding code is part of the *LOGITECH Modula-2* run-time support.

### 8.2.4.1 The Device Mask

The run-time support maintains a device mask that indicates from which devices interrupts are enabled. When a program is not running at any priority, the mask register of the interrupt controller is identical to this device mask. The initial value of the device mask corresponds to the value of the interrupt controller mask at the time when the *LOGITECH Modula-2* program was started.

The library module *Devices* provides procedures that allow a program to enable or disable interrupts from a device. These procedures are implemented by calls to functions of the run-time support, which modify the device mask. The device numbers used by module *Devices* and by the *LOGITECH Modula-2* run-time support correspond to the order and meaning of the bits in the mask register of the interrupt controller.

The run-time support maintains only one copy of the device mask. Thus, the device mask is shared among all processes and any subprograms of a *LOGITECH Modula-2* program.

### 8.2.4.2 The Priority Masks

To each priority level a particular priority mask corresponds, which masks out the interrupts from all devices with the same or a lower priority. The order and meaning of the bits in the priority mask are the same as those in the device mask and in the mask register of the interrupt controller. The mapping between the priorities and the priority masks is done by the run-time support. The value of the priority level is used as an index to a table of priority masks.

The table of priority masks is initialized as follows: It masks bit seven for priority level zero, the lowest priority. It masks bit six and seven for priority level one, and so on. For priority level seven, the highest priority, all bits in the mask are set such that all interrupts are disabled. These default settings correspond to the *IBM-PC* hardware. If necessary, the values in this table may be modified to implement a different priority scheme that reflects the hardware properties of a given *8086* based system.
8.2.4.3 The Interrupt Controller Mask

When a program is not running at any priority, LOGITECH Modula-2 sets the mask register of the interrupt controller such that it is identical to the device mask. If a program is running at a particular priority, the mask register of the interrupt controller is set to the logical OR of the device mask and the corresponding priority mask. In this way, all interrupts are disabled which are masked out either in the device mask or in the current priority mask.

The field PriorityMask of the process descriptor holds the priority mask that corresponds to the priority at which the process is running. When creating a new process (procedure NEWPROCESS), the initial value of the priority mask in the process descriptor is zero. This initial value indicates that the process is not running at any priority. If the procedure which constitutes the process is declared in a priority module, its priority becomes effective when the process is started. A process starts execution upon the first TRANSFER of control to it, after it was created by NEWPROCESS.

The mask register of the interrupt controller is always equal to the logical OR of the current device mask and the priority mask that corresponds to the priority at which the current process is running. When a coroutine transfer occurs upon a call to TRANSFER, IOTRANSFER, or upon an interrupt, the mask register of the interrupt controller is set according to the priority of the process that takes control and according to the value of the device mask. The mask register of the interrupt controller is also set accordingly whenever the priority changes because of a call to, or a return from, a priority module. When the device mask is modified, the mask register of the interrupt controller is updated according to the new device mask and according to the priority mask of the current process.
8.2.4.4 Monitor Entry and Exit

Priority modules are also called monitors. When entering or leaving a monitor, some code is executed to change the priority of the current process. This code is part of the LOGITECH Modula-2 run-time support.

The compiler generates a call to the run-time support (RTS call) in the procedure entry code (to the Monitor Entry function) and in the procedure exit code (to the Monitor Exit function) for every procedure exported from a priority module. The procedure LISTEN from module SYSTEM is translated to another RTS call, the Listen function.

The Monitor Entry function is called after the possible stack-test and after the stack pointer is decremented by the size of the local data. It saves the current priority mask, from the process descriptor of the current process, onto the stack of the entered procedure. The new priority is used as an index in a table that contains the value of the priority mask for each priority level. The new priority mask is stored in the process descriptor. The mask register of the interrupt controller is set to the logical OR of the new priority mask and the current device mask.

The Monitor Exit function restores the old priority mask from the top of the stack back into the process descriptor. It also sets the mask register of the interrupt controller to the logical OR of the old priority mask and the current device mask. Unless the device mask has been changed while running on priority, the mask register of the interrupt controller will have the same value as before entering the priority module.

The Listen function first sets the current priority to no priority, in a way similar to the Monitor Entry function. The value of the priority mask for no priority is not stored in the table of priority masks. The mask for no priority has all bits set to zero. Therefore, the mask register of the interrupt controller will be equal to the device mask. The Listen function then sets the interrupt enable flag of the processor. At this point, all pending interrupts may come through, if they were enabled in the device mask. After the execution of a no-operation instruction, the Listen function restores the old priority in a way similar to the Monitor Exit function.
### 8.2.5 The Definition Module for "InputDevice"

```
DEFINITION MODULE InputDevice;
(*
  Sample Input Device

This is the sample interface definition for a small input device driver, which
shows how interrupt driven devices should be handled in LOGITECH Modula-2.
A corresponding scheme can be used for interrupt driven output devices.
*)

EXPORT QUALIFIED
  ReadInfo;

PROCEDURE ReadInfo (VAR info: Information);
(*
   get information from the device, where 'Information' might be
   of type 'CHAR' for a character device
*)

END InputDevice.
```

### 8.2.6 The Implementation Module for "InputDevice"

```
IMPLEMENTATION MODULE InputDevice [priority];
(*
  Sample Input Device

This is a small sample input device driver, which shows how interrupt driven
devices should be handled in LOGITECH Modula-2.
A corresponding scheme can be used for interrupt driven output devices
*)

FROM SYSTEM IMPORT
  PROCESS, NEWPROCESS, TRANSFER, IOTRANSFER, ADR, SIZE, BYTE, ADDRESS;

FROM RTSMain IMPORT
  InstallTermProc;

FROM Devices IMPORT
  GetDeviceStatus, SetDeviceStatus, SaveInterruptVector, RestoreInterruptVector,
  InstallHandler;

CONST
  device = ??;
  (* bit number in interrupt controller mask *)

  interruptVectorNumber = ??;
  (* interrupt vector used by device *)
```
VAR
mainP, driverP: PROCESS; (* Modula-2 coroutines *)
workspace: ARRAY [0..?] OF BYTE; (* workspace for driver coroutine *)
oldInterruptVector : ADDRESS;
oldDeviceStatus: BOOLEAN;
activ: BOOLEAN; (* indicates whether the device driver has been activated *)

PROCEDURE ReadInfo (VAR info : Information);
BEGIN
(* get info from a buffer *)
END ReadInfo;

PROCEDURE DeviceDriver;
BEGIN
(* here we could associate the process permanently to the given interrupt vector number by: InstallHandler(driverP, interruptVectorNumber); This call improves the performance of the interrupt handling *)
LOOP
  IOTRANSFER(driverP, mainP, interruptVectorNumber);
  (* handle the interrupt, put info into a buffer *)
END; (* LOOP *)
END DeviceDriver;

PROCEDURE StartDevice;
BEGIN
  IF NOT activ THEN
    SaveInterruptVector (interruptVectorNumber, oldInterruptVector);
    (* save interrupt vector used by device *)
  GetDeviceStatus(device, oldDeviceStatus);
  (* save old device status (interrupts enabled/disabled *)
  activ := TRUE;
  NEWPROCESS (DeviceDriver, ADR (workspace), SIZE (workspace), driverP);
  (* create a Modula-2 process for the driver *)
  TRANSFER(mainP, driverP);
  (* transfer control to the driver process *)
  SetDeviceStatus(device, TRUE);
  (* allow (enable) interrupts from the device *)
END;
END StartDevice;
Chapter 8

PROCEDURE StopDevice;
BEGIN
  IF activ
    THEN activ := FALSE;
  SetDeviceStatus(device, oldDeviceStatus);
  (* restore the original device status (interrupts enabled/disabled) *)
  RestoreInterruptVector (interruptVectorNumber, oldInterruptVector);
  (* restore the original value of the interrupt vector used by the device *)
END;
END StopDevice;

PROCEDURE InitDevice;
BEGIN
  (* initialize device if necessary *)
END InitDevice;

BEGIN
  activ := FALSE;
  InitDevice;
  StartDevice;
  InstallTermProc(StopDevice);
  (*
    install 'StopDevice' as a termination routine, in order to properly stop
    the device driver when the program that uses the driver terminates
  *)
END InputDevice.
8.3 DOSCALL

The DOSCALL procedure must be imported from the SYSTEM module. It provides a rather simple way to access the underlying operating system from programs written in Modula-2. For the description of each of these functions we refer to the corresponding MS-DOS or PC-DOS Manual. The actual parameters of the procedures should not be very complicated. The compiler might easily run out of registers.

The first line is a Modula-2 procedure declaration. The second line notes for each parameter the register(s) in which it is passed. The type BYTEWORD (which doesn’t exist in Modula-2) means that any type compatible with BYTE or WORD is possible for the actual parameter.

Example:

```
DOSCALL (15;
    FCBAddr : ADDRESS;
    VAR returnCode : BYTEWORD);
    AH DS : DX AL
```

possible use:

```
VAR FCB : ARRAY[0...35] OF CHAR;
    returnVal : CARDINAL

DOSCALL(15, ADR(FCB), returnVal);
    IF returnVal =...
```

The standard procedure DOSCALL has a variable parameter list. This parameter list depends on the first parameter that must be a constant. This constant is the number of the DOS function to be called.
Chapter 8

The formats of these functions are:

Function 0H: Program Terminate

```c
DOSCALL (0H)
```

Function 1H: Keyboard Input

```c
DOSCALL (1H; VAR char:BYTEWORD);
AH AL
```

Function 2H: Display Output

```c
DOSCALL (2H; char:BYTEWORD);
AH DL
```

Function 3H: Auxiliary Input

```c
DOSCALL (3H; VAR char:BYTEWORD);
AH AL
```

Function 4H: Auxiliary Output

```c
DOSCALL (4H; char:BYTEWORD);
AH AL
```

Function 5H: Printer Output

```c
DOSCALL (5H; char:BYTEWORD);
AH DL
```

Function 6H: Direct Console I/O

```c
DOSCALL (6H; OFFH; VAR char:BYTEWORD;
AH DL AL
VAR ready : BOOLEAN); (input)
ZF
DOSCALL (6H; char:BYTEWORD); (output)
AH DL
```

Function 7H: Direct console Input without echo

```c
DOSCALL (7H; VAR char:BYTEWORD);
AH AL
```

Function 8H: Console input without echo

```c
DOSCALL (8H; VAR char:BYTEWORD);
AH AL
```
Implementation Features

Function 9H: Print String

\[
\text{DOSCALL (9H; stringaddr:ADDRESS);}
\]
\[
\text{AH } DS : DX
\]

Function 0AH: Buffered Keyboard input

\[
\text{DOSCALL (0AH; stringaddr:ADDRESS);}
\]
\[
\text{AH } DS : DX
\]

Function 0BH: check standard input status

\[
\text{DOSCALL (0BH; VAR status:BYTEWORD);}
\]
\[
\text{AH AL}
\]

Function 0CH: Clear standard input buffer and invoke a standard input function

The second parameter (input function) determines the form of the parameter list.
It must be one of the constants (functions) 1H, 6H, 7H, 8H, or 0AH.

\[
\text{DOSCALL (0CH; 1H; VAR char:BYTEWORD);} \\
\text{AH AL AL}
\]
\[
\text{DOSCALL (0CH; 6H; VAR char:BYTEWORD);} \\
\text{AH AL AL}
\]
\[
\text{[DL = OFFH implicitly]} \\
\text{VAR ready:BOOLEAN); ZF}
\]
\[
\text{DOSCALL (0CH; 7H; VAR char:BYTEWORD);} \\
\text{AH AL AL}
\]
\[
\text{DOSCALL (0CH; 8H; VAR char:BYTEWORD);} \\
\text{AH AL AL}
\]
\[
\text{DOSCALL (0CH; OAH; stringaddr:ADDRESS);} \\
\text{AH AL DS : DX}
\]

Function ODH: Disk reset

\[
\text{DOSCALL (ODH)} \\
\text{AH}
\]

Function OEH: Select Disk

\[
\text{DOSCALL(0EH; drive : BYTEWORD;}} \\
\text{AH DL}
\]
\[
\text{VAR nrofdrives : WORD); AL}
\]
Chapter 8

Function 0FH: Open File

DOSCALL (0FH; FCBaddr:ADDRESS;
AH DS:DX
VAR returnCode : BYTEWORD);
AL

Function 10H: Close File

DOSCALL (10H; FCBaddr:ADDRESS;
AH DS : DX
VAR returnCode : BYTEWORD);
AL

Function 11H: Search for the first entry

DOSCALL (11H; FCBaddr:ADDRESS;
AH DS : DX
VAR returnCode : BYTEWORD);
AL

Function 12H: Search for the next entry

DOSCALL (12H; FCBaddr : ADDRESS;
AH DS:DX
VAR returnCode:BYTEWORD);
AL

Function 13H: Delete File

DOSCALL (13H; FCBaddr : ADDRESS;
AH DS:DX
VAR returnCode : BYTEWORD);
AL

Function 14H: Sequential Read

DOSCALL(14H; FCBaddr : ADDRESS;
AH DS : DX
VAR returnCode : BYTEWORD);
AL

Function 15H: Sequential Write

DOSCALL (15H; FCBaddr : ADDRESS;
AH DS : DX
VAR returnCode : BYTEWORD);
AL
Function 16H: Create File

DOSCALL (16H; FCBaddr : ADDRESS;
   AH  DS : DX
   VAR returnCode : BYTEWORD);
   AL

Function 17H: Rename File

DOSCALL (17H; FCBaddr : ADDRESS;
   AH  DS : DX
   VAR returnCode : BYTEWORD);
   AL

Function 19H: Current Disk

DOSCALL (19H; VAR currDrive : BYTEWORD);
   AH  AL

Function 1AH: Set Disk Transfer Address

DOSCALL (1AH; DTA : ADDRESS);
   AH  DS : DX

Function 1BH: Allocation table information

DOSCALL (1BH; VAR FATaddr : ADDRESS;
   AH  DS : BX
   VAR nrallocUnits, nrSectors,
       DX  AL
   sectSize : BYTEWORD);
   CX

Function 1CH: (not implemented)

Function 21H: Random Read

DOSCALL(21H; FCBaddr:ADDRESS;
   AH  DS : DX
   VAR returnCode : BYTEWORD);
   AL

Function 22H: Random Write

DOSCALL(22H; FCBaddr:ADDRESS;
   AH  DS : DX
   VAR returnCode : BYTEWORD);
   AL
Chapter 8

Function 23H: File Size

DOSCALL(23H; FCBaddr : ADDRESS;
AH   DS : DX
VAR returnCode : BYTEWORD);
AL

Function 24H: Set Random Record Field

DOSCALL(24H; FCBaddr : ADDRESS);
AH   DS : DX

Function 25H: Set Interrupt Vector

DOSCALL(25H; vectorVal : ADDRESS;
IntNumber : BYTEWORD);
AH   DS : DX   AL

Function 26H: Create a new program segment

DOSCALL(26H; progSegment:BYTEWORD);
AH   DX

Function 27H: Random Block Read

DOSCALL(27H; FCBaddr:ADDRESS;
AH   DS:DX
VAR nrofBytes:BYTEWORD;
CX
VAR returnCode:BYTEWORD);
AL

Function 28H: Random Block Read

DOSCALL(28H; FCBaddr:ADDRESS;
AH   DS:DX
VAR nrofBytes:BYTEWORD;
CX
VAR returnCode:BYTEWORD);
AL

Function 29H: Parse Filename

DOSCALL (29H; FCBaddr : ADDRESS;
mode : BYTEWORD;
AH   ES:D   AL
VAR stringaddr : ADDRESS;
DS:SI
VAR returnCode : BYTEWORD);
Implementation Features

Function 2AH: Get Date

\[
\text{DOSCALL(2AH; VAR year:WORD; VAR monthday:WORD); }
\]
\[
\text{AH} \quad \text{CX} \quad \text{DX}
\]

Function 2BH: Set Date

\[
\text{DOSCALL(2BH; year:WORD; monthday:WORD; }
\]
\[
\text{AH} \quad \text{CX} \quad \text{DX}
\]
\[
\text{VAR returnCode:BYTEWORD};
\]
\[
\text{AL}
\]

Function 2CH: Get Time

\[
\text{DOSCALL(2CH; VAR hourminute, secondmillisec:WORD); }
\]
\[
\text{AH} \quad \text{CX} \quad \text{DX}
\]

Function 2DH: Set Time

\[
\text{DOSCALL(2DH; hourminute, secondmillisec:WORD; }
\]
\[
\text{AH} \quad \text{CX} \quad \text{DX}
\]
\[
\text{VAR returnCode:BYTEWORD};
\]
\[
\text{AL}
\]

Function 2EH: Set/Reset Verify Switch

\[
\text{DOSCALL(2EH; zero:BYTEWORD; onoff:BYTEWORD); }
\]
\[
\text{AH} \quad \text{DL} \quad \text{AL}
\]
8.3.1 Extensions for DOS 2.0

Function 2FH: Get DTA

DOSCALL (2FH; VAR DTAaddr: ADDRESS);
AH   ES:BX

Function 30H: Get DOS version

DOSCALL (30H; VAR major, minor: BYTE);
AH   AL   AH

Function 31H: Terminate and remain resident

DOSCALL (31H; exitCode : BYTEWORD);
AH   AL
paragraphs : WORD);
DX

Function 33H: Ctrl-Break-Check

DOSCALL(33H; mode:BYTEWORD; VAR state:BYTE);
AH   AL   DL

Function 35H: Get Vector

DOSCALL(35H; vector:BYTEWORD; VAR vector:ADDRESS
AH   AL   ES:BX

Function 36H: Get disk free space

DOSCALL(36H; drive:BYTEWORD; VAR valid:BYTEWORD;
AH   DL   AX
VAR availClusters:BYTEWORD;
BX
VAR totclust:BYTEWORD;
DX
VAR bytesPerSect:BYTEWORD);
CX

Function 38H: Return Country dependent information

DOSCALL(38H; buffAddr:ADDRESS; fctcode:BYTEWORD);
AH   DS:DX   AL

Function 39H: Create a subdirectory (MKDIR)

DOSCALL(39H; stringaddr:ADDRESS;
AH   DS : DX
VAR error : WORD);
AX,CF
(error = 0 means no error;
refer to the table in the DOS manual for other errors)
Implementation Features

Function 3AH: Remove a directory entry (RMDIR)

```
DOSCALL(3AH; stringaddr:ADDRESS; VAR error:WORD);
   AH   DS:DX   AX,CF
```

Function 3BH: Change the current directory (CHDIR)

```
DOSCALL(3BH; stringaddr:ADDRESS; VAR error:WORD);
   AH   DS:DX   AX,CF
```

Function 3CH: Create a File

```
DOSCALL(3CH; stringaddr:ADDRESS; attrib:BYTEWORD;
   AH   DS:DX   CX
VAR handle:BYTEWORD; VAR error:WORD);
   AX   AX,CF
```

Function 3DH: Open a File

```
DOSCALL(3DH; stringaddr:ADDRESS; access:BYTEWORD;
   AH   DS:DX   AL
VAR handle:BYTEWORD; VAR error:WORD);
   AX   AX,CF
```

Function 3EH: Close a file handle

```
DOSCALL(3EH; handle:WORD; VAR error:WORD);
   AH   BX   AX,CF
```

Function 3FH: Read from a file or device

```
DOSCALL(3FH; handle:WORD; nrbytes:WORD;
   AH   BX   CX
buffAddr:ADDRESS;
   DS:DX
VAR readBytes:BYTEWORD;
   AX
VAR error:WORD);
   AX,CF
```

Function 40H: Write to a file or device

```
DOSCALL(40H; handle:WORD; nrbytes:WORD;
   AH   BX   CX
buffAddr:ADDRESS;
   DS:DX
VAR writtenBytes:BYTEWORD; VAR error:WORD);
   AX   AX,CF
```
Chapter 8

Function 41H: Delete a file from a specified directory

DOSCALL(41H; stringaddr:ADDRESS; VAR error:WORD);
   AH DS:DX  AX,CF

Function 42H: Move file read/write pointer

DOSCALL(42H; handle:WORD; method:BYTEWORD;
   AH BX AL
   inHigh,inLow:WORD;
   CX DX
   VAR outHigh,outLow:WORD; VAR error:WORD);
   DX AX  AX,CF

Function 43H: Change File Mode

DOSCALL(43H; stringaddr:ADDRESS; fctcode:BYTEWORD;
   AH DS:DX AL
   VAR mode:BYTEWORD; VAR error:WORD);
   CX AX,CF

Function 44H: I/O control for devices

The procedure depends on the value of the second parameter that must be a constant. This parameter determines the function to execute:

Get device info

DOSCALL(44H; 0; handle:WORD;
   AH AL BX
   VAR deviceinfo:BYTEWORD;
   DX
   VAR error:WORD);
   AX,CF

Set device info

DOSCALL(44H; 1; handle:WORD;
   AH AL BX
   deviceinfo:BYTEWORD;
   DX
   VAR error:WORD);
   AX,CF

196
Read Bytes from device control channel

DOSCALL(44H; 2; handle:WORD;
AH AL BX
nrBytes:BYTEWORD; buffAddr:ADDRESS;
CX DS:DX
VAR transferredbytes:BYTEWORD;
AX
VAR error:WORD);
AX,CF

Write Bytes to device control channel

DOSCALL(44H; 3; handle:WORD;
AH AL BX
nrBytes:BYTEWORD; buffAddr:ADDRESS;
CX DS:DX
VAR transferredbytes:BYTEWORD;
AX
VAR error:WORD);
AX,CF

Read Bytes from drive control channel

DOSCALL(44H; 4; drive:BYTEWORD;
AH AL BL
nrBytes : BYTEWORD;
buffAddr : ADDRESS;
CX DS:DX
VAR transferredbytes : BYTEWORD;
AX
VAR error:WORD);
AX,CF

Write Bytes to drive control channel

DOSCALL(44H; 5; drive:BYTEWORD;
AH AL BL
nrBytes:BYTEWORD; buffAddr:ADDRESS;
CX DS:DX
VAR transferredbytes:BYTEWORD;
AX
VAR error:WORD);
AX,CF
Chapter 8

Get Input Status

DOSCALL(44H; 6; handle: WORD; VAR status:BYTEWORD;
AH AL BX AX
VAR error:WORD);
AX,CF

Get Output Status

DOSCALL(44H; 7; handle: WORD; VAR status:BYTEWORD;
AH AL BX AX
VAR error:WORD);
AX,CF

Function 45H: Duplicate a file handle

DOSCALL(45H; handle1:WORD; VAR handle2:BYTEWORD;
AH BX AX
VAR error:WORD);
AX,CF

Function 46H: Force a duplicate of a file

DOSCALL(46H; handle1:WORD; VAR handle2:BYTEWORD;
AH BX CX
VAR error:WORD);
AX,CF

Function 47H: Get Current Directory

DOSCALL(47H; drive:BYTEWORD; straddr:ADDRESS;
AH DL DS:SI
VAR error:WORD);
AX,CF

Function 48H: Allocate Memory

DOSCALL(48H; VAR paragraphs:BYTEWORD;
AH BX
VAR membase:BYTEWORD;
AX
VAR error:WORD);
AX,CF

Function 49H: Free allocated Memory

DOSCALL(49H; segaddr:ADDRESS;
AH ES must be a paragraph address
VAR error:WORD);
AX,CF
Function 4AH: SETBLOCK-Modify allocated memory blocks

DOSCALL(4AH; blockaddr:ADDRESS;
    AH   ES must be a paragraph address

VAR paragraphs:BYTEWORD;
    BX

VAR error:WORD);
    AX, CF

Function 4BH: Load or execute a program

DOSCALL (4BH;stringaddr : ADDRESS;
    parblock : ADDRESS;
    AH    DS : DX    ES:BX

fctval : BYTWORD;
    AL

VAR error : WORD);
    AX, CF

Function 4CH: Terminate a process(Exit)

DOSCALL(4CH; returnCode:BYTEWORD);
    AH    AL

Function 4DH: Retrieve the return code of a sub-process(Wait)

DOSCALL (4DH; VAR retCode:BYTEWORD);
    AH    AX

Function 4EH: Find first matching file

DOSCALL (4EH; stringaddr:ADDRESS; attribut:BYTEWORD;
    AH    DS:DX    CX

VAR error:WORD);
    AX, CF

Function 4FH: Find next matching file

DOSCALL (4FH; VAR error:WORD);
    AH    AX, CF

Function 54H: Get Verify state

DOSCALL (54H; VAR state:BYTE);
    AH    AL

Function 56H: Rename a file

DOSCALL (56H; fromstring,tostring:ADDRESS;
    AH    DS : DX    ES : DI

VAR error:WORD);
    AX, CF
Chapter 8

Function 57H: Get/Set a file's date and time

DOSCALL(57H; handle : WORD;
mode : BYTEWORD;
AH BX AL
VAR date,time : BYTEWORD;
DX CX
VAR error : WORD);
AX, CF
8.4 Decimals

The module Decimals provides functions for arithmetic and formatting with decimal numbers of 18 or less digits. These functions are appropriate for business-oriented computation.

8.4.1 Internal and External Format

Decimal numbers have two formats -- external and internal. Numbers in external format are represented by character strings. This external format is used for reading and writing numbers to the console or printer in a form you can understand. Arithmetic operations are performed on numbers stored in an encoded, internal format. The procedures StrToDec and DecToStr convert decimal numbers between internal and external format. StrToDec encodes decimal numbers and DecToStr decodes decimal numbers.

8.4.2 Types

Module Decimals provides the following types:

**DECIMAL**

Used for internal representation of a decimal number. Arithmetic operations are performed on variables of type DECIMAL.

**DecState**

A variable of type DECIMAL has a state of type DecState associated with it. This state may have the following values:

- **NegOvfl** indicates negative overflow
- **Minus** indicates negative decimal value
- **Zero** indicates value 0
- **Plus** indicates positive decimal value
- **PosOvfl** indicates positive overflow
- **Invalid** indicates invalid number

The procedure DecStatus returns the state of a decimal variable.
8.4.3 Variables

The following variables are exported from module Decimals:

DecValid

Indicates the success of the last operations. DecValid is set after each call to a conversion or arithmetic procedure. It is set to FALSE if the operation failed.

Remainder

Remainder is set after each division operation with procedure DivDec. DivDec returns an integer number for the quotient. If the result is not an integer number, Remainder indicates the first digit that appears after the decimal point. For example, the division of 39 by 8 yields a quotient of 4. The remainder is equal to 8 because this digit appears immediately after the decimal point in the exact result of 4.875. If the division operation fails, the remainder is ‘?’.

8.4.4 Conversion and Status Procedures

The following conversion and status procedures are provided by module Decimals:

StrToDec

Converts numbers from external to internal format. (Explained in greater detail below.)

DecToStr

Converts numbers from internal to external format. (Explained in greater detail below.)

DecStatus

Returns the current state of a decimal variable. DecStatus can be used to get the sign of a valid decimal number. When an operation fails, you can call the procedure DecStatus to determine the actual arithmetic error. DecStatus specifies the error status of the decimal variable according to type DecState.
8.4.5 Arithmetic Operations

The following arithmetic operations can be performed with variables of type DECIMAL:

**CompareDec**

Compares two decimal values, Dec0 and Dec1. Output is an integer value as follows:

<table>
<thead>
<tr>
<th>Output</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Dec0 is greater than Dec1</td>
</tr>
<tr>
<td>0</td>
<td>Dec0 equals Dec1</td>
</tr>
<tr>
<td>-1</td>
<td>Dec0 is less than Dec1</td>
</tr>
</tbody>
</table>

**AddDec**

Adds two decimal values, Dec0 and Dec1. The output is the sum of the two values, a decimal value.

**SubDec**

 Performs the subtraction of one decimal value, Dec1, from another decimal value, Dec0. The output is the difference of the two values, a decimal value.

**MulDec**

Performs the multiplication of two decimal values, a multiplicand, Dec0 and a multiplier, Dec1. The output is the product of the two values, a decimal value.

**DivDec**

Performs the division of one decimal value by another. The dividend, Dec0 is divided by the divisor, Dec1. Output is the quotient of the two values, a decimal value.

The remainder is placed in the global variable Remainder as previously explained.

**NegDec**

Returns the negative value of a decimal value.
8.4.6 Pictures

Numbers in external format are stored in character strings. These strings may include a currency character, commas and decimal points. For example:

$923,841,371.38

is a decimal number in external format.

So called "pictures" are used for the conversion between the string representation of decimal numbers in external format and their representation in internal format. Pictures indicate how decimal numbers appear in external format. They control the occurrence of leading blanks, leading zeros, number signs, currency characters, commas and decimal points.

For example, the picture which corresponds to the decimal number shown above is:

$$,,$$$,$$$,,$$9.99

With the picture $ZZZZZZZZZZ9$ the same decimal number would have appeared as:

92384137138
8.4.7 Picture Characters

Blank spaces may not appear in a picture. Pictures may consist of the following characters only:

- **9** digit
- **Z** nonzero digit or leading blank
- **$** nonzero digit, leading blank, or $
- **S** sign of number (+ or –)
  - decimal point
  - comma or leading blank

If the first character of a picture is a dollar sign ($), it will appear as a currency character in the external format. The currency character floats across any leading blanks so it appears adjacent to the leftmost digit. However, if a decimal value consists of the same number of digits as the picture which represents it, each dollar sign will be replaced by a digit and thus, no currency character will appear.

Numbers without leading zeros are represented with 'Z's. 'Z' is replaced by a digit if there is one, otherwise it is replaced by a blank.

'9's represent numbers which require leading zeros to be displayed. A '9' is replaced by a digit if there is one, otherwise it is replaced by a zero.

In the following picture, the '9's guarantee that dollar amounts less than $1.00 appear in standard form.

$$$,$$9.99

The following numbers correspond to this picture:

- $0.39
- $369.00
- $48,327.04

Sign characters (S) and decimal points (.) do not float across leading blanks, they appear in their specified position. Commas (,), 'Z' and $ characters correspond to leading blanks when they appear to the left of a number.
8.4.8 Procedure StrToDec

The procedure StrToDec uses pictures to convert numbers from external to internal format. If the input string is shorter than the picture string, leading blanks are added until it is the same length as the picture. A currency character can appear only once in the input string, and it must be adjacent to the leftmost digit. Commas are matched if they are within the number, or ignored if they appear to the left of the number. The sign character is matched by a '+', '-' or a blank. Decimal points are matched unconditionally.

Pictures ensure that input strings will be within a limited range. StrToDec sets DecValid to FALSE and the state of the decimal result to Invalid under the following conditions:

- The input string does not match the picture specification.
- The input string is longer than the picture string.
- The input string and the picture specify more than 18 digits.

8.4.9 Procedure DecToStr

If the number of digits in a number in external format exceeds the number of digit characters in the picture which represents it, DecToStr sets DecValid to FALSE and returns an Invalid format string. Thus, pictures can be used to control the maximum number of digits that can appear in a number.

Procedure DecToStr represents erroneous decimal variables with special character strings depending on the state of the decimal variable:

<table>
<thead>
<tr>
<th>State</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>PosOvfl</td>
<td>+ + + + + + + +</td>
</tr>
<tr>
<td>NegOvfl</td>
<td>- - - - - - -</td>
</tr>
</tbody>
</table>

The length of the string is determined by the length of the corresponding picture.

8.4.10 Error Propagation

Once an error occurs in a decimal variable as the result of an operation, the error remains through all the operations involving the variable. The following tables show how errors are propagated by the arithmetic operations. For operations in the form A <operation> B, the leftmost column represents states of A and the topmost row represents states of B.
## Implementation Features

### ADDITION and SUBTRACTION

<table>
<thead>
<tr>
<th>A/B</th>
<th>NegOvfl</th>
<th>Minus</th>
<th>Zero</th>
<th>Plus</th>
<th>PosOvfl</th>
<th>Invalid</th>
</tr>
</thead>
<tbody>
<tr>
<td>NegOvfl</td>
<td>NegOvfl</td>
<td>NegOvfl</td>
<td>NegOvfl</td>
<td>NegOvfl</td>
<td>Invalid</td>
<td>Invalid</td>
</tr>
<tr>
<td>Minus</td>
<td>NegOvfl</td>
<td>NegOvfl</td>
<td>PosOvfl</td>
<td>PosOvfl</td>
<td>Invalid</td>
<td>Invalid</td>
</tr>
<tr>
<td>Zero</td>
<td>NegOvfl</td>
<td>PosOvfl</td>
<td>NegOvfl</td>
<td>PosOvfl</td>
<td>Invalid</td>
<td>Invalid</td>
</tr>
<tr>
<td>Plus</td>
<td>NegOvfl</td>
<td>Invalid</td>
<td>Invalid</td>
<td>Invalid</td>
<td>Invalid</td>
<td>Invalid</td>
</tr>
<tr>
<td>PosOvfl</td>
<td>Invalid</td>
<td>Invalid</td>
<td>Invalid</td>
<td>Invalid</td>
<td>Invalid</td>
<td>Invalid</td>
</tr>
<tr>
<td>Invalid</td>
<td>Invalid</td>
<td>Invalid</td>
<td>Invalid</td>
<td>Invalid</td>
<td>Invalid</td>
<td>Invalid</td>
</tr>
</tbody>
</table>

### MULTIPLICATION

<table>
<thead>
<tr>
<th>A/B</th>
<th>NegOvfl</th>
<th>Minus</th>
<th>Zero</th>
<th>Plus</th>
<th>PosOvfl</th>
<th>Invalid</th>
</tr>
</thead>
<tbody>
<tr>
<td>NegOvfl</td>
<td>PosOvfl</td>
<td>PosOvfl</td>
<td>Zero</td>
<td>NegOvfl</td>
<td>NegOvfl</td>
<td>Invalid</td>
</tr>
<tr>
<td>Minus</td>
<td>PosOvfl</td>
<td>Plus</td>
<td>Zero</td>
<td>Minus</td>
<td>NegOvfl</td>
<td>Invalid</td>
</tr>
<tr>
<td>Zero</td>
<td>Zero</td>
<td>Zero</td>
<td>Zero</td>
<td>Zero</td>
<td>Zero</td>
<td>Invalid</td>
</tr>
<tr>
<td>Plus</td>
<td>NegOvfl</td>
<td>Minus</td>
<td>Zero</td>
<td>Plus</td>
<td>PosOvfl</td>
<td>Invalid</td>
</tr>
<tr>
<td>PosOvfl</td>
<td>NegOvfl</td>
<td>NegOvfl</td>
<td>Zero</td>
<td>PosOvfl</td>
<td>PosOvfl</td>
<td>Invalid</td>
</tr>
<tr>
<td>Invalid</td>
<td>Invalid</td>
<td>Invalid</td>
<td>Invalid</td>
<td>Invalid</td>
<td>Invalid</td>
<td>Invalid</td>
</tr>
</tbody>
</table>

### DIVISION

<table>
<thead>
<tr>
<th>A/B</th>
<th>NegOvfl</th>
<th>Minus</th>
<th>Zero</th>
<th>Plus</th>
<th>PosOvfl</th>
<th>Invalid</th>
</tr>
</thead>
<tbody>
<tr>
<td>NegOvfl</td>
<td>Invalid</td>
<td>Invalid</td>
<td>Invalid</td>
<td>Invalid</td>
<td>Invalid</td>
<td>Invalid</td>
</tr>
<tr>
<td>Minus</td>
<td>Invalid</td>
<td>Plus</td>
<td>Invalid</td>
<td>Minus</td>
<td>Invalid</td>
<td>Invalid</td>
</tr>
<tr>
<td>Zero</td>
<td>Invalid</td>
<td>Zero</td>
<td>Invalid</td>
<td>Invalid</td>
<td>Invalid</td>
<td>Invalid</td>
</tr>
<tr>
<td>Plus</td>
<td>Invalid</td>
<td>Minus</td>
<td>Invalid</td>
<td>Plus</td>
<td>Invalid</td>
<td>Invalid</td>
</tr>
<tr>
<td>PosOvfl</td>
<td>Invalid</td>
<td>Invalid</td>
<td>Invalid</td>
<td>Invalid</td>
<td>Invalid</td>
<td>Invalid</td>
</tr>
<tr>
<td>Invalid</td>
<td>Invalid</td>
<td>Invalid</td>
<td>Invalid</td>
<td>Invalid</td>
<td>Invalid</td>
<td>Invalid</td>
</tr>
</tbody>
</table>
8.5 REAL Arithmetic

*Modula-2* provides the data type REAL for floating point arithmetic. *LOGITECH Modula-2* supports the type REAL according to the IEEE standard for double precision floating point numbers. This format uses 8 bytes and is precise for 15 to 16 decimal digits. The values that can be represented range from $2.23 \times 10^9$ to the minus $308$th power, to $1.79 \times 10^{308}$:

$$2.23 \times 10^{-308} \leq |x| \leq 1.79 \times 10^{308}.$$

An optional 8087 math coprocessor can be added to many 8086/8088 based microcomputer systems. The 8087 performs floating point operations at very high speed. It provides a set of instructions which manipulate operands and yield results in the IEEE standard floating point format. If an 8087 coprocessor is not present, the REAL arithmetic functions must be emulated on the 8086 or 8088 processor.

*LOGITECH Modula-2* allows you to create programs that use an 8087 coprocessor and run at very high speed, as well as programs that emulate REAL arithmetic on the 8086 or 8088. A compile time switch decides whether the compiler generates 8087 inline code or code that uses the *LOGITECH REAL Arithmetic Emulator*. 
8.5.1 Simple Use of REAL Arithmetic

If you know your program will be running on a system with an 8087, you can use the compiler option Coprocessor. When compiling with the Coprocessor option, the compiler generates 8087 inline code for REAL operations. The library file C87LIB.LIB contains those LOGITECH Modula-2 library modules with 8087 inline code. If you compile with the Coprocessor option you should use these files when linking. A program that includes 8087 inline code requires an 8087 to run and cannot be executed on a system without an 8087.

The following describes the easiest way to create a program that runs without an 8087 coprocessor:

You must choose the compiler option Emulator if your program should run on a system without an 8087. When compiling with the Emulator option, which is the default, the compiler generates code for the LOGITECH REAL emulator. It also generates a reference to module Reals and to M2REAL.LIB which provides the REAL emulation. The library file E87LIB.LIB contains those LOGITECH Modula-2 library modules that have been compiled for the emulator. If you compile with the Emulator option you should use these files when linking. When you link with the files from E87LIB.LIB, your program will always use the emulator for REAL arithmetic.

A LOGITECH Modula-2 program compiled with the Emulator option can also be linked so that it uses an 8087 for execution and executes at maximum speed. It is also possible to link the program such that it uses an 8087 if one is present, and otherwise uses the emulator. The library M87LIB.LIB contains two versions of the library modules.
8.5.2 Choices for Using REAL Arithmetic

You may decide what kind of REAL arithmetic to use, either at compile-time, link-time or run-time.

The later the decision about which kind of REAL arithmetic to use is made, the more the portability of a program is improved. However, postponing this decision also increases memory requirements and decreases execution speed. The relative benefits and disadvantages of each alternative are detailed below.

LOGITECH Modula-2 offers the following alternatives for REAL arithmetic:

- At compile time the compiler options Coprocessor and Emulator determine the kind of code generated. Choose to generate 8087 inline code (Coprocessor option) or to compile for the LOGITECH REAL emulator (Emulator option). When compiling for the emulator, the compiler automatically generates a reference to the module Reals and to M2LIB.LIB which provides the REAL emulation. The compiler implicitly knows the interface of module Reals, therefore no symbol file needs to be provided.

- To generate 8087 inline code, an 8087 coprocessor is required to execute the program. The program will not run on a system without an 8087.

- If you choose to compile a module for the LOGITECH REAL emulator, how the program will be linked is still flexible. LOGITECH Modula-2 provides the following implementations for the REAL emulator (module Reals) and for the mathematical functions (module MathLib0):
  
  - If the program will not be executed on a system with an 8087, using the pure emulator version is the best choice. To use this version of the emulator, you must link with the files with E87LIB.LIB. A program linked in this way may be executed on a system with an 8087; however, it will never use the 8087.
  
  - The pure 8087 version of the emulator can be useful if the program being linked is executed on a system with an 8087. To use this version of the emulator, you should link with the files from C87LIB.LIB. A program linked with these files requires an 8087 for execution.
• Using a "mixed" version of the emulator postpones, until run-time, the decision of whether or not to use an 8087. The mixed emulator version is a combination of the other two versions, and thus is the most flexible option. If you choose to link with the mixed version of the emulator, then it will be determined at runtime whether an 8087 is present or not. Based on this determination:

• The program will use the 8087 if executed on a system with an 8087.

• The program will use the emulator to perform REAL arithmetic when the 8087 coprocessor is not present.

• Linking with the mixed version of the emulator increases flexibility and improves the portability of a program. The main disadvantage of the mixed version of the emulator is that the program requires more memory to run because the code for both forms of the emulation must be present. When linking the program, the M87LIB library file must be used.
Chapter 8

The distribution diskettes contain all three of the above-mentioned libraries with the emulator version E87LIB.LIB as the default, since it has been copied to M2REAL.LIB. Thus when you use the compiler and the linker with the default options, the LOGITECH REAL Arithmetic Emulator will be used.

M2REAL.LIB contains the following modules of the *LOGITECH Modula-2 Library*.

<table>
<thead>
<tr>
<th>Reals</th>
<th>The <em>LOGITECH Real Arithmetic Emulator</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>MathLib0</td>
<td></td>
</tr>
</tbody>
</table>

These two modules are differently implemented in each version of M2REAL.LIB.

- E87LIB.LIB: Contains emulator code
- C87LIB.LIB: Contains 8087 code
- M87LIB.LIB: Has both emulator and 8087 code, triggered by a switch that indicates the presence of the mathematical co-processor.

<table>
<thead>
<tr>
<th>RealConversions</th>
</tr>
</thead>
<tbody>
<tr>
<td>RealInOut</td>
</tr>
<tr>
<td>FloatingUtilities</td>
</tr>
<tr>
<td>Random</td>
</tr>
<tr>
<td>DurationOps</td>
</tr>
</tbody>
</table>

These are those modules that use the type REAL. They all have some implementation, but are compiled with different options.

<table>
<thead>
<tr>
<th>Library</th>
<th>Use Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>E87LIB.LIB</td>
<td>/E</td>
</tr>
<tr>
<td>C87LIB.LIB</td>
<td>/C</td>
</tr>
<tr>
<td>M87LIB.LIB</td>
<td>/E</td>
</tr>
</tbody>
</table>
Implementation Features

1) For 8087 Inline Code

Copy the library C87LIB.LIB onto M2REAL.LIB

Compile all your modules which use floating point arithmetic with the /C option.

2) For The Pure Emulator

Copy the library E87LIB.LIB onto M2REAL.LIB

Compile all your modules which use floating point arithmetic with the /E option.

3) For the 8087 Version of the Emulator

Copy the library C87LIB.LIB onto M2REAL.LIB

Compile all your modules which use floating point arithmetic with the /E option.

4) For the Mixed Emulator

Copy the library M87LIB.LIB onto M2REAL.LIB

Compile all your modules which use floating point arithmetic with the /E option.

The difference between 1) and 3) is that in 1) the module Real is not referenced or used, since the code has been generated inline.
8.5.3 Accuracy of the Computations

For all the basic arithmetic operations on operands of type REAL, the 8087 coprocessor and the LOGITECH REAL Emulator yield the same results. To compute mathematical functions such as sine or cosine, the emulator uses the Chebyshev polynomial approximation. Because the 8087 coprocessor uses a different scheme for approximation, the results of mathematical functions may sometimes differ slightly. For all practical purposes, these differences between the 8087 and the emulator are not significant.

8.5.4 Memory Requirements

When you compile with the Coprocessor option, the compiler generates 8087 inline code and makes no reference to the emulator (module Reals). Module Reals is only linked into a program if some part of the program was compiled using the Emulator option. As a general rule, the memory requirements are larger for programs that use the emulator.

Table 8-1 lists the approximate memory requirements (code and data) for the different implementations of the modules Reals and MathLib0. All numbers are given in bytes.

<table>
<thead>
<tr>
<th>Reals</th>
<th>MathLib0</th>
<th>Reals and MathLib0</th>
</tr>
</thead>
<tbody>
<tr>
<td>8087 Inline Code</td>
<td>---</td>
<td>1700</td>
</tr>
<tr>
<td>8087 Version of Emulator</td>
<td>500</td>
<td>1700</td>
</tr>
<tr>
<td>Pure Emulator</td>
<td>2300</td>
<td>4200</td>
</tr>
<tr>
<td>Mixed Emulator</td>
<td>2800</td>
<td>6100</td>
</tr>
</tbody>
</table>

Table 8-1
8.5.5 Performance

Table 8-2 lists the time measured for 1000 executions of the addition, multiplication, and division of two REAL numbers. Subtraction and addition require the same amount of time. The last row lists the times measured for 1000 executions of a loop that performs once, each basic operation (addition, subtraction, etc.), each kind of comparison (equal, less than, etc.), and calls once, each of the functions provided by MathLib0.

All times are given in seconds. However, the times measured may differ from system to system. Table 8-2 allows for a relative comparison of the performance you can expect when choosing a particular alternative for REAL arithmetic in LOGITECH Modula-2. On the average, 8087 inline code is approximately ten times faster than full emulation of REAL arithmetic on the 8086.

<table>
<thead>
<tr>
<th></th>
<th>Addition</th>
<th>Multiplication</th>
<th>Division</th>
<th>Combination</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pure Emulator</td>
<td>1.0</td>
<td>1.2</td>
<td>2.1</td>
<td>61.0</td>
</tr>
<tr>
<td>8087 Inline Code</td>
<td>0.11</td>
<td>0.11</td>
<td>0.11</td>
<td>6.6</td>
</tr>
<tr>
<td>8087 Version of Emulator</td>
<td>0.22</td>
<td>0.22</td>
<td>0.22</td>
<td>8.2</td>
</tr>
<tr>
<td>Mixed Emulator (with 8087)</td>
<td>0.31</td>
<td>0.31</td>
<td>0.31</td>
<td>12.5</td>
</tr>
<tr>
<td>Mixed Emulator (without 8087)</td>
<td>1.1</td>
<td>1.3</td>
<td>2.2</td>
<td>63.0</td>
</tr>
</tbody>
</table>

Table 8-2
8.6 RTS

*LOGITECH Modula-2* Run-Time Support provides support to the application for error handling, arithmetic coprocessor use, interrupt handling, and language dependent facilities.

8.6.1 Organization

The main module of the *RTS* is RTSMain, which must always be linked to the application. Other modules of the *RTS* may be linked or not, depending on their needs.

<table>
<thead>
<tr>
<th>Module</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTSMain</td>
<td>Entrypoint Initialization</td>
</tr>
<tr>
<td>RTSError</td>
<td></td>
</tr>
<tr>
<td>RTSRealError</td>
<td>Error Checks</td>
</tr>
<tr>
<td>RTSLanguage</td>
<td>Special Language Constructs Support</td>
</tr>
<tr>
<td>RTSCoroutine</td>
<td></td>
</tr>
<tr>
<td>RTSInterrupt</td>
<td></td>
</tr>
<tr>
<td>RTSPriority</td>
<td>Interrupt and Process Management</td>
</tr>
<tr>
<td>RTSDevice</td>
<td></td>
</tr>
<tr>
<td>RTSIntProc</td>
<td>I/O Device Control</td>
</tr>
<tr>
<td>RTS87</td>
<td></td>
</tr>
<tr>
<td>RTSM87</td>
<td>Math Coprocessor Support</td>
</tr>
</tbody>
</table>
8.6.2 RTSMain

RTSMain is the entry point of any LOGITECH Modula-2 application. It initializes the run-time support, and then gives control to the application.

8.6.3 RTSError

The RTSError module will be present if run-time checks (e.g., range error checks) are requested at compile time. It provides an entry point for the different kinds of error. This module has also the task, in case of run-time error, of clearing the stack. RTSRealError does the same if REAL type is used.

8.6.4 Language Dependent Facilities

The RTSLanguage module provides the routines that are called in the case of dynamic parameter copy, (e.g., a parameter of type ARRAY OF CHAR). This module provides also support for special cases of CASE statements.

8.6.5 Interrupt Handling

The RTSPriority module is used if some modules of the application are priority modules. It provides routines to change and restore the priority mask of the application when a procedure is called that belongs to a priority module.

The RTSInterrupt module is needed when binding a multiprocess application. An IOTRANSFER call will refer to this module.

8.6.6 I/O Device Control

RTSDevice and RTSIntPROC let you get control on the processors interrupt vector. They also let you change and restore such vectors.

8.6.7 Arithmetic Coprocessors

RTS87 or RTSM87 provide support for the 80n87 arithmetic coprocessor.

RTS87 will be needed is some modules of the application have been compiled with the /C switch (for coprocessor use).

RTSM87 is used in the mixed Emulator/Coprocessor mode.
8.7 Graphics

The *LOGITECH Modula-2* library provides a graphics module `Graphics`. This module is provided with different implementations, depending on the graphics you want to use.

Currently the following adaptors are supported:

<table>
<thead>
<tr>
<th>Graphics Type</th>
<th>Object File</th>
</tr>
</thead>
<tbody>
<tr>
<td>CGA</td>
<td>GRAPHCGA.OBJ</td>
</tr>
<tr>
<td>Hercules</td>
<td>GRAPHHEC.OBJ</td>
</tr>
</tbody>
</table>

By default, GRAPHCGA.OBJ is copied onto GRAPHICS.OBJ. These object files are not part of any library, but are distributed as .OBJ files.
Chapter 9
Libraries

9.1 Library Search Strategy

The LOGITECH Modula-2 Compiler, Linker, Debuggers and the other Utilities automatically search for all referenced modules. The default search strategy can be modified by command options.

The following discussion uses PATH or PATHNAME as a synonym for drive and/or pathname. For example, C:\M2LIB\SYM is the PATHNAME for the file C:\M2LIB\SYM\STORAGE.SYM.

9.1.1 Default Names

The LOGITECH Modula-2 Compiler, Linker, Debuggers, M2MAKE, M2CHECK and M2DECODE construct default filename from module names. This is done by truncating the module name to the length of a DOS file name, and appending the appropriate extension (.SYM, .OBJ, etc.) to that name. This default filename is then used to find the corresponding file on all paths that are defined by the search strategy.
9.1.2 The Default Search Strategy

When a module is needed, several paths will be checked automatically to find the corresponding file. The search strategy, as described below, is applied by all LOGITECH Modula-2 Utilities that look for referenced modules.

- Since all utilities per default generate their output files into the current directory, the current directory is the first path used to find the needed file.

- If the file was not found in the current directory, the so-called master path is used to find the file. The master path is the path where the main (or master) input module for the currently executed utility comes from.

- If the file was not found in the master path, those paths defined by the environment variables are taken one after the other to look for the file. The environment variables are set by the user after his own fashion. A default setting for the default installation of the LOGITECH Modula-2 System has been provided:

```
SET M2SYM=%M2LIB\SYM
SET M2OBJ=%M2LIB\OBJ
SET M2REF=%M2LIB\REF
SET M2MAP=%M2LIB\MAP
SET M2LIB=%M2LIB\LIB
SET M2MOD=%M2LIB\MOD
SET M2DEF=%M2LIB\DEF
```

Each of these environment strings can denote a number of paths, separated by semicolons.

If the needed file was not found on one of the defined paths, the default search strategy will take effect.

Assume the following environment settings:

```
SET M2SYM=%M2LIB\SYM;MYLIB\SYM
SET M2OBJ=%M2LIB\OBJ;MYLIB\OBJ
SET M2REF=%M2LIB\REF;MYLIB\REF;
SET M2MAP=%M2LIB\MAP;MYLIB\MAP;
SET M2LIB=%M2LIB\LIB;MYLIB\LIB;
SET M2MOD=%M2LIB\MOD;MYLIB\MOD;
SET M2DEF=%M2LIB\DEF;MYLIB\DEF;
```
Calling the compiler with the following command will automatically take the indicated symbol files as input:

M2C \TESTPGMS\REALTEST
### 9.1.3 The Query Search Strategy

The query search strategy is always applied by the LOGITECH Modula-2 system when you are prompted to type in the (path and) file name of a library file. This can happen when a file is not found using the default search strategy, or when you specify the Query option when compiling.

When you see a prompt on the LOGITECH Modula-2 Compiler screen, several responses are available.

<table>
<thead>
<tr>
<th>What You Enter</th>
<th>What It Means</th>
</tr>
</thead>
<tbody>
<tr>
<td>Esc</td>
<td>Means &quot;no file&quot;. Use this to indicate that the file is not available. Depending on context, Esc may not let you complete the program.</td>
</tr>
<tr>
<td>←</td>
<td>Means the FILENAME should be constructed from the module name, and that the default search strategy (as explained above) will be applied.</td>
</tr>
<tr>
<td>FILENAME only</td>
<td>(no PATH name) The FILENAME will be used, but will still be searched for automatically according to the default search strategy.</td>
</tr>
<tr>
<td>PATH NAME only</td>
<td>The file name will be constructed from the module name, and searched for using the specified path. Only one attempt to open the file will be made.</td>
</tr>
<tr>
<td>Complete PATH and FILENAME.</td>
<td>Here too, only one attempt will be made to open the file.</td>
</tr>
</tbody>
</table>
### 9.2 Library .DEF Files

**ASCII**

DEFINITION MODULE ASCII;

(*
Symbolic constants for non-printing ASCII characters.
This module has an empty implementation.
*)

EXPORT QUALIFIED

nul, soh, stx, etx, eot, enq, ack, bel,
bs, ht, lf, vt, ff, cr, so, si,
dle, dcl, dc2, dc3, dc4, nak, syn, etb,
can, em, sub, esc, fs, gs, rs, us,
del,
EOL;

CONST

nul = 00C; soh = 01C; stx = 02C; etx = 03C;
eot = 04C; enq = 05C; ack = 06C; bel = 07C;
bs = 10C; ht = 11C; lf = 12C; vt = 13C;
ff = 14C; cr = 15C; so = 16C; si = 17C;
dle = 20C; dcl = 21C; dc2 = 22C; dc3 = 23C;
dc4 = 24C; nak = 25C; syn = 26C; etb = 27C;
can = 30C; em = 31C; sub = 32C; esc = 33C;
fs = 34C; gs = 35C; rs = 36C; us = 37C;
del = 177C;

CONST

EOL = 36C;

(*
- end-of line character

This (non-ASCII) constant defines the internal name of the end-of-line character. Using this constant has the advantage, that only one character is used to specify line ends (as opposed to cr/lf).

The standard I/O modules interpret this character and transform it into the (sequence of) end-of-line code(s) required by the device they support. See definition modules of 'Terminal' and 'FileSystem'.
*)

END ASCII.

223
BitBlockOps

DEFINITION MODULE BitBlockOps;

(* Bitwise operations on blocks.
Blocks are defined as a starting address and a size, i.e. the number of
bytes they hold.
In a block, the left or low bit is the low bit of the byte located
at (starting address); and the right or high bit is the high bit of the byte
located at (starting address + size - 1)
*)

FROM SYSTEM IMPORT ADDRESS;

PROCEDURE BlockAnd (destination, source: ADDRESS;
size : CARDINAL);

(* ANDs the block destination with the block source *)

PROCEDURE BlockOr (destination, source: ADDRESS;
size : CARDINAL);

(* Bitwise OR *)

PROCEDURE BlockXor (destination, source: ADDRESS;
size : CARDINAL);

(* Bitwise XOR *)

PROCEDURE BlockNot (block : ADDRESS;
size : CARDINAL);

(* Bitwise complement to 1 *)

PROCEDURE BlockShr (block : ADDRESS;
size : CARDINAL;
count : CARDINAL);

(* Shift Logical Right
shifts the bits in block to the right by the number of bits specified
in count. Zeros are shifted in on the left. *)

PROCEDURE BlockSar (block : ADDRESS;
size : CARDINAL;
count : CARDINAL);

(* Shift Arithmetic Right
shifts the bits in block to the right by the number of bits specified
in count. Bits equal to the original high order bit are shifted in
on the left, preserving the sign of the original value. *)

PROCEDURE BlockShl (block : ADDRESS;
size : CARDINAL;
count : CARDINAL);

(* Shift Left
shifts the bits in block to the left by the number of bits specified
in count. Zeros are shifted in on the right. *)

PROCEDURE BlockRor (block : ADDRESS;
size : CARDINAL;
count : CARDINAL);

(* Rotate Right
rotates block right by the number of bits specified in count *)
PROCEDURE BlockRol (block : ADDRESS;
    size   : CARDINAL;
    count  : CARDINAL);

(* Rotate Left
    rotates block left by the number of bits specified in count *)

END BitBlockOps.
DEFINITION MODULE BitByteOps;

(* Bitwise operations on bytes. 
   Bits in bytes are numbered from 0 to 7 *)

FROM SYSTEM IMPORT BYTE;

PROCEDURE GetBits (source : BYTE; 
   firstBit, lastBit : CARDINAL): BYTE;
   (* Extracts the bits of source from firstBit to lastBit and returns them 
      as a byte in which bit 0 correspond to the firstBit of the source. *)

PROCEDURE SetBits (VAR byte BYTE; 
   firstBit, lastBit: CARDINAL; 
   pattern BYTE); 
   (* Masks byte with pattern from firstBit to lastBit. The first 
      (lastBit - firstBit + 1 of pattern are used, with leading zeros if necessary. 
      Examples : To set the bits to 1, the pattern OFFH should be passed, 
      and to set the bits to 0, the pattern 0 should be passed. *)

PROCEDURE ByteAnd (left, right : BYTE): BYTE; (* Bitwise AND *)

PROCEDURE ByteOr (left, right : BYTE): BYTE; (* Bitwise OR *)

PROCEDURE ByteXor (left, right : BYTE): BYTE; (* Bitwise XOR *)

PROCEDURE ByteNot (byte : BYTE): BYTE; (* Bitwise complement to 1 *)

PROCEDURE ByteShr (byte : BYTE; 
   count : CARDINAL): BYTE; 
   (* Shift Logical Right 
      shifts the bits in byte to the right by the number of bits specified in count. 
      Zeros are shifted in on the left. *)

PROCEDURE ByteSar (byte : BYTE; 
   count : CARDINAL): BYTE; (* Shift Arithmetic Right 
   shifts the bits in byte to the right by the number of bits specified in count. 
   Bits equal to the original high order bit are shifted in on the left, 
   preserving the sign of the original value. *)

PROCEDURE ByteShl (byte : BYTE; 
   count : CARDINAL): BYTE; (* Shift Left 
   shifts the bits in byte to the left by the number of bits specified in count. 
   Zeros are shifted in on the right. *)
PROCEDURE ByteRor (byte : BYTE;
    count : CARDINAL): BYTE;
(* Rotate Right
   rotates byte right by the number of bits specified in count *)

PROCEDURE ByteRol (byte : BYTE;
    count : CARDINAL): BYTE;
(* Rotate Left
   rotates byte left by the number of bits specified in count *)

PROCEDURE HighNibble (byte : BYTE): BYTE;
(* Returns the high order nibble (4 bits) value of byte *)

PROCEDURE LowNibble (byte : BYTE): BYTE;
(* Returns the low order nibble (4 bits) value of byte *)

PROCEDURE Swap (VAR byte : BYTE);
(* Swaps the high and low order nibble values of byte *)

END BitByteOps.
DEFINITION MODULE BitWordOps;

(* Bitwise operations on words.
  Bits in words are numbered from 0 to 15 *)

FROM SYSTEM IMPORT WORD;

PROCEDURE GetBits (source : WORD;
  firstBit, lastBit : CARDINAL): WORD;
(* Extracts the bits of source from firstBit to lastBit and returns them
  as a word in which bit 0 correspond to the firstBit of the source. *)

PROCEDURE SetBits (VAR word :
  firstBit, lastBit: CARDINAL;
  pattern : WORD);
(* Masks word with pattern from firstBit to lastBit. The first
(lastBit - firstBit + 1 of pattern are used, with leading zeros
if necessary.
Examples: To set the bits to 1, the pattern 0FFFFH should be passed,
and to set the bits to 0, the pattern 0 should be passed. *)

PROCEDURE WordAnd (left, right : WORD): WORD;
(* Bitwise AND *)

PROCEDURE WordOr (left, right : WORD): WORD;
(* Bitwise OR *)

PROCEDURE WordXor (left, right : WORD): WORD;
(* Bitwise XOR *)

PROCEDURE WordNot (word : WORD): WORD;
(* Bitwise complement to 1 *)

PROCEDURE WordShr (word :
  count : CARDINAL): WORD;
(* Shift Logical Right
  shifts the bits in word to the right by the number of bits specified
  in count. Zeros are shifted in on the left. *)

PROCEDURE WordSar (word :
  count : CARDINAL): WORD;
(* Shift Arithmetic Right
  shifts the bits in word to the right by the number of bits specified in count.
  Bits equal to the original high order bit are shifted in on the left,
  preserving the sign of the original value.
*)

PROCEDURE WordShl (word :
  count : CARDINAL): WORD;
(* Shift Left
  shifts the bits in word to the left by the number of bits specified in count.
  Zeros are shifted in on the right.
*)
PROCEDURE WordRor (word : WORD;
    count : CARDINAL): WORD;
(* Rotate Right
    rotates word right by the number of bits specified in count *)

PROCEDURE WordRol (word : WORD;
    count : CARDINAL): WORD;
(* Rotate Left
    rotates word left by the number of bits specified in count *)

PROCEDURE HighByte (word : WORD): WORD;
(* Returns the high order byte value of word *)

PROCEDURE LowByte (word : WORD): WORD;
(* Returns the low order byte value of word *)

PROCEDURE Swap (VAR word : WORD);
(* Swaps the high and low order bytes value of word *)

END BitWordOps.
DEFINITION MODULE BlockOps;

(* Block operations. 
Blocks are defined with a starting address and a size, i.e. the number 
of bytes they contain. *)

FROM SYSTEM IMPORT ADDRESS;

PROCEDURE BlockMoveForward (destination, source : ADDRESS; 
    size : CARDINAL);
(* Moves size bytes from source to destination, starting at the address 
of source and going up until address of (source+size) is reached *)

PROCEDURE BlockMoveBackward (destination, source : ADDRESS; 
    size : CARDINAL);
(* Moves size bytes from source to destination, starting at (source+size) 
and going down until address of source is reached *)

PROCEDURE BlockMove (destination, source : ADDRESS; 
    size : CARDINAL);
(* Moves size bytes from source to destination, test is made on the 
addresses of source and destination to decide whether MoveBackward or 
MoveForward is to be used. Note that because of this comparison, 
Move is slightly slower than the two previous procedures *)

PROCEDURE BlockClear (block : ADDRESS; 
    size : CARDINAL);
(* Fills size bytes with 0, starting from block. *)

PROCEDURE BlockSet (block : ADDRESS; 
    blockSize : CARDINAL; 
    pattern : ADDRESS; 
    patternSize : CARDINAL);
(* Fills blockSize bytes starting from block with the pattern of 
patternSize bytes. *)

PROCEDURE BlockEqual (block1, block2 : ADDRESS; 
    size : CARDINAL): BOOLEAN;
(* Returns TRUE if the blocks starting at left and right have the same 
first size bytes. *)

PROCEDURE BlockPosition (block : ADDRESS; 
    blockSize : CARDINAL; 
    pattern : ADDRESS; 
    patternSize : CARDINAL): CARDINAL;
(* Searches pattern in block, returns the index of the first successful 
match, MaxCard if no match. *)

END BlockOps.
Break

DEFINITION MODULE Break;
(*
   Handling of the Ctrl-Break interrupt
*)

This module provides an interrupt handler for the Ctrl-Break interrupt IBH of MS-DOS and PC-DOS on the IBM-PC. This module depends on the ROM BIOS of the IBM-PC and will not run on any machine which is not compatible to an IBM-PC at this level.

Module 'Break' installs a default break handler, which stops the execution of the current program with 'System.Terminate(stopped)' when Ctrl-Break is typed. This produces a memory dump for the stopped program.

Module 'Break' allows a program to install its own break handler, and to enable or disable the break handler which is currently installed.

EXPORT QUALIFIED
   EnableBreak, DisableBreak, InstallBreak, UnInstallBreak;

PROCEDURE EnableBreak;
(*
   Enable the activation of the current break handler
   If Ctrl-Break is detected, the currently installed break handler will be called.
*)

PROCEDURE DisableBreak;
(*
   Disable the activation of the current break handler
   If a Ctrl-Break is detected, no action takes place. The Ctrl-Break is ignored.
*)

PROCEDURE InstallBreak (BreakProc: PROC);
(*
   Install a break handler
   in: BreakProc break procedure to be called upon Ctrl-Break

   A program can install its own break handler. Module 'Break' maintains a stack of break procedures. The break procedure on top of the stack (i.e. the one which was installed most recently) will be called upon the occurrence of a ctrl-break. The default break handler which is installed initially terminates the program with a call to 'System.Terminate(stopped)'.

   Up to four user defined break procedure may be installed at the same time.
*)
PROCEDURE UnInstallBreak;
(*
   - Uninstall the current break handler

   Removes the break procedure which is currently on top of the stack. So the last
   installed break procedure will be deactivated, and the one installed previously
   becomes active again.
*)

END Break.
DEFINITION MODULE Calendar;

(* This module defines a Date type and operations on dates of
the Gregorian Calendar, introduced in 1582 *)

FROM DurationOps IMPORT
  Duration, Unit, UnitSet;

FROM TimeDate IMPORT
  Time;

TYPE Date =
  RECORD
    year : CARDINAL;
    month : [1..12];
    day : [1..31];
    hour : [0..23];
    minute : [0..59];
    second : [0..59];
    thousandth: [0..999];
  END; (* Date *)

PROCEDURE GetMachineDate (VAR date: Date);
(* Gets the machine date *)

PROCEDURE SetMachineDate (date: Date);
(* Sets the machine date *)

PROCEDURE TimeToDate (time: Time;
  VAR date: Date);
(* Type conversion from Time (in TimeDate) to Date (in Calendar) *)

PROCEDURE DateToTime (date: Date;
  VAR time: Time);
(* Type conversion from Date (in Calendar) to Time (in TimeDate) *)

PROCEDURE IsValid (date: Date): BOOLEAN;
(* Returns TRUE if date is valid, according to the Gregorian calendar *)

PROCEDURE DaysIn (month: CARDINAL;
  year : CARDINAL): CARDINAL;
(* Returns the number of days in the month of the year, according to
the Gregorian calendar, 0 if month is out of range. *)

PROCEDURE LeapYear (year: CARDINAL): BOOLEAN;
(* Returns TRUE if year is a leap year, according to the Gregorian
calendar (year number divisible by 400 or by 4 and not by 100) *)

PROCEDURE SameDate (date1, date2: Date;
  precision : Unit): BOOLEAN;
(* Returns TRUE if date1 and date2 are the same date, within precision *)
PROCEDURE Later (date1, date2 : Date;
    precision : Unit) : BOOLEAN;
(* Returns TRUE if date1 comes after date1, within precision *)

PROCEDURE LaterOrSameDate (date1, date2 : Date;
    precision : Unit) : BOOLEAN;
(* Returns TRUE if date2 is after date1 or if date1 and date2 are the same
date, within precision *)

(* The following operations give good results only with dates following
October 15, 1582, when the Gregorian Calendar was first used.
Accuracy to the second over long periods cannot be achieved, due to fluctuations
in the Earth rotation that often cause annual corrections of one second.
*)

PROCEDURE AddToDate (date : Date;
    duration : Duration;
    VAR resultDate : Date);
(* Add a duration to a date, gives a new date *)

PROCEDURE SubToDate (date : Date;
    duration : Duration;
    VAR resultDate : Date);
(* Subtract a duration from a date, gives a new date *)

PROCEDURE DeltaDate (date1, date2 : Date;
    unitFormat : UnitSet;
    VAR duration : Duration);
(* Absolute value of the difference between two dates, given a duration
with units in unitFormat (see module Duration) *)

END Calendar.
CardinalIO

DEFINITION MODULE CardinalIO;
(*
   Terminal input/output of CARDINALs in decimal and hex

   Derived from the Lilith Modula-2 system developed by the group of Prof. N. Wirth
   at ETH Zurich, Switzerland.
*)

EXPORT QUALIFIED
   ReadCardinal, WriteCardinal, ReadHex, WriteHex;

PROCEDURE ReadCardinal (VAR c: CARDINAL);
(*
   - Read an unsigned decimal number from the terminal.
     out: c the value that was read.

     The read terminates only on ESC, EOL, or blank, and the terminator
     must be re-read, for example with Terminal.Read.

     If the read encounters a non-digit, or a digit which would cause the number to
     exceed the maximum CARDINAL value, the bell is sounded and that character is
     ignored. No more than one leading '0' is allowed.
*)

PROCEDURE WriteCardinal (c: CARDINAL; w: CARDINAL);
(*
   - Write a CARDINAL in decimal format to the terminal.
     in: c value to write,
     w minimum field width.

     The value of c is written, even if it takes more than w digits. If it takes
     fewer digits, leading blanks are output to make the field w characters wide.
*)

PROCEDURE ReadHex (VAR c: CARDINAL);
(*
   - Read a CARDINAL in hexadecimal format from the terminal.
     [see ReadCardinal above]
*)

PROCEDURE WriteHex (c: CARDINAL; digits: CARDINAL);
(*
   - Write a CARDINAL in hexadecimal format to the terminal.
     [see WriteCardinal above]
*)

END CardinalIO.
Chapter 9

Chronometer

DEFINITION MODULE Chronometer;

(* Management and use of 'Chrono' objects, which permits to measure times
with an estimated accuracy of 0.02 second.
All the operations on these chronos are similar to those on a real chronometer.
*)

FROM DurationOps IMPORT
Duration, (* The measured time will be of this type *)
UnitSet; (* The units to represent the time. *)

TYPE
Chrono;

PROCEDURE NewChrono (VAR chrono : Chrono);
(* Creates a new variable of type Chrono ('Takes a chrono'), and resets it.
A call to NewChrono is mandatory before any other operation, otherwise
the program will be HALTed at any call of such an operation.
*)

PROCEDURE DisposeChrono (VAR chrono : Chrono);
(* Destroys variable of type Chrono ('Drops the chrono') It is illegal to call
any operation with chrono as parameter other than NewChrono
after a call to DisposeChrono.
*)

PROCEDURE StartChrono (chrono : Chrono);
(* Starts the chrono.
The chrono begins to measure elapsing time.
*)

PROCEDURE ReadChrono (chrono : Chrono;
format : UnitSet;
VAR elapsedTime : Duration);
(* Reads the chrono, without stopping it.
If format is empty then elapsedTime will be in seconds.
A chrono can be read several times, elapsedTime holds the time elapsed since
the last StartChrono of this chrono.
Accuracy : 0.02 second
*)

PROCEDURE StopChrono (chrono : Chrono);
(* Stops the chrono.
The time elapsing after a call to StopChrono is not taken in account.
*)

PROCEDURE ResetChrono (chrono : Chrono);
(* Stops and Resets the chrono.
After a call to Reset the chrono is prepared to measure times from zero.
Reset is automatically called by NewChrono.
*)

END Chronometer.
Conversions

DEFINITION MODULE Conversions;
(*
  Convert from INTEGER and CARDINAL to string

  Derived from the Lilith Modula-2 system developed by the group of Prof. N. Wirth
  at ETH Zurich, Switzerland.
*)

EXPORT QUALIFIED
ConvertOctal, ConvertHex,
ConvertCardinal, ConvertInteger, ConvertLongInt;

PROCEDURE ConvertOctal(num, len: CARDINAL;
  VAR str: ARRAY OF CHAR);
(*
  - Convert number to right-justified octal representation

  in:  num    value to be represented,
        len    minimum width of representation,
  out: str    result string.

  If the representation of 'num' uses fewer than 'len' digits, blanks are added
  on the left. If the representation will not fit in 'str', it is truncated
  on the right.
*)

PROCEDURE ConvertHex(num, len: CARDINAL;
  VAR str: ARRAY OF CHAR);
(*
  - Convert number to right-justified hexadecimal representation.
    [see ConvertOctal]
*)

PROCEDURE ConvertCardinal(num, len: CARDINAL;
  VAR str: ARRAY OF CHAR);
(*
  - Convert a CARDINAL to right-justified decimal representation.
    [see ConvertOctal]
*)

PROCEDURE ConvertInteger(num: INTEGER;
  len: CARDINAL;
  VAR str: ARRAY OF CHAR);
(*
  - Convert an INTEGER to right-justified decimal representation.
    [see ConvertOctal]

    Note that a leading '-' is generated if num < 0, but never a '+'.
*)
PROCEDURE ConvertLongInt(num : LONGINT;
    len : CARDINAL;
    VAR str : ARRAY OF CHAR);
(*
   - Convert a LONGINT to right-justified decimal representation.
     [see ConvertOctal]

   Note that a leading '-' is generated if num < 0, but never a '+'.
*)

END Conversions.
DateFormat

DEFINITION MODULE DateFormat;

(* Conversion between Date (from Calendar) and string types.

An internal format, called current format holds the template of a string,
  i.e. the way in which a date is represented. Routines are provided to
change this format, as a whole or field by field. *)

FROM Calendar IMPORT Date;

TYPE
  Format;

  Order = (DateOnly, /* Select Date and/or Time, and the */
            DateAndTime, /* order in which they are represented. */
            TimeOnly,
            TimeAndDate);

  DayFormat = (European, /* day month year */
              US, /* month day year */
              ISO); /* year month day */

  YearFormat = (Short, /* 87 */
                Long); /* 1987 */

  MonthFormat = (InDigits, /* 03 */
                 InLetters); /* March, Mars, ... */

  MonthName = ARRAY [0 .. 15] OF CHAR; /* Holds the months names, can */
             (* be changed by user. *)

  MonthList = ARRAY [1 .. 12] OF MonthName; /* Holds the months names, can */
             (* be changed by user. *)

  HourFormat = (PMSec, /* 1:17:05 pm */
                PMNoSec, /* 1:17 pm */
                H24Sec, /* 13:17:05 */
                H24NoSec); /* 13:17 */

  SeparatorList = ARRAY [0 .. 5] OF CHAR; /* Holds the separators of the */
                (* different date/time compo - *)
                (* components, can be changed *)
                (* by the user. *)

PROCEDURE DefaultFormat (): Format;
  (* Returns default date format :
    dd-mmm-yyyy hh:mm:ss i.e. 13-Jun-1987 17:45:30 *)

PROCEDURE CurrentFormat (): Format;
  (* Returns current date format *)

PROCEDURE SetFormat (format: Format);
  (* Sets the current format to format *)
PROCEDURE SetOrder (order: Order);
(* Sets the current format’s order to order."
   (default: DateAndTime) *)

PROCEDURE SetDayFormat (dayformat: DayFormat);
(* Sets the current format’s day format to dayFormat"
   (default: European) *)

PROCEDURE SetYearFormat (yearFormat: YearFormat);
(* Sets the current format’s year format to yearFormat"
   (default: Long) *)

PROCEDURE SetMonthFormat (monthFormat: MonthFormat);
(* Sets the current format’s month format to monthFormat"
   (default: InLetters) *)

PROCEDURE SetMonthList (monthList: MonthList);
(* Sets the current format’s month list to monthList"
   (default: Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec) *)

PROCEDURE SetHourFormat (hourFormat: HourFormat);
(* Sets the current format’s hour format to hourFormat"
   (default: H24Sec) *)

PROCEDURE SetSeparator (separator: SeparatorList);
(* Sets the current format’s list to separator"
   (default: "- ::") *)

PROCEDURE DateToString (date: Date;
   VAR image: ARRAY OF CHAR;
   VAR done: BOOLEAN);
(* Converts a Date in a string of current format *)

PROCEDURE StringToDate (image: ARRAY OF CHAR;
   VAR date: Date;
   VAR done: BOOLEAN;
   VAR errorPos: CARDINAL);
(* Converts a string in a date. The syntax of this string should be the one"
   defined by the current format, otherwise done is set to FALSE and errorPos to"
   the index of the first unexpected character of the string."
   *)

END DateFormat.
DebugPMD

DEFINITION MODULE DebugPMD;

END DebugPMD.
DebugTrace

DEFINITION MODULE DebugTrace;
(*
   (c) COPYRIGHT 1986,1987 LOGITECH SA, CH-1111 Romanel/Morges, Switzerland.

   Abstract : text
   Created : 19-MAR-87 Reuteler

   Modified : dd-mmm-yy Author Reason
*)

END DebugTrace.
Decimals

**DEFINITION**

MODULE Decimals;

(* Decimal Arithmetic *)

EXPORT QUALIFIED

DECIMAL, DecDigits, DecPoint, DecSep, DecCur, DecStatus,
DecState, DecValid, StrToDec, DecToStr, NegDec, CompareDec,
AddDec, SubDec, MulDec, DivDec, Remainder, DecRepr;

CONST

DecDigits = 18;
DecRepr = 10;
DecCur = '$';
DecPoint = '.';
DecSep = ',';

TYPE

DECIMAL = ARRAY [0..DecRepr-1] OF CHAR;

(* WARNING: Representation is implementation dependent! *)

DecState = (NegOvfl, Minus, Zero, Plus, PosOvfl, Invalid);

VAR

DecValid: BOOLEAN;
(* set after every operation *)

Remainder: CHAR;
(* remainder digit - set after DivDec *)

PROCEDURE StrToDec (String: ARRAY OF CHAR;
Picture: ARRAY OF CHAR;
VAR Dec: DECIMAL);

(* Converts a DECIMAL number from an external format to an internal format; after checking and matching between the picture and the input string. The result is placed in variable Dec. *)

PROCEDURE DecToStr (Dec: DECIMAL;
Picture: ARRAY OF CHAR;
VAR RsltStr: ARRAY OF CHAR);

(* Converts a DECIMAL number from an internal format to an external format; after checking and matching between the picture and the DECIMAL number. The result is placed in variable RsltStr. *)
PROCEDURE DecStatus (Dec: DECIMAL): DecState;
(*
Detects the state of the number represented as DECIMAL
and returns one of the following states:
- Negative overflow --> NegOvfl
- Negative --> Minus
- Null --> Zero
- Positive --> Plus
- Positive overflow --> PosOvfl
- Invalid representation --> Invalid
*)

PROCEDURE CompareDec (DecO, Dec1: DECIMAL): INTEGER;
(*
Compares two DECIMAL numbers and returns an integer value indicating the
comparison result:
-1 if DecO is less than Dec1
0 if DecO equals Dec1
1 if DecO is greater than Dec1
*)

PROCEDURE AddDec (Dec0, Dec1: DECIMAL; VAR Sum: DECIMAL);
(*
Adds two DECIMAL numbers (Dec0 and Dec1) together and places the result in the
variable Sum.
*)

PROCEDURE SubDec (Dec0, Dec1: DECIMAL; VAR Sub: DECIMAL);
(*
Subtracts Dec1 from Dec0 and places the result in Sub.
*)

PROCEDURE MulDec (Dec0, Dec1: DECIMAL; VAR Prod: DECIMAL);
(*
Multiplies two DECIMAL numbers and places the result in the variable Prod.
*)

PROCEDURE DivDec (Dec0, Dec1: DECIMAL; VAR Quot: DECIMAL);
(*
Dec0 is divided by Dec1. The quotient is placed in the variable Quot and the
remainder is placed in the global variable Remainder.
*)

PROCEDURE NegDec (Dec: DECIMAL; VAR NDec: DECIMAL);
(*
The negative DECIMAL value of Dec is placed in the variable NDec.
*)

END Decimals.
DEFINITION MODULE Delay;

EXPORT QUALIFIED
  Delay;

PROCEDURE Delay(milliSec: INTEGER);
  (*
     Interrupts the program execution for approximately 'milliSec' milliseconds.
     *)

END Delay.
DEFINITION MODULE Devices;
(*
  Additional facilities for device and interrupt handling

The MODULA-2/86 run-time support maintains a device mask that indicates from which
devices interrupts are enabled. The bits of the device mask have the same meaning
as the bits in the mask register of the interrupt controller.

Module 'Devices' provides access to the device mask. It allows a program to
inquire and change the status of a device (interrupts enabled or disabled). The
device numbers used by module 'Devices' and by the run-time support are equal to
the number of the bit in the device mask, that indicates whether interrupts from
this device are enabled or disabled.

When a program is running at no priority, the mask register of the interrupt
controller is identical to this device mask. When a program is running at some
priority, then the mask register of the interrupt controller is set to the logical
OR of the device mask and the corresponding priority mask. When the priority or
the device mask changes, the MODULA-2/86 run-time support sets the mask register
of the interrupt controller accordingly. At any point in time, all the interrupts
masked out, either in the device mask or in the current priority mask, are
disabled. The priority mask for 'no priority' does not mask out any interrupt, i.e. its value is all zeros.

When writing interrupt handlers in MODULA-2/86, it is strongly recommended to use
only the procedures provided by module 'Devices', and not to access directly the
mask register of the interrupt controller.

The following should be performed in order to install an interrupt handler: First
save the old interrupt vector, then set up the interrupt handler (IOTRANSFER), and
if necessary, save the current device status (interrupts enabled or disabled) and
enable interrupts from the device.

Before the program terminates, or in order to remove an interrupt handler, the
following sequence of procedure calls should be performed: If necessary, restore
the old device status or disable interrupts from the device, and then restore the
old interrupt vector.

At the end of a program the MODULA-2/86 run-time support resets the mask register
of the interrupt controller to its initial value.

In general, a call to IOTRANSFER in Modula-2 associates a process with only the
next occurrence of the specified interrupt. The procedure 'InstallHandler' provided by module 'Devices' allows to install an interrupt handler permanently.
It associates a process, the interrupt handler, permanently with a certain
interrupt.

While it is not required to install an interrupt handler in this way, it may be
useful for handling time critical interrupts. Installing an interrupt handler
permanently improves the performance of IOTRANSFER and of the implicit coroutine
transfer that takes place when the interrupt occurs by about 20 percent.

'InstallHandler' must only be called after the process has been created (by means
of NEWPROCESS) and before the process has called IOTRANSFER. For instance, it may
be called right at the beginning of the code of the process.
*)
FROM SYSTEM IMPORT ADDRESS, PROCESS;

EXPORT QUALIFIED
  GetDeviceStatus, SetDeviceStatus,
  SaveInterruptVector, RestoreInterruptVector,
  InstallHandler, UninstallHandler;

PROCEDURE GetDeviceStatus(deviceNr: CARDINAL;
  VAR enabled: BOOLEAN);

{*
  - Return the status of a device in the device mask
  in:  deviceNr number of the device to be checked bitnumber (0..7)
       of bit for device in interrupt controller 8259 mask
  out: enabled  TRUE if interrupts from the device are enabled, FALSE otherwise
  *}

PROCEDURE SetDeviceStatus(deviceNr: CARDINAL;
  enable: BOOLEAN);

{*
  - Set the status of a device in the device mask
  in:  deviceNr number of the device to enable or disable bitnumber (0..7)
       of bit for device in interrupt controller 8259 mask
       enable if TRUE, enable interrupts from the device,
       otherwise disable them
  The mask register of the interrupt controller will be updated according to the
  current priority and the new device mask.
  *}

PROCEDURE SaveInterruptVector (vectorNr : CARDINAL;
  VAR vector : ADDRESS);

{*
  - Save the current value of an interrupt vector
  in:  vectorNr  interrupt vector number
  out:  vector  value of current interrupt vector
  *}

PROCEDURE RestoreInterruptVector (vectorNr: CARDINAL;
  vector : ADDRESS);

{*
  - Restore the value of an interrupt vector
  in:  vectorNr  interrupt vector number
         vector  value to restore (previously saved with 'SaveInterruptVector')
  *}
PROCEDURE InstallHandler (process : PROCESS;
   vectorNr: CARDINAL);
(*
   - Install an interrupt handler permanently
   in:   process   process associated with the interrupt handler
         vectorNr interrupt vector number

   The process is associated permanently to the given interrupt vector number. This
   improves the performance of IOTRANSFER and of the implicit coroutine transfer that
   takes place when the interrupt occurs. A process may be associated to at most one
   interrupt, and at most one process may be associated to the same interrupt.

   'InstallHandler' must only be called after the process has been created (by means
   of NEWPROCESS) and before the process has called IOTRANSFER. For instance, it may
   be called right at the beginning of the code of the process. Except for the call
   to 'InstallHandler', the code of a permanently installed interrupt handler is
   identical to the code of a regular interrupt handler.
   *)

PROCEDURE UninstallHandler(process: PROCESS);
(*
   - Uninstall an interrupt handler which has been installed permanently
   in:   process   process associated with the interrupt handler

   In general, there is no need to call this procedure. The MODULA-2/86 run-time
   support automatically uninstalls interrupt handlers upon termination of a
   (sub-)program.
   *)

END Devices.
Directories

DEFINITION MODULE Directories;
(*
 Additional directory operations
*)

EXPORT QUALIFIED
  DirQueryProc, DirResult, DirQuery,
  Delete, Rename;

TYPE
  DirQueryProc = PROCEDURE (ARRAY OF CHAR, VAR BOOLEAN);
  DirResult = (OK,
                ExistingFile, (* rename to existing name *)
                NoFile, (* file not found *)
                OtherError);

PROCEDURE DirQuery( wildFileName : ARRAY OF CHAR;
                     DirProc    : DirQueryProc;
                     DirResult  : DirResult);
(*
 - Apply the a procedure to all matching files
 in:  wildFileName  file name, wild-characters are allowed
       DirProc     procedure to be called for each file matching 'wildFileName'
 out: result  result of directory operation

'DirQuery' executes 'DirProc' on each file which satisfies the specification of
'wildFileName' where wild-characters are allowed. If no more files are found,
or as soon as 'DirProc' returns FALSE, the execution is stopped.

If an incorrect filename is passed, this may return a 'result <> OK', and
'DirProc' will not be called.

Possible results are OK, NoFile, or OtherError.
*)

PROCEDURE Delete (FileName : ARRAY OF CHAR;
                  VAR result : DirResult);
(*
 - Delete a file.
 in:   FileName  name of the file to delete
 out:  result  result of directory operation

Possible results are OK, or NoFile.
*)
PROCEDURE Rename( FromName : ARRAY OF CHAR;
ToName : ARRAY OF CHAR;
VAR result : DirResult);
(*
    Rename a file.
    in: FromName   name of the file to rename
        ToName     new name of the file
    out: result    result of directory operation
    Possible results are OK, NoFile, ExistingFile, or OtherError.
*)
END Directories.
**DiskDirectory**

**DEFINITION MODULE** DiskDirectory;

/* Interface to directory functions of the underlying OS

Derived from the Lilith Modula-2 system developed by the group of Prof. N. Wirth at ETH Zurich, Switzerland. */

**EXPORT QUALIFIED**

CurrentDrive, SelectDrive,
CurrentDirectory, ChangeDirectory,
MakeDir, RemoveDir,
ResetDiskSys, ResetDrive;

**PROCEDURE** CurrentDrive (VAR drive: CHAR);

/* - Returns the current default drive.

out: drive name of the default drive, given in character format (e.g. 'A'). */

**PROCEDURE** SelectDrive (drive: CHAR; VAR done: BOOLEAN);

/* - Set default drive.

in: drive name of drive to make default, specified in character format (e.g. 'A').

out: done TRUE if operation was successful.

The default drive will be used by all routines referring to DK: . */

**PROCEDURE** CurrentDirectory (drive: CHAR; VAR dir: ARRAY OF CHAR);

/* - Gets the current directory for the specified drive.

in: drive name of the drive, specified in character format (e.g. 'A');
blank or OC denotes the current drive.

out: dir current directory for that drive.

Because CP/M-86 does not support named directories, dir[0] will always be set to nul (OC) under CP/M-86. */

251
PROCEDURE ChangeDirectory (dir: ARRAY OF CHAR;
   VAR done: BOOLEAN);
(*
   - Set the current directory
   in:   dir drive and directory path name.
   out:  done TRUE if successful; FALSE if the directory does not exist.
   Because CP/M-86 does not support named directories, this function has no effect
   and 'done' returns always FALSE under CP/M-86.
*)

PROCEDURE MakeDir (dir : ARRAY OF CHAR;
   VAR done : BOOLEAN);
(*
   - Create a sub-directory
   in:   dir drive, optional pathname and name of sub-directory to create.
   out:   done TRUE if successful; FALSE if path or drive does not exist.
   Because CP/M-86 does not support named directories, this function has no effect
   and 'done' returns always FALSE under CP/M-86.
*)

PROCEDURE RemoveDir   (dir : ARRAY OF CHAR;
   VAR done : BOOLEAN);
(*
   - Remove a directory
   in:   dir drive and name of the sub-directory to remove.
   out:   done:  TRUE if successful; FALSE if directory does not exist.
   The specified directory must be empty, otherwise 'done' returns FALSE and the
   directory is not removed.
   Because CP/M-86 does not support named directories, this function has no effect
   and 'done' returns always FALSE under CP/M-86.
*)

PROCEDURE ResetDiskSys;
(*
   - MS-DOS or CP/M-86 disk reset
*)

PROCEDURE ResetDrive (d: CHAR): CARDINAL;
(*
   - CP/M-86 reset drive.
   in:   drive name of drive to make default, specified in
         character format (e.g. 'A').
   out:   returns always zero under CP/M-86
   Under DOS this function has no effect and returns always the value 255.
*)

END DiskDirectory.
DiskFiles

DEFINITION MODULE DiskFiles;
(*
 Interface to disk file functions of the underlying OS.
 [Private module of the MODULA-2/86 system.]

 The default drive 'DK:', and drives 'A:' through 'P:' are supported
 under DOS or CP/M-86. This driver provides buffering. The maximum number of open
 files is 12.

 Derived from the Lilith Modula-2 system developed by the group of Prof. N. Wirth
 at ETH Zurich, Switzerland.
 *)

FROM FileSystem IMPORT File;

EXPORT QUALIFIED
   InitDiskSystem, DiskFileProc, DiskDirProc;

PROCEDURE InitDiskSystem;
(*
 - Initialize mediums for further disk file operations

 This procedure has to be imported by FileSystem. This has the side-effect, that
 this module is referenced and will therefore be linked to the user program.
 *)

PROCEDURE DiskFileProc (VAR f: File);
(*
 - low-level interface for disk operations within a file

 This procedure is passed as a parameter to the procedure
 CreateMedium in FileSystem.
 *)

PROCEDURE DiskDirProc (VAR f : File;
 name : ARRAY OF CHAR);
(*
 - low-level interface for disk operations within a directory

 This procedure is passed as a parameter to the procedure
 CreateMedium in FileSystem.
 *)

END DiskFiles.
DEFINITION MODULE Display;
(*
   Low-level Console Output
   [Private module of the MODULA-2/86 system]

   Derived from the Lilith Modula-2 system developed by the group of Prof. N. Wirth
   at ETH Zurich, Switzerland.
*)

EXPORT QUALIFIED Write;

PROCEDURE Write (ch: CHAR);
(*
   - Display a character on the console.

   in:   ch    character to be displayed.

   The following codes are interpreted:

   ASCII.EOL (36C) = go to beginning of next line
   ASCII.ff  (14C) = clear screen and set cursor home
   ASCII.del (177C) = erase the last character on the left
   ASCII.bs  (10C) = move 1 character to the left
   ASCII.cr  (15C) = go to beginning of current line
   ASCII.lf  (12C) = move 1 line down, same column

   Write uses direct console I/O.
*)

END Display.
DEFINITION MODULE DOS3;
  (* Additional DOS 3.0 functions *)

FROM SYSTEM IMPORT
  BYTE, WORD, ADDRESS;

EXPORT QUALIFIED
  GetExtendedError,
  CreateTemporaryFile,
  CreateNewFile,
  LockUnlockFileAccess,
  GetProgramSegmentPrefix;

(* DOS 3.0 function 59H *)
PROCEDURE GetExtendedError(version : WORD;
  (* BX *)
  VAR extendedError : WORD;
  (* AX *)
  VAR errorCode : BYTE;
  (* BH *)
  VAR suggestedAction : BYTE;
  (* BL *)
  VAR locus : BYTE);
  (* CH *)

(* DOS 3.0 function 5AH *)
PROCEDURE CreateTemporaryFile(path : ADDRESS;
  (* DS:DX *)
  attribute : WORD;
  (* CX *)
  VAR errorCode : WORD;
  (* AX,CF *)
  VAR handle : WORD;
  (* AX,CF *)
  VAR pathAndName : ADDRESS);
  (* DS:BX *)

(* DOS 3.0 function 5BH *)
PROCEDURE CreateNewFile(pathAndName : ADDRESS;
  (* DS:BX *)
  attribute : WORD;
  (* CX *)
  VAR errorCode : WORD;
  (* AX,CF *)
  VAR handle : WORD);
(*) DOS 3.0 function 5CH *)
PROCEDURE LockUnlockFileAccess(lock : BYTE;
(* AL *)
handle : WORD;
(* BX *)
offsetHigh : WORD;
(* CX *)
offsetLow : WORD;
(* DX *)
lengthHigh : WORD;
(* SI *)
lengthLow : WORD;
(* DI *)
VAR errorCode : WORD);
(* AX,CF *)

(*) DOS 3.0 function 62H *)
PROCEDURE GetProgramSegmentPrefix(VAR PSPsegment : WORD);
(* BX *)

END DOS3.
DEFINITION MODULE DOS31;
(* Additional DOS 3.1 functions *)

FROM SYSTEM IMPORT
BYTE, WORD, ADDRESS;

EXPORT QUALIFIED
GetMachineName,
SetPrinterSetup,
GetPrinterSetup,
GetRedirectionListEntry,
RedirectDevice,
CancelRedirection;

(* DOS 3.1 function 5E00H *)
PROCEDURE GetMachineName(computerName
(* DS:DX *)
VAR nameNumberIndFlag : BYTE;
(* CH *)
VAR nameNumber : BYTE;
(* CL *)
VAR errorCode : WORD);

(* DOS 3.1 function 5E02H *)
PROCEDURE SetPrinterSetup(redirectionListIndex
(* BX *)
setupStringLength : WORD;
(* CX *)
setupBuffer
(* DS:SI *)
VAR errorCode
(* AX,CF *)

(* DOS 3.1 function 5E03H *)
PROCEDURE GetPrinterSetup(redirectionListIndex
(* BX *)
ssetupBuffer
(* ES:DI *)
VAR setupStringLength : WORD;
(* CX *)
VAR errorCode
(* AX,CF *)

257
Chapter 9

(* DOS 3.1 function 5F02H *)
PROCEDURE GetRedirectionListEntry
  (redirectionIndex : WORD;
   (* BX *)
   localDeviceName : ADDRESS;
   (* DS:SI *)
   networkName : ADDRESS;
   (* ES:DI *)
   VAR deviceStatusFlag : BYTE;
   (* BH *)
   VAR deviceType : BYTE;
   (* BL *)
   VAR storedParmValue : WORD;
   (* CX *)
   VAR errorCode : WORD);
   (* AX,CF *)

(* DOS 3.1 function 5F03H *)
PROCEDURE RedirectDevice(deviceType : BYTE;
   (* BL *)
   valueToSaveForCaller : WORD;
   (* CX *)
   deviceName : ADDRESS;
   (* DS:SI *)
   networkPath : ADDRESS;
   (* ES:DI *)
   VAR errorCode : WORD);
   (* AX,CF *)

(* DOS 3.1 function 5F04H *)
PROCEDURE CancelRedirection(deviceName : ADDRESS;
   (* DS:SI *)
   VAR errorCode : WORD);
   (* AX,CF *)

END DOS31.
DosError

DEFINITION MODULE DosError;

(* Get the DOS error message associated to an error code. *)

PROCEDURE GetErrorMessage (errorCode : CARDINAL;
    VAR errorMessage : ARRAY OF CHAR);

(* errorCode is an error number returned by DOS functions. 
   errorMessage is at most 40 character long *)

END DosError.


DOSMemory

(* This is an interface to the DOS memory allocation ( DOSCALL 48H, 49H, 4AH ) 
* The blocks are linked to the Modula-2 RunTime Support, thus they are known 
* by the system and dumped in case of error. *)

DEFINITION MODULE DOSMemory;
FROM SYSTEM IMPORT ADDRESS;
EXPORT QUALIFIED DOSAlloc, DOSDeAlloc, DOSAvail, DOSSetSize, DOSGetMaxSize;

PROCEDURE DOSAlloc( VAR a: ADDRESS; paraSize: CARDINAL );
(* Allocates a block of paraSize paragraphs : *)
(* a is the address of the block returned or NIL if the size *)
(* is not available or an error occured *)

PROCEDURE DOSDeAlloc( VAR a: ADDRESS; paraSize: CARDINAL );
(* DeAllocates a block previously allocated with DOSAlloc. The *)
(* paraSize passed must be the size given for allocate or setsize *)
(* a is set to the NIL value if DeAlloc succeds, not modified *)
(* an error occured. *)
(* NOTE: the address passed MUST BE the address returned by *)
(* DOSAlloc *)

PROCEDURE DOSAvail(): CARDINAL;
(* Function that returns the size ( in paragraphs ) of the largest *)
(* space available. *)

PROCEDURE DOSSetSize( a: ADDRESS; paraSize: CARDINAL; VAR errorCode: CARDINAL );
(* Sets the size of the block given to the new size given in paraSize. *)
(* The returned errorCode is : *)
(* 0 : No Error *)
(* 7 : memory control block destroyed *)
(* 8 : insufficient memory *)
(* 9 : incorrect block address *)
(* NOTE: the address passed MUST BE the address returned by DOSAlloc *)

PROCEDURE DOSGetMaxSize( a: ADDRESS ): CARDINAL;
(* Gets the maximal paragraph size to which the block given as *)
(* parameter can be extended *)
(* NOTE: the address passed MUST BE the address returned by DOSAlloc *)

END DOSMemory.
DEFINITION MODULE DurationOps;
(* This module defines a Duration type and the relevant units. It allows comparisons, addition and subtraction on the Duration type, and a way to do conversion between units with ease. *)

TYPE Unit = (Millenium, Century, Year, Month, Day, Hour, Minute, Second, Tenth, Hundredth, Thousandth);
(* Year = mean solar time year: 365 days 5 hours 49 minutes 12 seconds 31 556 952 seconds
Month = Year / 12 : 2 629 746 seconds *)

TYPE Duration = ARRAY Unit OF REAL;
(* Each cell of this array will hold the real amount of the relevant unit. *)

TYPE UnitSet = SET OF Unit;

CONST
FullUnitSet = UnitSet {Millenium, Century, Year, Month, Day, Hour, Minute, Second, Tenth, Hundredth, Thousandth};
EmptyUnitSet = UnitSet {};

PROCEDURE Clear (VAR duration : Duration);
(* Set duration to zero *)

PROCEDURE Format (VAR duration : Duration; format : UnitSet);
(* Formatting of duration in format. Allows conversions between duration units. Unit cells of duration not in format are set to 0.0. Those in format are set to the greatest possible 'integer' value, except for the smallest unit which contains the remainder which may not be integer. If format is empty, duration is reformatted with the same units. *)

PROCEDURE FormatOf (duration: Duration): UnitSet;
(* Returns the format of duration, i.e. the set of the non zero unit cells. *)

PROCEDURE Sum (left, right: Duration; format: UnitSet; VAR result: Duration);
(* Addition of left and right, result being formatted with format. If format is empty then result is formatted with the union of left and right formats *)
PROCEDURE Diff (left, right : Duration;
              format : UnitSet;
              VAR result : Duration);
(* Subtraction of left and right, result being formatted with format. If format
  is empty then result is formatted with the union of left and right formats *)
PROCEDURE Equal (left, right : Duration;
                 accuracy : Unit) : BOOLEAN;
(* Returns TRUE if left and right are equal within accuracy *)
PROCEDURE Greater (left, right : Duration;
                   accuracy : Unit) : BOOLEAN;
(* Returns TRUE if left is greater than right within accuracy *)
PROCEDURE GreaterOrEqual (left, right : Duration;
                            accuracy : Unit) : BOOLEAN;
(* Returns TRUE if left is greater or equal than right within accuracy *)
END DurationOps.
DynMem

DEFINITION MODULE DynMem;

FROM SYSTEM IMPORT ADDRESS;

EXPORT QUALIFIED InstallDynMem, Alloc, DeAlloc, Avail;

(* for all procedures below, the block address must be paragraph aligned *)
(* with offset 0 *)

PROCEDURE InstallDynMem( block : ADDRESS;
                        size : CARDINAL );
(* size is the size in bytes usable by DynMem and it must be < MaxInt *)

PROCEDURE Alloc( block : ADDRESS;
                VAR adr : ADDRESS;
                size : CARDINAL );
(* adr will be the allocated block address or NIL if no space available *)
(* size is in bytes *)

PROCEDURE DeAlloc ( block : ADDRESS;
                  VAR adr : ADDRESS;
                  size : CARDINAL ): BOOLEAN;
(* adr return value will be NIL *)

PROCEDURE Avail ( block : ADDRESS;
                size : CARDINAL ): BOOLEAN;
(* returns TRUE if size is available in the block *)

END DynMem.
DEFINE MODULE ErrorCode;

(*
 handle return code to operating system
 *)

EXPORT QUALIFIED
  SetErrorCode, GetErrorCode, ExitToOS;

PROCEDURE SetErrorCode(value: CARDINAL);

(*
  Sets the error return code that will be used on normal termination; but it doesn't terminate the program immediately.
 *)

PROCEDURE GetErrorCode(VAR value: CARDINAL);

(*
  Allows to inspect the set return code
 *)

PROCEDURE ExitToOS;

(*
  Terminate current program and return to operating system. Set the error code corresponding to value defined by a previous call to SetErrorCode. Implementation restriction: if the program is using overlays, only the current overlay will be terminated.
 *)

END ErrorCode.
DEFINITION MODULE Exec;
FROM SYSTEM IMPORT ADDRESS;

EXPORT QUALIFIED
   DosShell, DosCommand, Run, Execute;

PROCEDURE DosShell(VAR done: BOOLEAN);
   (* call "COMMAND.COM"
   (* remain in DOS command shell, until user types EXIT
   (* finds COMMAND.COM through environment variable COMSPEC=

PROCEDURE DosCommand (command, parameters : ARRAY OF CHAR;
   VAR done: BOOLEAN);
   (* execute just one DOS command and return
   (* finds COMMAND.COM through environment variable COMSPEC=
   (* here, the DOS shell will perform a search strategy, 
   (* using the PATH= environment variable
   (* This call can be used to perform built in commands of 
   (* DOS (e.g. dir, ren, copy ...)

PROCEDURE Run (programFileName, parameters : ARRAY OF CHAR;
   VAR done : BOOLEAN);
   (* call program with parameters
   (* the complete filename with drive,
   (* path and extension has to be passed.
   (* no search strategy will be performed

PROCEDURE Execute (programFileNameAdr: ADDRESS;
   environment : CARDINAL;
   commandLineAdr : ADDRESS;
   FCB1Adr, FCB2Adr : ADDRESS;
   VAR errorCode : CARDINAL
   );
   (* call program with given parameter block information *
   (* no search strategy will be performed

END Exec.
Chapter 9

**FileMessage**

DEFINITION MODULE FileMessage;
(*
  Write file status/response to the terminal
*)
FROM FileSystem IMPORT Response;
EXPORT QUALIFIED WriteResponse;
PROCEDURE WriteResponse (r: Response);
(*
  - Write a short description of a FileSystem response on the terminal.
    
in:     r     the response from some FileSystem operation.
          
The actual argument for 'r' is typically the field 'res' of a variable of
          
type 'File'. The printed message is up to 32 characters long.
  *)
END FileMessage.
DEFINITION MODULE FileNames;
(*
   Read a file specification from the terminal.

   Derived from the Lilith Modula-2 system developed by the group of Prof. N. Wirth
   at ETH Zurich, Switzerland.
*)

EXPORT QUALIFIED
   FNParts, FNPartSet, ReadFileName;

TYPE
   FNParts = (FNDrive, FNPath, FNName, FNExt);
   FNPartSet = SET OF FNParts;

PROCEDURE ReadFileName (VAR resultFN : ARRAY OF CHAR;
                        defaultFN : ARRAY OF CHAR;
                        VAR ReadInName : FNPartSet);
(*
   - Read a file specification from terminal.

   in:  defaultFN     default file specification,
        ReadInName   which parts are in specification.

   out: resultFN      the file specification read,

   Reads until a <cr>, blank, <can>, or <esc> is typed. After a call to
   ReadFileName, Terminal.Read must be called to read the termination character.
   The format of the specification depends on the host operating system.
*)

END FileNames.
Chapter 9

FileSystem

**FileSystem**

**DEFINITION** MODULE FileSystem;

(*
  File manipulation routines

This implementation is based on the underlying operating system for file handling. It distinguishes between BINARY files and TEXT files.

File structure:

After any file operation the result should be checked for errors, by testing the field 'res' of the file variable (see type declarations for 'File' and 'Response').

The BOOLEAN field 'eof' in a file variable (variable of type 'File') allows to determine the end-of-file. It is set to TRUE after the first unsuccessful attempt to read information from the file. This first attempt to read beyond end-of-file does not set any error condition; the field 'res' of the file variable still indicates 'done'. However, the character (or other data) returned is not valid.

Binary files:

A file is a sequence of bytes with no other structure implied.

Under some operating systems (e.g. CP/M-86) the file may be organized in records (128 bytes each), and therefore, the length of a file will always be a multiple of this record size.

Text files:

A file is a sequence of characters. The character code 32C (Ctrl-Z) indicates end-of-file. All other character codes from 0C to 377C are legal. When reading a text file, 'eof' becomes TRUE when encountering the character 32C, or at the physical end of the file. When closing a text file that has been modified, the character 32C is written on the file.

When reading from a text file (by means of procedure 'ReadChar'), the character ASCII.EOL is returned for the sequence <CR, LF>, or for a single <CR> or <LF>. When writing to a text file (by means of procedure 'WriteChar'), the character ASCII.EOL is changed to the sequence <CR,LF>.

268
An open file is in one of the states 'opened', 'reading', 'writing', or 'modifying'. These states have the following meaning:

- **opened**: Content of file buffer is undefined and not associated with a position in the file. When starting to read or write from a file that is in state open, the state is changed implicitly to reading or writing.
- **reading**: No writing is allowed.
- **writing**: No reading is allowed. Writing always takes place at the end-of-file position. When writing on an existing file, which is (physically) longer than the current write position, it is undefined, whether the file is truncated upon a close.
- **modifying**: Reading and writing are allowed. Writing an element inside of a file means 'overwriting' the value of the element with a new value. Upon a close, the file is not truncated.

The state of the file is given by the field 'flags' of a file variable. By means of the procedures `SetRead`, `SetWrite`, `SetModify`, and `SetOpen`, it is possible to change the state of an open file.

To every file is associated a 'current position'. This corresponds to the number of the current byte inside the file, starting with zero for the first byte. The next reading or writing takes place at the current position. This position is updated automatically after reading or writing. It can also be inquired or set through the procedure `GetPos` or `SetPos`.

After the opening of a file (by means of `Lookup` or `Create`) it is state 'opened' and positioned at the beginning (low = 0, high = 0).

Conventions for filenames:

For the procedures `Lookup` and `Rename`, a filename has to be given, including a medium name (drive name), a file name and an optional file type. For the procedure `Create`, a medium name has to be given. The medium name is up to three characters long (alphanumeric, starting with a letter). It is separated from the file name by a colon (':'). If no medium name is given, the current default medium (drive) is assumed. The default medium may also be denoted by 'DK:'.

Depending on the operating system, the file name may include a path name, specifying the directory where the file exists. The length of the file (and path) name, and the characters legal for file names, depend on the operating system.

By default, the mediums (i.e. disk drives) handled by module 'DiskFiles' are installed.

Derived from the Lilith Modula-2 system developed by the group of Prof. N. Wirth at ETH Zurich, Switzerland.
FROM SYSTEM IMPORT ADDRESS, WORD, BYTE;

EXPORT QUALIFIED
File, Response, Command,
Flag, FlagSet,
(* basic file operations: *)
Create, Close, Lookup, Rename, Delete,
SetRead, SetWrite, SetModify, SetOpen,
DoIo,
SetPos, GetPos, Length,
(* stream-like I/O: *)
Reset, Again,
ReadWord, ReadChar, ReadByte, ReadNBytes,
WriteWord, WriteChar, WriteByte, WriteNBytes,
(* medium handling: *)
MediumType,
FileProc, DirectoryProc,
CreateMedium, RemoveMedium,
FileNameChar;

TYPE

MediumHint = CARDINAL;
(*- medium index used in DiskFiles *)

MediumType = ARRAY [0..2] OF CHAR;
(*- medium name (A, B...) *)

Flag = (er, ef, rd, wr, ag, txt);
(*
- status flag for file operations:
er = error occurred, ef = end-of-file reached,
rd = in read mode, wr = in write mode,
ag = "Again" has been called after last read,
txt = text-file (the last access to the file was a
'WriteChar' or 'ReadChar').
*)
FlagSet = SET OF Flag;
(*- status flag set *)

Response = (done, notdone, notsupported, callerror,
unknownmedium, unknownfile, paramerror,
toomanyfiles, eom, userdeverror);
(*
- result of a file operation

done:
    FileSystem routine successfully terminated
    notsupported:
    for internal purposes only
    callerror:
        a) You tried to write to a file currently in state reading. Use SetWrite
to change a file's state from reading to writing.
        b) You tried to read from a file currently in state writing. Use SetRead
to change a file's state from writing to reading.
        c) You tried to read from or write to a file marked as invalid by the
following operations:
            unsuccessful Create or Lookup
            successful Close or Delete
unknownmedium:
    The medium, or drive, which you addressed does not exist or is not known to
    the MODULA-2/86 System (it has not been installed by means of the
    CreateMedium routine).
unknownfile:
    The file you specified as the parameter for the Delete routine
could not be found.
paramerror:
    a) The syntax of the medium name, or drive name, which you specified
    is incorrect.
    b) When renaming a file, you must not change the medium name (drive name).
    c) You tried to position a file after its physical end.
toomanyfiles:
    Only 12 files can be opened at the same time.
eom:
    'end of medium' - The medium (disk) holding the file, to which you wanted
to write is short of storage space.
userdeverror:
    Not used in this implementation of the FileSystem.
notdone:
    a) You tried to read from a file for which the BOOLEAN field eof of the
    corresponding file variable is true.
    b) You tried to open a non-existing file with Lookup
    (with parameter newFile = FALSE).
    c) Any other error, not covered by the above meanings of the
    values of the FileSystem Response Type.
 *)

Command = (create, close, lookup, rename, delete,
    setread, setwrite, setmodify, setopen,
    doio, setpos, getpos, length);
(*- commands passed to module 'DiskFiles' *)

BuffAdd = POINTER TO ARRAY [0..OFFFEH] OF CHAR;
(*- file buffer pointer type *)
Chapter 9 FileSystem

File = RECORD
  bufa : BuffAdd;
  (*- buffer address *)
  buflen : CARDINAL;
  (*- size of buffer in bytes. In the current release it is always a multiple of 128. *)
  validlen : CARDINAL;
  (*- number of valid bytes in buffer *)
  bufind : CARDINAL;
  (*- byte-index to the buffer of the current position *)
  flags : FlagSet;
  (*- status of the file *)
  eof : BOOLEAN;
  (*- TRUE if last access was past the end of the file *)
  res : Response;
  (*- result of last operation *)
  lastRead : CARDINAL;
  (*- the word or byte (char) last read *)
  mt : MediumType;
  (*- selects the driver that supports this file *)
  fHint : CARDINAL;
  (*- used internally by device driver *)
  mHint : MediumHint;
  (*- used internally by medium handler *)
  CASE com : Command OF
    lookup : new : BOOLEAN;
    | setpos, getpos,
    | length: highpos, lowpos : CARDINAL;
  END;
END;
(*- file structure used for bookkeeping by DiskFiles *)

PROCEDURE Create (VAR f : File;
  mediumName : ARRAY OF CHAR);
(*
- create a temporary file
  in: mediumName name of medium to create file on, in character format
  out: f initialized file structure
A temporary file is characterized by an empty name. To make the file permanent, it has to be renamed with a non-empty name before closing it. For subsequent operations on this file, it is referenced by 'f'. *)

PROCEDURE Close (VAR f: File);
(*
- Close a file
  in: f structure referencing an open file
  out: f the field f.res will be set appropriately.
Terminates the operations on file "f". If "f" is a temporary file, it will be destroyed, whereas a file with a non-empty name remains on its medium and is accessible through "Lookup". When closing a text-file after writing, the end-of-file code 32C is written on the file (MS-DOS and CP/M-86 convention). *)
PROCEDURE Lookup (VAR f: File; fileName: ARRAY OF CHAR;
newFile: BOOLEAN);

(* - look for a file

in: filename drive and name of file to search for
newFile TRUE if file should be created if not found

out: f initialized file structure; f.res will be set appropriately.

Searches the medium specified in "filename" for a file that matches the name and type given in "filename". If the file is not found and "newFile" is TRUE, a new (permanent) file with the given name and type is created. If it is not found and "newFile" is FALSE, no action takes place and "notdone" is returned in the result field of "f".
*)

PROCEDURE Rename (VAR f: File; newname: ARRAY OF CHAR);

(* - rename a file

in: f structure referencing an open file
newname filename to rename to, with device:name.type specified

out: f file name in f will be changed and the field f.res will be set appropriately.

The medium, on which the files reside can not be changed with this command. The medium name inside "newname" has to be the old one.
*)

PROCEDURE Delete (name: ARRAY OF CHAR; VAR f: File);

(* - delete a file

in: name name of file to delete, with dev:name.type specified

out: f the field f.res will be set appropriately.
*)

PROCEDURE ReadWord (VAR f: File; VAR w: WORD);

(* - Returns the word at the current position in f

in: f structure referencing an open file

out: w word read from file
f the result field f.res will be set appropriately.

The file will be positioned at the next word when the read is done.
*)
PROCEDURE WriteWord (VAR f: File; w: WORD);
(*)
  - Write one word to a file
    in:   f     structure referencing an open file
          w     word to write
    out:  f     the field f.res will be set appropriately.

When overwriting, the file will be positioned at the next word when the write is done.
*)

PROCEDURE ReadChar (VAR f: File; VAR ch: CHAR);
(*)
  - Read one character from a file
    in:   f     structure referencing an open file
    out:  ch    character read from file
            f     the field f.res will be set appropriately.

ReadChar returns the character contained in the referenced file at the file's current position, with the following exceptions:

(The symbolic constants are from the Standard Library module ASCII.DEF)

<table>
<thead>
<tr>
<th>Character Sequence in File:</th>
<th>Character Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbolic</td>
<td>Octal</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>&lt;cr, lf&gt;</td>
<td>15C, 12C</td>
</tr>
<tr>
<td>&lt;cr&gt;</td>
<td>15C</td>
</tr>
<tr>
<td>&lt;lf&gt;</td>
<td>12C</td>
</tr>
<tr>
<td>&lt;Ctrl-z&gt;</td>
<td>32C</td>
</tr>
</tbody>
</table>

<Ctrl-z>, i.e. 32C, indicates end-of-file.

If ReadChar encounters the end of the file or tries to read beyond it, a nul character, or 0C, is returned.

The file will be positioned at the next character when the read is done.
*)
PROCEDURE WriteChar (VAR f: File; ch: CHAR);
/* - Write one character to a file
   in:  f structure referencing an open file
         ch character to write
   out:  f the field f.res will be set appropriately.

WriteChar writes the character to the referenced file at the file's current position, with the following exceptions:

(The symbolic constants are from the Standard Library module ASCII.DEF)

<table>
<thead>
<tr>
<th>Character to write</th>
<th>Character Sequence in File:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbolic</td>
<td>Octal</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>&lt;EOI&gt;</td>
<td>36C</td>
</tr>
<tr>
<td>&lt;cr&gt;</td>
<td>15C</td>
</tr>
<tr>
<td>&lt;lf&gt;</td>
<td>12C</td>
</tr>
<tr>
<td>&lt;Ctrl-z&gt;</td>
<td>32C</td>
</tr>
</tbody>
</table>

When overwriting, the file will be positioned at the next character when the write is done.

*)

PROCEDURE ReadByte (VAR f: File; VAR b: BYTE);
/* - Read one byte from a file
   in:  f structure referencing an open file
   out:  b byte read from file
         f the field f.res will be set appropriately.

The file will be positioned at the next byte when the read is completed.

*)

PROCEDURE WriteByte (VAR f: File; b: BYTE);
/* - Write one byte to a file
   in:  f structure referencing an open file
         b byte to write
   out:  f the field f.res will be set appropriately.

When overwriting, the file will be positioned at the next byte when the write is done.

*)
PROCEDURE ReadNBytes (VAR f: File;
bufPtr : ADDRESS;
requestedBytes : CARDINAL;
VAR read : CARDINAL);

(* - Read a specified number of bytes from a file

in:   f structure referencing an open file
bufPtr pointer to buffer area to read bytes into
requestedBytes number of bytes to read

out:   bufPtr^ bytes read from file
f the field f.res will be set appropriately.
read the number of bytes actually read.

The file will be positioned at the next byte after the requested sequence of bytes.
*)

PROCEDURE WriteNBytes (VAR f: File;
bufPtr : ADDRESS;
requestedBytes : CARDINAL;
VAR written : CARDINAL);

(* - Write a specified number of bytes to a file

in:   f structure referencing an open file
bufPtr pointer to string of bytes to write
requestedBytes number of bytes to write

out:   f the field f.res will be set appropriately.
written the number of bytes actually written

When overwriting, the file will be positioned at the next byte after the requested sequence of bytes.
*)

PROCEDURE Again (VAR f: File);

(* - returns a character to the buffer to be read again

in:   f structure referencing an open file

out:   f the f.res field will be set appropriately.

This should be called after a read operation only (it has no effect otherwise). It prevents the subsequent read from reading the next element; the element just read before will be returned a second time. Multiple calls to Again without a read in between have the same effect as one call to Again. The position in the file is undefined after a call to Again (it is defined again after the next read operation).
*)
PROCEDURE SetRead (VAR f: File);
 (*
- Sets the file in reading-state, without changing the current position.

 in:  f  structure referencing an open file
 out:  f  f.res will be set appropriately.

 Upon calling SetRead, the current position must be before the eof.
 In reading state, no writing is allowed.
*)

PROCEDURE SetWrite (VAR f: File);
 (*
- Sets the file in writing-state, without changing the current position.

 in:  f  structure referencing an open file
 out:  f  f.res will be set appropriately.

 Upon calling SetWrite, the current position must be a legal position in the file
 (including eof). In writing state, no reading is allowed, and a write always
 takes place at the eof. The current implementation does not truncate the file.
*)

PROCEDURE SetModify (VAR f: File);
 (*
- Sets the file in modifying-state, without changing the current position.

 in:  f  structure referencing an open file
 out:  f  f.res will be set appropriately.

 Upon calling SetModify, the current position must be before the eof. In
 modifying-state, reading and writing are allowed. Writing is done at the
 current position, overwriting whatever element is already there.
 The file is not truncated.
*)

PROCEDURE SetOpen (VAR f: File);
 (*
- Set the file to opened-state, without changing the current position.

 in:  f  structure referencing an open file
 out:  f  f.res will be set appropriately.

 The buffer content is written back on the file, if the file has been in writing
 or modifying status. The new buffer content is undefined. In opened-state,
 neither reading nor writing is allowed.
*)

PROCEDURE Reset (VAR f: File);
 (*
- Set the file to opened state and position it to the top of file.

 in:  f  structure referencing an open file
 out:  f  f.res will be set appropriately.
*)
PROCEDURE SetPos (VAR f: File; high, low: CARDINAL);
(*
- Set the current position in file
in:  f  structure referencing an open file
   high  high part of the byte offset
   low  low part of the byte offset
out:  f  f.res will be set appropriately.
The file will be positioned (high*2^16 + low) bytes from the top of file. *)

PROCEDURE GetPos (VAR f: File; VAR high, low: CARDINAL);
(*
- Return the current byte position in file
in:  f  structure referencing an open file
out:  high  high part of byte offset
      low  low part of byte offset
The actual position is (high*2^16 + low) bytes from the top of file. *)

PROCEDURE Length (VAR f: File; VAR high, low: CARDINAL);
(*
- Return the length of the file in bytes.
in:  f  structure referencing an open file.
out:  high  high part of byte offset
      low  low part of byte offset
The actual length is (high*2^16 + low) bytes. Depending on the operating system,
this length may always be a multiple of some record size reflecting the physical
length of the file and maybe not the true logical file length. *)

PROCEDURE Doio (VAR f: File);
(*
- Do various read/write operations on a file
in:  f  structure referencing an open file
out:  f  f.res will be set appropriately.
The exact effect of this command depends on the state of the file (flags):
opened  = NOOP,
reading  = reads the record that contains the current byte from the file. The
          old content of the buffer is not written back.
writing  = the buffer is written back. It is then assigned to the record, that
          contains the current position. Its content is not changed.
modifying = the buffer is written back and the record containing
          the current position is read.
Note that 'Doio' does not need to be used when reading through the
stream-like I/O routines. Its use is limited to special applications. *)
PROCEDURE FileNameChar (c: CHAR): CHAR;
(*
  - Check the character c for legality in a filename.
  in:  c     character to check
  out: DC for illegal characters and c otherwise;
       lowercase letters are transformed into uppercase letters.
Which characters are legal in a filename depends on the host operating system.
*)

TYPE
FileProc = PROCEDURE (VAR File);
(*
  - Procedure type to be used for internal file operations
    A procedure of this type will be called for the following functions
    (see TYPE 'Command'):
    setread, setwrite, setmodify, setopen, doio, setpos, getpos, and length.
  *)
DirectoryProc = PROCEDURE (VAR File, ARRAY OF CHAR);
(*
  - Procedure type to be used for operations on entire files
    A procedure of this type will be called for the following functions
    (see TYPE 'Command'):
    create, close, lookup, rename, and delete.
  *)

PROCEDURE CreateMedium (mt : MediumType;
                         fproc : FileProc;
                         dproc : DirectoryProc;
                         VAR done : BOOLEAN);
(*
  - Install the medium "mt" in the file system
  in:  mt     medium type to install
        fproc procedure to handle internal file operations
        dproc procedure to handle operations on an entire file
  out  done   TRUE if medium was installed successfully
Before accessing or creating a file on a medium, this medium has to be announced to the file system by means of the routine CreateMedium. FileSystem calls "fproc" and "dproc" to perform operations on a file of this medium. Up to 24 mediums can be announced.
*)
PROCEDURE RemoveMedium (mt: MediumType; VAR done: BOOLEAN);

(* - Remove the medium "mt" from the file system

in: mt medium type to remove

out: done TRUE if medium was removed successfully

Attempts to access a file on this medium result in an error (unknownmedium).
*)

END FileSystem.
DEFINITION MODULE FloatingUtilities;

EXPORT QUALIFIED
  Frac, Int, Round, Float, Trunc;

PROCEDURE Frac ( r : REAL ) : REAL;
(*
  Returns the fractional part of r, i.e. Frac( r ) = r + Int( r )
*)

PROCEDURE Int ( r : REAL ) : REAL;
(*
  Returns the integer part of r, i.e. the greatest integer number less than
  or equal to r, if r >= 0, or the smallest integer number greater than or
  equal to r, if r < 0.
*)

PROCEDURE Round ( num : REAL ) : INTEGER;
(*
  Returns the value of num rounded to the nearest integer as it follows :
  if num >= 0, then Round( num ) = TRUNC( num - 0.5 )
  num must be of type real, and result is of type integer.
*)

PROCEDURE Float ( int : INTEGER ) : REAL;

PROCEDURE Trunc ( real : REAL ) : INTEGER;

END FloatingUtilities.
Graphics

DEFINITION MODULE Graphics;
(* Graphics module *)

FROM SYSTEM IMPORT BYTE;

EXPORT QUALIFIED
(* colors *)
Black, Blue, Green, Cyan, Red, Magenta, Brown, LightGray,
DarkGray, LightBlue, LightGreen, LightCyan, LightRed,
LightMagenta, Yellow, White,

(* screen modes *)
txtMedRes, txtHiRes, txtCMedRes, txtCHiRes,
gphMedRes, gphCMedRes, gphHiRes,

(* screen control *)
ScreenMode, GetScreenMode, GetScreenExt,
Palette, ColorTable, ForegroundColor, BackgroundColor,

(* graphics *)
Window, GetWindow, ClearWindow, BackgroundPattern,
ClipDot, Dot, GetDotColor,
ClipLine, Line, Arc, Circle, Text,
FloodFill, FillRect, Pattern,
SavePicture, RestorePicture,

(* cursor control *)
cursorWidth, cursorHeight,
CURSORSHAPE, CURSORSHAPEPOINTER,
CursorShape, CursorColor, CursorWrap, CursorShow,
EraseCursor, DisplayCursor, MoveCursor,
GetCursorPosition, CursorVisible;

(* colors *)
CONST
(* Dark colors ; Light colors *)
Black = 0 ; DarkGray = 8 ;
Blue = 1 ; LightBlue = 9 ;
Green = 2 ; LightGreen = 10 ;
Cyan = 3 ; LightCyan = 11 ;
Red = 4 ; LightRed = 12 ;
Magenta = 5 ; LightMagenta = 13 ;
Brown = 6 ; Yellow = 14 ;
LightGray = 7 ; White = 15 ;

(* screen control *)
CONST
(* supported screen modes *)
txtMedRes = 0 ; (* text 40x25 monochrome medium resolution mode *)
txtCMedRes = 1 ; (* text 40x25 color medium resolution mode *)
txtHiRes = 2 ; (* text 80x25 monochrome high resolution mode *)
txtCHiRes = 3 ; (* text 80x25 color high resolution mode *)
gphCMedRes = 4 ; (* graphic 320x200 color medium resolution mode *)
gphMedRes = 5 ; (* graphic 320x200 monochrome medium resolution mode *)
gphHiRes = 6 ; (* graphic 640x200 monochrome high resolution mode *)
PROCEDURE ScreenMode (mode: INTEGER);
(*
  Sets the screen in the given mode. The screen is cleared.
  The supported text modes are the following:
  1. txtMedRes
     It activates the monochrome medium resolution text mode
     with 40x25 characters.
  2. txtCMedRes
     It activates the color medium resolution text mode
     with 40x25 characters.
  3. txtHiRes
     It activates the monochrome high resolution text mode
     with 80x25 characters.
  4. txtCHiRes
     It activates the color high resolution text mode
     with 80x25 characters.

  The supported graphic modes are the following:
  1. gphCMedRes
     It activates the 320x200 dots color graphics screen.
     x-coordinates are in a range between 0..319,
     y-coordinates are in a range between 0..199.
     The drawing colors may be selected with the procedure Palette.
  2. gphMedRes
     It activates the 320x200 dots monochrome graphics screen.
     x-coordinates are in a range between 0..319,
     y-coordinates are in a range between 0..199.
     If you have an RGB monitor (like the IBM Color/Graphics display),
     you can even use the colors from Palette(0) and Palette (1).
  3. gphHiRes
     It activates the 640x200 dots (high resolution)
     monochrome graphics screen.
     x-coordinates are in a range between 0..639,
     y-coordinates are in a range between 0..199.
     The background in the high resolution mode is always black.
     The drawing color may be selected by procedure ForegroundColor.
*)

PROCEDURE GetScreenMode (VAR mode: INTEGER);
(*
  Returns the current screen mode.
*)

PROCEDURE GetScreenExt (VAR x, y: INTEGER);
(*
  Returns the extension of the screen.
  If the screen is in a mode which is not supported,
  x, y are set to 0.
*)
PROCEDURE Palette (palette: INTEGER);
(*
Selects the current palette in gphC40 and gphBW40.
A change of the palette will cause everything on the screen to change to
the colors of the new palette.
Four palettes are available and for each palette there is a choice of four colors.

<table>
<thead>
<tr>
<th>Color number</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Palette(0)</td>
<td>Background</td>
<td>Green</td>
<td>Red</td>
<td>Brown</td>
</tr>
<tr>
<td>Palette(1)</td>
<td>Background</td>
<td>Cyan</td>
<td>Magenta</td>
<td>LightGray</td>
</tr>
<tr>
<td>Palette(2)</td>
<td>Background</td>
<td>LightGreen</td>
<td>LightRed</td>
<td>Yellow</td>
</tr>
<tr>
<td>Palette(3)</td>
<td>Background</td>
<td>LightCyan</td>
<td>LightMagenta</td>
<td>White</td>
</tr>
</tbody>
</table>
*)

PROCEDURE ColorTable (color1, color2, color3, color4: INTEGER);
(*
Defines the color translation table for subsequent drawings.
The given colors are colors of the palette.
All the drawing procedures use the color table if the color -1 is specified. The
SavePic procedure always uses the color table.
When a point has to be written on the screen and the color table is specified,
the point’s current color is used to index the color table.
The point is drawn in the color so obtained.
The default color table setting is (0,1,2,3).
That means that when a point is written on the screen,
it does not change the color which is already there.

The color table (0,1,2,3) means that:
  - color 0 becomes color 0,
  - color 1 becomes color 1,
  - color 2 becomes color 2,
  - color 3 becomes color 3.
The color table (3,2,1,0) means that:
  - color 0 becomes color 3,
  - color 1 becomes color 2,
  - color 2 becomes color 1,
  - color 3 becomes color 0.
*)

PROCEDURE ForegroundColor (color: INTEGER);
(*
Selects the foreground color in gphBW640 mode.
Changing the foreground color causes anything on the screen
to change to the new color.
The color constants defined above may be used.
*)

PROCEDURE BackgroundColor (color: INTEGER);
(*
Sets the background color in gphBW320 and gphC320 modes.
The color constants defined above may be used.
Color is an integer in the range 0..15.
*)
PROCEDURE Window (xl, y1, x2, y2: INTEGER);
(*
  Defines a window, that is the area on the screen where all the drawing occurs.
  The coordinates are absolute screen coordinates.
  The coordinates are clipped to the screen boundaries.
  If the specified window has no intersection with the screen,
  the new window is not defined. The previous window is still
  valid.
  The current window can be retrieved by using the procedure GetWindow.
  The point (xl, y1) is the upper left corner;
  the point (x2, y2) is the lower right corner of the window.
  The entire screen is the default graphic window 0,0,319,199
  in the 320x200 dot mode and 0,0,639,199 in the 640x200 dot mode.
  After defining a window, all the coordinates are relative to the window.
  The origin of the reference system is the upper left corner of the window.
*)

PROCEDURE GetWindow (VAR xl, y1, x2, y2: INTEGER);
(*
  Returns the coordinates of the window.
  (xl, y1) is the upper left corner and
  (x2, y2) is the lower right corner of the window.
*)

PROCEDURE BackgroundPattern (pat: ARRAY OF BYTE);
(*
  Defines the background pattern which is used by the ClearWindow procedure.
*)

PROCEDURE ClearWindow (color: INTEGER);
(*
  Fills the current window with the current background pattern in the given color.
  The colors 0..3 will be selected from the color palette;
  the color -1 will make use of the color table.
  The background pattern is defined with the procedure BackgroundPattern.
*)

PROCEDURE ClipDot (x, y: INTEGER): BOOLEAN;
(*
  Returns TRUE if the dot at coordinates (x, y) is inside the window.
*)

PROCEDURE Dot (x, y: INTEGER; color: INTEGER);
(*
  Plots a point at the screen coordinates x and y in the given color.
  If color = -1, the color table is used.
*)

PROCEDURE GetDotColor (x, y: INTEGER): INTEGER;
(*
  Returns the color value of the dot located at coordinate xpos, ypos.
  In the 320 x 200 dot graphic mode, values of 0..3 may be returned.
  In the 640 x 200 dot graphic mode, values of 0..1 may be returned.
  If the dot is outside the window, GetDotColor returns -1.
*)
PROCEDURE ClipLine (VAR x1, y1, x2, y2: INTEGER): BOOLEAN;
(*
Returns the in variables x1, y1 and x2, y2 the coordinates of the end points of the segment obtained by clipping the line at the window boundaries. The procedure also returns TRUE if at least a portion of the line lies in the window.
*)

PROCEDURE Line (x1, y1, x2, y2: INTEGER; color: INTEGER);
(*
Draws a straight line from (x1, y1) to (x2, y2) in the given color.
If the color is -1, the color will be obtained from the color table.
*)

PROCEDURE Arc (centerX, centerY, radius, startAngle, arcAngle, color: INTEGER);
(*
Draws a circular arc centered at (centerX, centerY) and with given radius. The starting position is given by startAngle and the extent of the arc is given by arcAngle.
startAngle and arcAngle are given in positive or negative degrees.
0 degrees is at 3 o’clock.
A positive angle goes counterclockwise, while a negative angle goes clockwise.
startAngle is treated mod 360.
arcAngle is in the range (-360, 360).
The arc is drawn in the given color.
If the color is -1, the color table will be used.
*)

PROCEDURE Circle (x, y, radius, color: INTEGER);
(*
Draws a circle with center at (x, y), with the given radius and in the given color.
In the 640 x 200 mode, the circle will appear as an ellipse.
If the color is -1, the color table will be used.
*)

PROCEDURE FloodFill (x, y, fillColor, borderColor: INTEGER);
(*
Fills an area on the display surface with the color specified by fillColor. The color table is not supported.
The area is bounded by the given borderColor.
(x, y) are the coordinates of any point within the area to be filled.
*)

PROCEDURE FillRect (x1, y1, x2, y2, color: INTEGER);
(*
Fills the rectangular area defined by the coordinates x1, y1, x2, y2 with the current pattern (see the procedure Pattern).
Bits set to 1 in the pattern cause a dot to be written in the given color; bits set to 0 cause no change to the display.
If color = -1, the color table is used.
*)

PROCEDURE Pattern (pattern: ARRAY OF BYTE);
(*
Defines the pattern used by the FillRect procedure.
Each byte corresponds to a horizontal line, each bit corresponds to a pixel.
*)
PROCEDURE Text (x, y: INTEGER; string: ARRAY OF CHAR; color: INTEGER);
(*
  Displays the given string at the given position.
The lower left corner of the first character in the string
is positioned at coordinates (x, y).
The width and height of a character is 8 pixels.
Clipping is performed at the window boundaries.
*)

PROCEDURE SavePicture (VAR buffer: ARRAY OF BYTE; x1, y1, x2, y2: INTEGER);
(*
  Saves the contents of a rectangular area on the screen into the variable buffer.
The rectangular area is defined by the coordinates (x1, y1), (x2, y2)
and it is clipped to the current graphic window.

The first 6 bytes of the buffer constitute a three word header.
The remaining bytes will contain the data.
The programmer has to ensure that the buffer is large enough
to accommodate the entire transfer.
The minimum buffer size in bytes is computed as following:
  320 x 200 modes:
    size = ((width + 3) div 4) * height * 2 + 6;
  640 x 200 modes:
    size = ((width + 7) div 8) * height + 6.

where:
x1, x2, y1, y2 have been clipped to the current graphic window;
width = abs(x1-x2) +1;
height = abs(y1-y2) +1;

After loading, the buffer has the following structure:
byte 0..1 : contains 2 in the 320 x 200 mode,
  contains 1 in the 640 x 200 mode;
byte 2..3 : width of the image;
byte 4..5 : height of the image;
byte 6... : data.

Data is stored with the leftmost pixels in the most significant bits of the bytes.
At the end of each row, the remaining bits of the last byte are skipped.
*)

PROCEDURE RestorePicture (VAR buffer: ARRAY OF BYTE; x, y: INTEGER);
(*
  Restores on the screen the contents of buffer (see SavePic).
The lower left corner of the picture is placed at (x, y).
*)

(* cursor data types and procedures *)

TYPE CURSORSHAPE = RECORD
  hotX : INTEGER;
  hotY : INTEGER;
  shape: ARRAY[0..7] OF BYTE;
END;

  CURSORSHAPEPOINTER = POINTER TO CURSORSHAPE;

  CONST
    cursorWidth = 7; (* cursor pattern width - 1 *)
    cursorHeight = 7; (* cursor pattern height - 1 *)
PROCEDURE CursorShape (shapePT: CURSORSHAPEPOINTER);
(*
  Selects the cursor shape.
  It does NOT redisplay the cursor.
*)

PROCEDURE CursorColor (color: INTEGER);
(*
  Selects the color in which the cursor will be drawn.
  It does NOT redisplay the cursor.
*)

PROCEDURE CursorShow (show: BOOLEAN);
(*
  If show is TRUE, the cursor will be displayed on the screen.
*)

PROCEDURE CursorWrap (wrap: BOOLEAN);
(*
  If wrap is TRUE, the cursor position is wrapped at the window boundaries.
  If wrap is FALSE, the cursor position is clipped at the window boundaries.
*)

PROCEDURE EraseCursor;
(*
  Erases the cursor if displayed.
*)

PROCEDURE DisplayCursor;
(*
  Displays the cursor on the screen with current shape and color.
  The hotX, hotY of the cursor indicate the bit in the cursor shape
  which has to be positioned at the current cursor location.
*)

PROCEDURE MoveCursor (x, y: INTEGER);
(*
  Moves the cursor to (x, y). (x, y) are coordinates relative to the window.
  If wrap is TRUE, the point (x, y) is wrapped. The cursor is then displayed.
  If wrap is FALSE, the point (x, y) is clipped.
  The cursor is then displayed only if the point (x, y) is inside the window.
*)

PROCEDURE GetCursorPosition (VAR x, y: INTEGER);
(*
  Returns the cursor coordinates relative to the window.
*)

PROCEDURE CursorVisible (): BOOLEAN;
(*
  Returns TRUE if the cursor is visible on the screen.
*)

END Graphics.
DEFINITION MODULE InOut;

(*
   Standard high-level formatted input/output,
   allowing for redirection to/from files
   
   From the book 'Programming in Modula-2' by Prof. N. Wirth.
*)

FROM SYSTEM IMPORT WORD;
FROM FileSystem IMPORT File;

EXPORT QUALIFIED
   EOL, Done, in, out, termCH,
   OpenInput, OpenOutput, CloseInput, CloseOutput,
   Read, ReadString, ReadInt, ReadCard, ReadWrd,
   Write, WriteLn, WriteString, WriteInt, WriteCard,
   WriteOct, WriteHex, WriteWrd;

CONST
   EOL = 36C;
   (*- end-of-line character *)

VAR
   Done: BOOLEAN;
   (*
      - set by several procedures;
      TRUE if the operation was successful, FALSE otherwise.
   *)
   termCH: CHAR;
   (*
      - terminating character from ReadString, ReadInt, ReadCard.
   *)
   in, out: File;
   (*
      - The currently open input and output files.
      Use for exceptional cases only.
   *)

PROCEDURE OpenInput(defext: ARRAY OF CHAR);
(*
   - Accept a file name from the terminal and open it for input (file variable 'in').

   in: defext default filetype or 'extension'.

   If the file name that is read doesn't end with '.', and it doesn't have an
   extension, then 'defext' is appended to the file name.

   If OpenInput succeeds, Done = TRUE and
   subsequent input is taken from the file until CloseInput is called.
*)
PROCEDURE OpenOutput(defext: ARRAY OF CHAR);
(*
  - Accept a file name from the terminal and open it for output
    (file variable 'out').
  in:   defext default filetype or 'extension'.

  If the file name that is read doesn't end with '.', and
  it doesn't have an extension, then 'defext' is appended to the file name.

  If OpenOutput succeeds, Done = TRUE and subsequent output
  is written to the file until CloseOutput is called.
*)
PROCEDURE CloseInput;
(*
  - Close current input file and revert to terminal for input.
*)
PROCEDURE CloseOutput;
(*
  - Close current output file and revert to terminal for output.
*)
PROCEDURE Read(VAR ch: CHAR);
(*
  - Read the next character from the current input.
  out:   ch   the character read; EOL for end-of-line

  Done = TRUE unless the input is at end of file.
*)
PROCEDURE ReadString(VAR s: ARRAY OF CHAR);
(*
  - Read a string from the current input.
  out:   s   the string that was read, excluding the terminator character.

  Leading blanks are accepted and thrown away, then characters are read into 's'
  until a blank or control character is entered. ReadString truncates the input
  string if it is too long for 's'. The terminating character is left in
  'termCH'. If input is from the terminal, BS and DEL are allowed for editing.
*)
PROCEDURE ReadInt(VAR x: INTEGER);
(*
  - Read an INTEGER representation from the current input.
  out:   x   the value read.

  ReadInt is like ReadString, but the string is converted to
  an INTEGER value if possible, using the syntax:
    ['+'|'-'] digit { digit }.

  Done = TRUE if some conversion took place.
*)
PROCEDURE ReadCard(VAR x: CARDINAL);
(*
  - Read an unsigned decimal number from the current input.
    out:  x  the value read.
    ReadCard is like ReadInt, but the syntax is:  
      digit { digit }.
  *)

PROCEDURE ReadWrd(VAR w: WORD);
(*
  - Read a WORD value from the current input.
    out:  w  the value read.
    Done is TRUE if a WORD was read successfully. This procedure cannot be
    used when reading from the terminal.
    Note that the meaning of WORD is system dependent.
  *)

PROCEDURE Write(ch: CHAR);
(*
  - Write a character to the current output.
    in:  ch  character to write.
  *)

PROCEDURE WriteLn;
(*
  - Write an end-of-line sequence to the current output.
  *)

PROCEDURE WriteString(s: ARRAY OF CHAR);
(*
  - Write a string to the current output.
    in:  s  string to write.
  *)

PROCEDURE WriteInt(x: INTEGER; n: CARDINAL);
(*
  - Write an integer in right-justified decimal format.
    in:  x  value to be output,
         n  minimum field width.
    The decimal representation of 'x' (including '-' if x is negative) is output,
    using at least n characters (but more if needed).
    Leading blanks are output if necessary.
  *)
PROCEDURE WriteCard(x, n: CARDINAL);
(*
- Output a CARDINAL in decimal format.

in:  x  value to be output,
     n  minimum field width.

The decimal representation of the value 'x' is output,
using at least n characters (but more if needed).
Leading blanks are output if necessary.
*)

PROCEDURE WriteOct(x, n: CARDINAL);
(*
- Output a CARDINAL in octal format.
[see WriteCard above]
*)

PROCEDURE WriteHex(x, n: CARDINAL);
(*
- Output a CARDINAL in hexadecimal format.

in:  x  value to be output,
     n  minimum field width.

Four uppercase hex digits are written, with leading blanks if n > 4.
*)

PROCEDURE WriteWrd(w: WORD);
(*
- Output a WORD

in:  w  WORD value to be output.

Note that the meaning of WORD is system dependent,
and that a WORD cannot be written to the terminal.
*)

END InOut.
DEFINITION MODULE Keyboard;
(*
  Default driver for terminal input.
  [Private module of the MODULA-2/86 system]

  Derived from the Lilith Modula-2 system developed by the group of Prof. N. Wirth
  at ETH Zürich, Switzerland.
*)

EXPORT QUALIFIED Read, KeyPressed;

PROCEDURE Read (VAR ch: CHAR);
(*
  - Read a character from the keyboard.

  out: ch character read

  If necessary, Read waits for a character to be entered.
  Characters that have been entered are returned immediately,
  with no echoing, editing or buffering.

  - Ctrl-C terminates the current program
  - ASCII.cr is transformed into ASCII.EOL
*)

PROCEDURE KeyPressed (): BOOLEAN;
(*
  - Test if a character is available from the keyboard.

  out: returns TRUE if a character is available for reading
*)

END Keyboard.
LoadPath

DEFINITION MODULE LoadPath;

PROCEDURE GetLoad(VAR str: ARRAY OF CHAR);
(* Get the complete filename of the file loaded by MSDOS
   In the environment:
   - Look for the sequence 0,0
   - Skip two bytes (meaning unknown, often 1,0)
   - Take next characters until a 0
   Return empty string if:
   - Doesn't find 0,0
   - filename > HIGH(str)
   *)

PROCEDURE GetLoadDir(VAR str: ARRAY OF CHAR);
(* Return the directory of the loaded file or empty string
   if problems
   *)

END LoadPath.
DEFINITION MODULE LogiFile;
(*
File sub-system for Logitech utilities;
*)

FROM SYSTEM IMPORT ADDRESS, WORD, BYTE;
FROM TimeDate IMPORT Time;

EXPORT QUALIFIED
File, OpenMode,
NUL, LF, CR, EOL, EOF,
Open, Create, Close, Delete,
GetFileDate,
GetPos, SetPos, Reset,
ReadChar, WriteChar,
ReadNBytes, WriteNBytes,
ReadByte, WriteByte, ModifyByte,
ReadWord, WriteWord, ModifyWord,
EndFile;

TYPE
File;
OpenMode = (ReadOnly, WriteOnly, ReadWrite);

CONST
NUL = 0C;
LF = 12C;
CR = 15C;
EOL = 36C; (* end of line character for character files *)
EOF = 32C; (* end of file character for character files *)

(* ----------------------------------------------- *)
(* Operations on the directory: *)
(* ----------------------------------------------- *)

PROCEDURE Open(VAR f : File; (* Open an existing file *)
    name : ARRAY OF CHAR;
    mode : OpenMode;
    VAR done : BOOLEAN);

PROCEDURE Create(VAR f : File; (* Open a new file *)
    name : ARRAY OF CHAR;
    VAR done : BOOLEAN);

PROCEDURE Close(VAR f : File; (* Close a file *)
    VAR done : BOOLEAN);

PROCEDURE Delete(VAR f : File; (* Remove a file *)
    VAR done : BOOLEAN); (* remove from directory *)
PROCEDURE GetFileDate(  
      f : File;
      VAR datetime : Time);
      (* returns the creation date and time of the given file *)

(* --------------------------------------------------------------- *)
(* Positioning inside an open file,
      highpos and lowpos represent a double precision value:
      highpos * 10000H + lowpos. *)

PROCEDURE GetPos(  
      f : File;
      VAR highpos CARDINAL;
      VAR lowpos CARDINAL);
      (* Get current byte position of the file *)

PROCEDURE SetPos(  
      f : File;
      highpos CARDINAL;
      lowpos CARDINAL);
      (* Set file to indicated byte position, and set to "ReadMode" *)

PROCEDURE Reset(f: File);
      (* Position the file at the beginning and set to "ReadMode" *)

(* --------------------------------------------------------------- *)
(* Reading and Writing of files in "TextMode":
   Calls to ReadChar and WriteChar set the internal file status to "TextMode".
   This means that EOL and EOF characters are interpreted,
   EOL:
      WriteChar(f,EOL) writes the physical EOL-character onto the file
      (under MS-DOS: CR,LF; under XENIX: LF).
   ReadChar(f,ch) with ch=EOL under MS-DOS means that the read procedure
      found a CR on the file, translated it into EOL, and skipped
      the character just after CR, assuming that it is a LF.
   ReadChar(f,ch) with ch=EOL under XENIX means that the read procedure
      found a LF on the file, and translated it into EOL.
   EOF (<ctrl-Z>, 32C)
      When writing, the EOF is treated like any other character,
      i.e., WriteChar(f,EOF) just writes EOF onto the file, thus setting a logical EOF.
      If the character EOF is found on a file,
         this results in ReadChar(f,ch) with ch=NUL, AND EndFile(f)=TRUE.
      Note: The EOF is never written automatically by LogiFile!
      *)

PROCEDURE ReadChar(  
      f : File;  (* Read a character from file *)
      VAR ch : CHAR);

PROCEDURE WriteChar(  
      f : File;  (* Write a character to file *)
      ch : CHAR);

(* --------------------------------------------------------------- *)
(* Reading and Writing of files in "BinaryMode":
   Calls to all the following read, write, and modify procedures
      set the internal file status to "BinaryMode".
      *)
PROCEDURE ReadNBytes( f : File;
        buffPtr : ADDRESS;
        requestedBytes : CARDINAL;
        VAR read : CARDINAL);
        (* Read requested bytes into buffer at address *)
        (* 'buffPtr', number of effective read bytes is *)
        (* returned in 'read' *)
PROCEDURE WriteNBytes( f : File;
        buffPtr : ADDRESS;
        requestedBytes : CARDINAL;
        VAR written : CARDINAL);
        (* Write requested bytes into buffer at address *)
        (* 'buffPtr', number of effective written bytes is *)
        (* returned in 'written' *)
PROCEDURE ReadByte( f : File;
        VAR b : BYTE);
PROCEDURE WriteByte( f : File;
        b : BYTE);
PROCEDURE ModifyByte( f : File;
        b : BYTE);
PROCEDURE ReadWord( f : File;
        VAR w : WORD);
PROCEDURE WriteWord( f : File;
        w : WORD);
PROCEDURE ModifyWord( f : File;
        w : WORD);

(* --------------------------------------------------------- *)
PROCEDURE EndFile( f: File): BOOLEAN;

(* End of file reached, with the following logic:

After Open : undefined
After Create : undefined

"WriteMode" : undefined
    i.e., after calls to Write- or Modify- procedures.

"ReadMode":
    When reading in "TextMode" (using ReadChar) a logical EOF (32C,<ctrl-Z>)
    in the file sets EndFile to TRUE.
    If there is no logical EOF in the file, or when reading in "BinaryMode",
    reading to or after the physical EOF sets EndFile to TRUE.

NOTE: All read procedures return NUL or read bytes=0
    when reading past the end of the file.
    In "BinaryMode" this is always the physical EOF,
    in "TextMode" this is either the logical EOF, or
    if there is no logical EOF, the physical EOF.
*)

END LogiFile.
LongIO

DEFINITION MODULE LongIO;

EXPORT QUALIFIED ReadLongInt, WriteLongInt;

PROCEDURE ReadLongInt (VAR longX : LONGINT);
PROCEDURE WriteLongInt(x: LONGINT; n: CARDINAL);

END LongIO.
lookup

DEFINITION MODULE Lookup;

FROM LogiFile IMPORT File;

EXPORT QUALIFIED LookupFile;

PROCEDURE LookupFile(prompt
name
paths
defext
VAR file
query, autoquery, acceptoptions
VAR effectivenname
VAR goodfile

(* for implementation the modules FileNames, *)
(* Options and CompFile are imported *)

(* prompt : string is displayed on terminal *)
(* name : for construction of a default file name *)
(* paths : drive and paths; separated by ';' *)
(* defext : default extension of searched file *)
(* file : opened file *)
(* query : explicit asking for file name *)
(* autoquery : switch automatically to mode query if not found *)
(* acceptoptions : accept options appended to file name *)
(* options are not evaluated *)
(* effectivenname : name of found file *)
(* goodfile : lookup was successful *)

END Lookup.
MathLibO

DEFINITION MODULE MathLibO;
(*
    Real Math Functions
    From the book 'Programming in Modula-2' by Prof. N. Wirth.
 *)

EXPORT QUALIFIED
    sqrt, exp, ln, sin, cos, arctan, real, entier;

PROCEDURE sqrt(x: REAL): REAL;
(*
    - returns square root x
    x must be positive.
 *)

PROCEDURE exp(x: REAL): REAL;
(*
    - returns e^x where e = 2.71828...
 *)

PROCEDURE ln(x: REAL): REAL;
(*
    - returns natural logarithm with base e = 2.71828.. of x
    x must be positive and not zero
 *)

PROCEDURE sin(x: REAL): REAL;
(*
    - returns sin(x) where x is given in radians
 *)

PROCEDURE cos(x: REAL): REAL;
(*
    - returns cos(x) where x is given in radians
 *)

PROCEDURE arctan(x: REAL): REAL;
(*
    - returns arctan(x) in radians
 *)

PROCEDURE real(x: INTEGER): REAL;
(*
    - type conversion from INTEGER to REAL
 *)
PROCEDURE entier(x: REAL): INTEGER;
(* - returns the largest integer number less or equal x
   Examples: entier(1.5) = 1; entier(-1.5) = -2;
   If x cannot be represented in an INTEGER, the result is undefined.
*)

END MathLib0.
Mouse

DEFINITION MODULE Mouse;
(*

Mouse Driver Interface

Short description:

The functions implemented in this module provide a Modula-2 interface for the
LOGITECH Mouse Driver. This driver interface is compatible with the
Microsoft Mouse Driver interface, so this module can be used with all the
compatible mouse drivers. For detailed description of these functions, please
refer to your mouse documentation:

e.g. LOGITECH Mouse Driver Programmer’s Reference Manual
Microsoft Mouse, Installation and Operation Manual

Microsoft is a registered trademark of Microsoft Corporation
*)

EXPORT QUALIFIED
   DriverInstalled,
   Button, ButtonSet,
   FlagReset,
   ShowCursor, HideCursor,
   GetPosBut,
   SetCursorPos,
   GetButPres, GetButRel,
   SetHorizontalLimits, SetVerticalLimits,
   GraphicCursor, SetGraphicCursor,
   SetTextCursor,
   ReadMotionCounters,
   Event, EventSet, EventHandler, SetEventHandler,
   LightPenOn, LightPenOff,
   SetMickeyPerPixel,
   ConditionalOff,
   SetSpeedThreshold;
VAR
DriverInstalled: BOOLEAN;
(* Flag that indicates, whether a mouse driver is loaded or not. If its value is FALSE, none of the following functions will work properly.*)

TYPE
Button = (LeftButton,
    RightButton, (* not available on some mice *)
    MiddleButton (* not available on some mice *)
);

ButtonSet = SET OF Button;

PROCEDURE FlagReset( VAR mouseStatus : INTEGER;
    VAR numberOfButtons : CARDINAL);
(* Microsoft Mouse Driver System Call 0
Input : AX = 0 System Call 0
Output: AX --> mouse status
0 (FALSE): mouse hardware and software not installed
-1 (TRUE): mouse hardware and software installed
BX --> number of mouse buttons
*)

PROCEDURE ShowCursor;
(* Microsoft Mouse Driver System Call 1
Input : AX = 1 System Call 1
*)

PROCEDURE HideCursor;
(* Microsoft Mouse Driver System Call 2
Input : AX = 2 System Call 2
*)

PROCEDURE GetPosBut(VAR buttonStatus : ButtonSet;
    VAR horizontal, vertical : INTEGER);
(* Microsoft Mouse Driver System Call 3
Input : AX = 3 System Call 3
Output: BX --> mouse button status
CX --> horizontal cursor position
DX --> vertical cursor position
*)

PROCEDURE SetCursorPos(horizontal, vertical : INTEGER);
(* Microsoft Mouse Driver System Call 4
Input : AX = 4 System Call 4
CX <-- horizontal mouse cursor position
DX <-- vertical mouse cursor position
*)
PROCEDURE GetButPres(button VAR buttonStatus : ButtonSet;
                     buttonPressCount VAR CARDINAL;
                     horizontal, vertical : INTEGER);
(* Microsoft Mouse Driver System Call 5
 Input : AX = 5 System Call 5
 BX <-- button
 Output: AX --> current button status
 BX --> count of button presses since last call to this function
 CX --> horizontal cursor position at last press
 DX --> vertical cursor position at last press *)

PROCEDURE GetButRel(button VAR buttonStatus : ButtonSet;
                     buttonReleaseCount VAR CARDINAL;
                     horizontal, vertical : INTEGER);
(* Microsoft Mouse Driver System Call 6
 Input : AX = 6 System Call 6
 BX <-- button
 Output: AX --> current button status
 BX --> count of button releases since last call to this function
 CX --> horizontal cursor position at last press
 DX --> vertical cursor position at last press *)

PROCEDURE SetHorizontalLimits(minPos, maxPos: INTEGER);
(* Microsoft Mouse Driver System Call 7
 Input : AX = 7 System Call 7
 CX <-- minimum horizontal position
 DX <-- maximum horizontal position *)

PROCEDURE SetVerticalLimits(minPos, maxPos: INTEGER);
(* Microsoft Mouse Driver System Call 8
 Input : AX = 8 System Call 8
 CX <-- minimum vertical position
 DX <-- maximum vertical position *)

TYPE GraphicCursor = RECORD
  screenMask, cursorMask: ARRAY [0..15] OF BITSET;
  hotX, hotY: [-16..16];
END;

(* The screenMask is first ANDed into the display; then the cursorMask is XORed into the display. The hot spot coordinates are relative to the upper-left corner of the cursor image, and define where the cursor actually 'points to'. *)
PROCEDURE SetGraphicCursor(VAR cursor: GraphicCursor);
(* Microsoft Mouse Driver System Call 9
Input : AX = 9 System Call 9
BX <-- cursor hot spot (horizontal)
CX <-- cursor hot spot (vertical)
ES:DX <-- pointer to screen and cursor masks *)

PROCEDURE SetTextCursor(selectedCursor,
screenMaskORscanStart,
cursorMaskORscanStop CARDINAL);
(* Microsoft Mouse Driver System Call 10
Input : AX = 10 System Call 10
BX <-- cursor select
0: Software text cursor
1: Hardware text cursor
CX <-- screen mask value or scan line start
DX <-- cursor mask value or scan line stop

For the software text cursor, the second two parameters specify the screen and cursor masks.
The screen mask is first ANDed into the display; then the cursor mask is XORed into the display.
For the hardware text cursor, the second two parameters contain the line numbers of the first and last scan line in the cursor to be shown on the screen *)

PROCEDURE ReadMotionCounters(VAR horizontal,
vertical :INTEGER);
(* Microsoft Mouse Driver System Call 11
Input : AX = 11 System Call 11
CX <-- horizontal count
DX <-- vertical count *)

TYPE
Event = {Motion,
LeftDown,
LeftUp,
RightDown, (* not available on some mice *)
RightUp, (* not available on some mice *)
MiddleDown, (* not available on some mice *)
MiddleUp (* not available on some mice *)
);

EventSet = SET OF Event;

EventHandler = 
PROCEDURE (EventSet, (* condition mask *)
ButtonSet, (* button state *)
INTEGER, (* horizontal cursor pos *)
INTEGER (* vertical cursor pos *)
);
PROCEDURE SetEventHandler(mask : EventSet;
                           handler : EventHandler);
(* Microsoft Mouse Driver System Call 12
 Input : AX = 12 System Call 12
        CX <-- call mask
        ES:DX <-- address of handler routine

 Establish conditions and handler for mouse events.
 After this, when an event occurs that is in the mask,
 the handler is called with the event set that actually happened,
 the current button status, and the cursor x and y.
 *)

PROCEDURE LightPenOn;
(* Microsoft Mouse Driver System Call 13
 Input : AX = 13 System Call 13
 *)

PROCEDURE LightPenOff;
(* Microsoft Mouse Driver System Call 14
 Input : AX = 14 System Call 14
 *)

PROCEDURE SetMickeysPerPixel(horPix, verPix: CARDINAL);
(* Microsoft Mouse Driver System Call 15
 Input : AX = 15 System Call 15
        CX <-- horizontal mickey/pixel ratio
        DX <-- vertical mickey/pixel ratio
 *)

PROCEDURE ConditionalOff(left, top, right, bottom: INTEGER);
(* Microsoft Mouse Driver System Call 16
 Input : AX = 16 System Call 16
        CX <-- left
        DX <-- top
        SI <-- right
        DI <-- bottom
 *)

PROCEDURE SetSpeedThreshold(threshold: CARDINAL);
(* Microsoft Mouse Driver System Call 19
 Input : AX = 19 System Call 19
        DX <-- threshold in mickeys/second
 *)

END Mouse.
DEFINITION MODULE NumberConversion;

(* Conversion between numbers and strings

Conventions for the routines that convert a string to a number:

- Leading blanks are skipped.
- A plus sign ('+') preceding the number is always accepted,
  a minus sign ('-') is only accepted when converting to INTEGER or LONGINT.
- Blanks between the plus or minus sign and the number are skipped.
- The last character in the string must belong to the number to be converted.
  No trailing blanks or other trailing characters are allowed.
- 'done' returns TRUE if the conversion is successful.

Conventions for the routines that convert a number to
a string:

- If the string is too small, the number is truncated.
- If less than 'width' digits are needed to represent the number,
  leading blanks are added.
*)

EXPORT QUALIFIED
MaxBase, BASE,
StringToCard, StringToInt, StringToLongInt, StringToNum,
CardToString, IntToString, LongIntToString, NumToString;

CONST MaxBase = 16;

TYPE BASE = [2..MaxBase];

PROCEDURE StringToCard(str : ARRAY OF CHAR;
                      VAR num : CARDINAL;
                      VAR done : BOOLEAN);

(*
- Convert a string to a CARDINAL number.

in: str  string to convert
out: num converted number
done TRUE if successful conversion,
     FALSE if number out of range,
     or contents of string non numeric.
*)
PROCEDURE StringToInt(str : ARRAY OF CHAR;
   VAR num : INTEGER;
   VAR done : BOOLEAN);
(* - Convert a string to an INTEGER number. *)
   in: str string to convert
   out: num converted number
do: done TRUE if successful conversion,
    FALSE if number out of range,
or contents of string non numeric.
*)

PROCEDURE StringToLongInt(str : ARRAY OF CHAR;
   VAR num : LONGINT;
   VAR done : BOOLEAN);
(* - Convert a string to a LONGINT number. *)
in: str string to convert
out: num converted number
do: done TRUE if successful conversion,
    FALSE if number out of range,
or contents of string non numeric.
*)

PROCEDURE StringToNum(str : ARRAY OF CHAR;
   base : BASE;
   VAR num : CARDINAL;
   VAR done : BOOLEAN);
(* - Convert a string to a CARDINAL number. *)
in: str string to convert
base: base the base of the number represented in the string
out: num converted number
do: done TRUE if successful conversion,
    FALSE if number out of range,
or contents of string not within base.
*)

PROCEDURE CardToString(num : CARDINAL;
   VAR str : ARRAY OF CHAR;
   width : CARDINAL);
(* - Convert a CARDINAL number to a string. *)
in: num number to convert
width: width width of the returned string
out: str returned string representation of the number
*)
PROCEDURE IntToString(num : INTEGER;
    VAR str : ARRAY OF CHAR;
    width : CARDINAL);
(*
   - Convert an INTEGER number to a string.
   in:   num    number to convert
         width  width of the returned string
   out:  str    returned string representation of the number
*)

PROCEDURE LongIntToString(num : LONGINT;
    VAR str : ARRAY OF CHAR;
    width : CARDINAL);
(*
   - Convert a LONGINT number to a string.
   in:   num    number to convert
         width  width of the returned string
   out:  str    returned string representation of the number
*)

PROCEDURE NumToString(num : CARDINAL;
    base : BASE;
    VAR str : ARRAY OF CHAR;
    width : CARDINAL);
(*
   - Convert a number to the string representation in the specified base.
   in:   num    number to convert
          base   the base of conversion
          width  width of the returned string
   out:  str    returned string representation of the number
*)

END NumberConversion.
Chapter 9

Options

DEFINITION MODULE Options;
/*
 * Read a file specification, with options, from the terminal

 Derived from the Lilith Modula-2 system developed by the group of Prof. N. Wirth
 at ETH Zurich, Switzerland.
 */

EXPORT QUALIFIED
NameParts, NamePartSet, Termination,
FileNameAndOptions, GetOption;

TYPE
Termination = (norm, empty, can, esc);
NameParts = (NameDrive, NamePath, NameName, NameExt);
NamePartSet = SET OF NameParts;

PROCEDURE FileNameAndOptions(default ARRAY OF CHAR;
ARRAY OF CHAR;
Termination;
BOOLEAN;
NamePartSet);
/*
 - Read file name and options from terminal.

 in: default the file specification to use if one is not entered,
 acceptOption if TRUE, allow options to be entered,

 out: name the file specification,
 term how the read ended,
 readInName which parts of specification are present.

 If the current drive is specified in the default name,
 and if no drive is entered, then the actual name of the current drive
 is returned with the name read.

 The variable 'term' indicates the status of the input termination:
 norm : normally terminated
 empty : normally terminated, but name is empty
 can : <can> is typed, input line cancelled
 esc : <esc> is typed, no file specified

 Input is terminated by a <cr>, blank, <can>, or <esc>.
 <bs> and <del> are allowed while entering the file name.
 */
PROCEDURE GetOption(VAR optStr: ARRAY OF CHAR;
       VAR length: CARDINAL);
(*
   - Get another option from the last call to FileNameAndOptions.

   out:   optStr        text of the option,
          length       length of optStr.

   Calls to GetOption return the options from the last call to FileNameAndOptions,
in the order they were entered. When there are no more options,
a length of 0 is returned.
*)
END Options.
DEFINITION MODULE Overlay;

FROM SYSTEM IMPORT ADDRESS;
FROM RTSMain IMPORT Status, OverlayPtr, OverlayDescriptor;

EXPORT QUALIFIED

ErrorCode,
OverlayId, InstallOverlay, DeInstallOverlay, CallOverlay, GetErrorCode,
LayerId, NewLayer, DisposeLayer, InstallOverlayInLayer,
CallOverlayInLayer;

TYPE

ErrorCode =
(Done, NotDone, FileNotFound, BadFormat, InsufMemory, VersionConflict);

PROCEDURE GetErrorCode (error : ErrorCode; VAR str : ARRAY OF CHAR);

PROCEDURE CallOverlay (*
fileName VAR done VAR status ARRAY OF CHAR;
ErrorCode Status)

(* Loads and executes the overlay defined by filename. Upon termination,
the overlay is unloaded from memory. The filename must be a complete DOS name.
IF there is no path given in the filename, the loader will search
in the directory from which came the base of the application,
else it search only the given directory. *)

TYPE

OverlayId = OverlayPtr;

(* defines a handle to a resident overlay *)

PROCEDURE InstallOverlay (*
fileName : ARRAY OF CHAR;
VAR done : ErrorCode ;
VAR status : Status ) : OverlayId;

(* Loads and executes a resident overlay. Upon termination, the resident overlay
is logically linked to the overlay which has loaded it.
The handle returned can be used to explicitly unload the resident overlay. *)
PROCEDURE DeInstallOverlay (overlayId : OverlayId);
 (* explicitly unloads the resident overlay defined by its handle *)

(* The following procedures perform the same task as the previous one.
The only difference is the use of a parameter 'layer'. A layer is
a piece of memory, reserved from DOS through the call to NewLayer, that
one or many overlays can share. Giving the handle returned by Newlayer as
parameter to these routines forces the loader to use the space left in this layer.

Note: for each overlay, the loader knows in which layer it is loaded.
So there is no need to tell the loader in which layer a resident overlay
is loaded. That is why there is only one procedure DeInstallOverlay. *)

TYPE
LayerId;

PROCEDURE NewLayer (VAR layer : LayerId; size : CARDINAL; VAR done : BOOLEAN);
(* ask DOS for memory. size is given in paragraphs *)

PROCEDURE DisposeLayer (layer : LayerId);
(* give back the reserved memory to DOS *)

PROCEDURE CallOverlayInLayer
    (fileName : ARRAY OF CHAR;
     layer : LayerId;
     VAR done : ErrorCode;
     VAR status : Status);

PROCEDURE InstallOverlayInLayer
    (fileName : ARRAY OF CHAR;
     layer : LayerId;
     VAR done : ErrorCode;
     VAR status : Status): OverlayId;

END Overlay.
DEFINITION MODULE Processes;
(* (pseudo-) concurrent programming with SEND/WAIT

From the book 'Programming in Modula-2' by Prof. N. Wirth.
*)

EXPORT QUALIFIED
SIGNAL, SEND, WAIT,
StartProcess, Awaited, Init;

TYPE
SIGNAL;
(*
- SIGNAL's are the means of synchronization between processes.
Any variable of type SIGNAL must be initialized explicitly
by means of procedure 'Init' before using it
with any other procedure of this module.
*)

PROCEDURE StartProcess (P: PROC; n: CARDINAL);
(*
- Start up a new process.

in:  P  top-level procedure that will execute in this process.
     n  number of bytes of workspace to be allocated to it.

Allocates (from Storage) a workspace of n bytes, and creates
a process executing procedure P in that workspace.
Control is given to the new process.

Caution : The caller must ensure that the workspace size is sufficient for P.

Errors : StartProcess may fail due to insufficient memory.
*)

PROCEDURE SEND (VAR s: SIGNAL);
(*
- Send a signal

in:  s  the signal to be sent.

out:  s  the signal with one less process waiting for it.

If no process is waiting for s, SEND has precisely no effect.
Otherwise, some process which is waiting for s is given control
and allowed to continue from WAIT.
*)
PROCEDURE WAIT (VAR s: SIGNAL);
(*
   - Wait for some other process to send a signal.
       in:  s the signal to wait for.

   The current process waits for the signal s. At some later time,
   a SEND(s) by some other process can cause this process to return from WAIT.

   Errors: If all other processes are waiting, WAIT terminates the program.
 *)
PROCEDURE Awaited (s:SIGNAL): BOOLEAN;
(*
   - Test whether any process is waiting for a signal.
       in:  s the signal of interest.
       out: TRUE if and only if at least one process is waiting for s.
 *)
PROCEDURE Init (VAR s: SIGNAL);
(*
   - Initialize a SIGNAL object.
       in:  s the signal to be initialized
       out: s the initialized signal (ready to be used
            with one of the procedures declared above)

   An object of type SIGNAL must be initialized with this procedure
   before it can be used with any of the other operations.
   After initialization of s, Awaited(s) is FALSE.
 *)
END Processes.
DEFINITION MODULE Random;

(* Random numbers generator. 
   Algorithm: Based on the additive congruential method 
   (Knuth, The art of computer programming, Vol.2, pp 26-27) *)

PROCEDURE Randomize;
(* Initializes the random number generator. 
The random number sequence following a call to Randomize cannot be reproduced. 
A call to Randomize is done automatically at the initialization 
of this module. *)

PROCEDURE RandomInit (seed: CARDINAL);
(* Initializes the random number generator. 
The 'seed' parameter is used to generate the first number of the 
sequence. Thus, following a call to RandomInit with a given seed, 
the random number sequence will always be the same, regardless of 
any previous call to Randomize, RandomCard, etc... 
Note: RandomCard, RandomInt, RandomReal are based on the same 
generator, so in order to get the same sequence, these functions 
must be called in the same order. *)

PROCEDURE RandomCard (bound : CARDINAL) : CARDINAL;
(* Returns a random cardinal in the range (0 <= r < bound) 
   if bound is greater than 0, or in the range (0 <= r <= MaxCard) if bound = 0. *)

PROCEDURE RandomInt (bound : INTEGER): INTEGER;
(* Returns a random integer in the range (0 <= r < bound) 
   if bound is greater than 0, or in the range (0 <= r <= MaxInt) if bound = 0. *)

PROCEDURE RandomReal (): REAL;
(* Returns a random real uniformly distributed the range (0.0 <= r < 1.0) with 
   15-16 decimal digits (IEEE double precision floating point numbers standard) *)

END Random.
DEFINITION MODULE RealConversions;
(* Conversion Module for floating numbers *)

EXPORT QUALIFIED
  RealToString, StringToReal;

PROCEDURE RealToString (r : REAL;
  digits, width : INTEGER;
  VAR str : ARRAY OF CHAR;
  VAR okay : BOOLEAN);

(* - Convert a REAL to right-justified fixed point or exponent representation

in:  r real number to be represented,
  digits number of digits to the right of the decimal point,
  width maximum width of representation,

out: str string result,
     okay TRUE if the conversion is done properly, FALSE otherwise.

If 'digits' < 0 then exponent notation is used,
otherwise fixed point notation is used.
Note that a leading '-' is generated if r < 0, but never a '+'.

If the representation of 'r' uses fewer than 'width' digits,
blanks are added on the left. If the representation will not fit in 'width'
then 'str' is returned empty and 'okay' is set to FALSE.

The minimum required 'width' is:
- if 'digits' < 0: width >= ABS(digits) + 8
- if 'digits' >= 0: width >= ABS(digits) + 2 + before,
  where 'before' is the number of digits before the decimal point of 'r' in fixed point notation
  (e.g. r = 123.456 --> before = 3, r = 0.012 --> before = 1)
*)
PROCEDURE StringToReal (str : ARRAY OF CHAR;
    VAR r : REAL;
    VAR okay : BOOLEAN);

(*
- Convert ARRAY OF CHAR to REAL representation.
in: str string to be represented,
out: r REAL result,
    okay TRUE if the conversion is done properly,
    FALSE otherwise.

Leading blanks are skipped, control code characters and space are considered as legal terminators. The syntax for a legal real representation in 'str' is:

realnumber = fixedpointnumber [exponent].
fixedpointnumber = [sign] {digit} [. {digit}].
exponent = {'e' | 'E'} [sign] digit {digit}.
sign = '+' | '−'.
digit = '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'.

The following numbers are legal representations of one hundred:
100, 10E1, 100E0, 1000E-1, E2, +E2, 1E2, +1E2, +1E+2, 1E+2.

At most 15 digits are significant, leading zeros not counting.
The range of representable real numbers is: 1.0E-307 <= ABS(r) < 1.0E308
*)

END RealConversions.
DEFINITION MODULE RealInOut;
(*
* Terminal input/output of REAL values
*
The implementation of this module uses the procedures ‘ReadString’
and ‘WriteString’ of module ‘InOut’ for reading and writing of REAL values.
Therefore, redirection of i/o through ‘InOut’ applies, too.

From the book ‘Programming in Modula-2’ by Prof. N. Wirth.
*)

EXPORT QUALIFIED
  ReadReal, WriteReal, WriteRealOct, Done;

VAR Done: BOOLEAN;

PROCEDURE ReadReal (VAR x: REAL);
(*
- Read a REAL from the terminal.
out: x the number read.

The range of representable valid real numbers is:
1.0E-307 <= ABS(r) < 1.0E308

The syntax accepted for input is:
realnumber = fixedpointnumber [exponent].
fixedpointnumber = [sign] {digit} [ '.' {digit} ].
exponent = ('e' | 'E') [sign] digit {digit}.
sign = '+' | '-'.
digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'.

The following numbers are legal representations of one hundred:
100, 10E1, 1000E-1, E2, +E2, 1E2, +1E2, +1E+2, 1E+2.

At most 15 digits are significant, leading zeros not counting.
Input terminates a control character or space.
DEL or BS is used for backspacing

The variable ‘Done’ indicates whether a valid number was read.
*)

PROCEDURE WriteReal (x: REAL; n: CARDINAL);
(*
- Write a REAL to the terminal, right-justified.
in: x number to write,
n minimum field width.

If fewer than n characters are needed to represent x, leading blanks are output.
At least 10 characters are needed to write any REAL number.
*)
PROCEDURE WriteRealOct (x: REAL);
(*
    - Write a REAL to terminal, as four words in octal form

    in: x number to write,
*)
END RealInOut.
RS232Code

DEFINITION MODULE RS232Code;
(*
High-speed interrupt-driven input/output via the
RS-232 asynchronous serial port

This module provides interrupt-driven I/O via the serial port, but the
Interrupt Service Routine is implemented using in-line code
(as opposed to IOTRANSFER). Characters received are stored in
a buffer of 100H characters.

This approach is NOT portable to other Modula-2 implementations,
but it allows for treatment of interrupts with a high frequency.

Derived from the Lilith Modula-2 system developed by the group of Prof. N. Wirth
at ETH Zurich, Switzerland.
*)

EXPORT QUALIFIED
Init, StartReading, StopReading,
BusyRead, Read, Write;

PROCEDURE Init (baudRate : CARDINAL;
stopBits : CARDINAL;
parityBit : BOOLEAN;
evenParity : BOOLEAN;
nbrOfBits : CARDINAL;
VAR result : BOOLEAN);
(*
- Initialize the serial port.

in: baudRate transmission speed,
stopBits number of stop bits (usually 1 or 2),
parityBit if TRUE, parity is used, otherwise not,
evenParity if parity is used, this indicates even/odd,
nbrOfBits number of data bits (usually 7 or 8),

out: result TRUE if the initialization was completed.

The legal values for the parameters depend on the implementation
(e.g. the range of supported baud rates).
*)

PROCEDURE StartReading;
(*
- Allow characters to be received from the serial port.

This procedure initializes the communication controller to generate interrupts
upon reception of a character. It also unmask the corresponding interrupt level in
the interrupt controller.
*)
PROCEDURE StopReading;
(*
  - Disable receiving from the serial port.

A call to this procedure disables the communication controller
from generating interrupts. In addition it terminates the coroutine which
listens to the line. The old interrupt vector as well as the old state
of the interrupt controller (mask) is restored.
*)

PROCEDURE BusyRead (VAR ch : CHAR;
VAR received : BOOLEAN);
(*
  - Read a character from serial port, if one has been received.

out:  ch     the character received, if any,
      received TRUE if a character was received.

If no character has been received, then ch = OC, and received = FALSE.
*)

PROCEDURE Read (VAR ch: CHAR);
(*
  - Read a character from the serial port.

out:  ch     the character received.

As opposed to BusyRead, Read waits for a character to arrive.
*)

PROCEDURE Write (ch: CHAR);
(*
  - Write a character to the serial port.

in:   ch     character to send.

Note: no interpretation of characters is made.
*)

END RS232Code.
DEFINITION MODULE RS232Int;

="/*

 RS232Int

 Interrupt-driven input/output via the RS-232 asynchronous serial port

 Interrupts are treated with the standard procedure IOTRANSFER. Charcters received are stored in a buffer of 400H characters.

 This module initializes the serial port as follows:
 baudRate = 1200,
 stopBits = 1,
 parityBit = FALSE,
 evenParity = don’t care,
 nbrOfBits = 8

 Derived from the Lilith Modula-2 system developed by the group of Prof. N. Wirth at ETH Zurich, Switzerland.
 */"

EXPORT QUALIFIED
 Init, StartReading, StopReading,
 BusyRead, Read, Write;

PROCEDURE Init (baudRate : CARDINAL;
 stopBits : CARDINAL;
 parityBit : BOOLEAN;
 evenParity : BOOLEAN;
 nbrOfBits : CARDINAL;
 VAR result : BOOLEAN);
/*
 - Initialize the serial port.

 in: baudRate transmission speed,
 stopBits number of stop bits (usually 1 or 2),
 parityBit if TRUE, parity is used, otherwise not,
 evenParity if parity is used, this indicates even/odd,
 nbrOfBits number of data bits (usually 7 or 8),

 out: result TRUE if the initialization was completed.

 The legal values for the parameters depend on the implementation
 (e.g. the range of supported baud rates).
 */

PROCEDURE StartReading;
/*
 - Allow characters to be received from the serial port.

 This procedure initializes the communication controller to generate interrupts
 upon reception of a character. It also unmarks the corresponding
 interrupt level in the interrupt controller.
 */
PROCEDURE StopReading;
(*
  - Disable receiving from the serial port.

  A call to this procedure disables the communication controller
  from generating interrupts. In addition it terminates the coroutine
  which listens to the line. The old interrupt vector as well as the old state of
  the interrupt controller (mask) is restored.
*)

PROCEDURE BusyRead (VAR ch: CHAR; VAR received: BOOLEAN);
(*
  - Read a character from serial port, if one has been received.

  out: ch the character received, if any,
       received TRUE if a character was received.

  If no character has been received, then ch = OC, and received = FALSE.
*)

PROCEDURE Read (VAR ch: CHAR);
(*
  - Read a character from the serial port.

  out: ch the character received.

  As opposed to BusyRead, Read waits for a character to arrive.
*)

PROCEDURE Write (ch: CHAR);
(*
  - Write a character to the serial port.

  in: ch character to send.

  Note: no interpretation of characters is made.
*)

END RS232Int.
RS232Polling

DEFINITION MODULE RS232Polling;
(*
 Polled input/output via the RS-232 asynchronous serial port

Since this module does not use interrupts, it is the responsibility of the
programmer to poll (by calling 'Read' or 'BusyRead') frequently enough to ensure
that no characters are lost.

Derived from the Lilith Modula-2 system developed by the group of Prof. N. Wirth
at ETH Zurich, Switzerland.
*)

EXPORT QUALIFIED
 Init, BusyRead, Read, Write;

PROCEDURE Init (baudRate : CARDINAL;
 stopBits : CARDINAL;
 parityBit : BOOLEAN;
 evenParity : BOOLEAN;
 nbrOfBits : CARDINAL;
 VAR result : BOOLEAN);
(* - Initialize the serial port.
 in: baudRate transmission speed,
 stopBits number of stop bits (usually 1 or 2),
 parityBit if TRUE, parity is used, otherwise not,
 evenParity if parity is used, this indicates even/odd,
 nbrOfBits number of data bits (usually 7 or 8),
 out: result TRUE if the initialization was completed.

The legal values for the parameters depend on the implementation
(e.g. the range of supported baud rates).
*)

PROCEDURE BusyRead (VAR ch: CHAR; VAR received: BOOLEAN);
(* - Read a character from serial port, if one has been received.
 out: ch the character received, if any,
 received TRUE if a character was received.

If no character has been received, then ch = OC, and received = FALSE.
*)

PROCEDURE Read (VAR ch: CHAR);
(* - Read a character from the serial port.
 out: ch the character received.

As opposed to BusyRead, Read waits for a character to arrive.
*)
PROCEDURE Write (ch: CHAR);
(*
    - Write a character to the serial port.

    in:  ch    character to send.

    Note: no interpretation of characters is made.
*)

END RS232Polling.
RTSCoroutine

DEFINITION MODULE RTSCoroutine;
FROM SYSTEM IMPORT PROCESS;
EXPORT QUALIFIED addProcess;
VAR
  addProcess : PROCEDURE( PROCESS );
END RTSCoroutine.
DEFINITION MODULE RTSDevice;

FROM SYSTEM IMPORT ADDRESS, PROCESS;

EXPORT QUALIFIED
   GetDeviceStatus, SetDeviceStatus,
   GetPrioMask, SetPrioMask,
   SaveInterruptVector, RestoreInterruptVector,
   InstallHandler, UninstallHandler;

PROCEDURE GetDeviceStatus( deviceNr : CARDINAL;
   VAR enabled : BOOLEAN);

   /*
   - Return the status of a device in the device mask
   
in:  deviceNr number of the device to be checked bitnumber (0..7) of bit
        for device in interrupt controller 8259 mask
   
   out: enabled  TRUE if interrupts from the device are enabled,
        FALSE otherwise
   */

PROCEDURE SetDeviceStatus(deviceNr : CARDINAL;
   enable : BOOLEAN);

   /*
   - Set the status of a device in the device mask
   
in:  deviceNr number of the device to enable or disable bitnumber (0..7)
        of bit for device in interrupt controller 8259 mask
   
enable  if TRUE, enable interrupts from the device,
        otherwise disable them
   
The mask register of the interrupt controller will be updated
   according to the current priority and the new device mask.
   */

PROCEDURE GetPrioMask( priorityLevel: CARDINAL ): BITSET;

   /*
   - Gets the mask used for the priorityLevel ( only low byte significant )
   */

PROCEDURE SetPrioMask( priorityLevel: CARDINAL; mask: BITSET);

   /*
   - Sets the used for the priorityLevel ( only low byte used )
   */

PROCEDURE SaveInterruptVector(vectorNr: CARDINAL;
   VAR vector: ADDRESS);

   /*
   - Save the current value of an interrupt vector
   
in:  vectorNr  interrupt vector number
   
   out: vector   value of current interrupt vector
   */
PROCEDURE RestoreInterruptVector (vectorNr : CARDINAL;
                              vector : ADDRESS);
(*
 - Restore the value of an interrupt vector

   in:    vectorNr    interrupt vector number
   vector    value to restore (previously saved with 'SaveInterruptVector')
*)

PROCEDURE InstallHandler( process : PROCESS;
                           vectorNr : CARDINAL);
(*
 - Install an interrupt handler permanently

   in:    process    process associated with the interrupt handler
           vectorNr    interrupt vector number

   The process is associated permanently to the given interrupt vector number.
   This improves the performance of IOTRANSFER and of the implicit coroutine
   transfer that takes place when the interrupt occurs. A process may be
   associated to at most one interrupt, and at most one process may be associated
   to the same interrupt.

   'InstallHandler' must only be called after the process has been created
   (by means of NEWPROCESS) and before the process has called IOTRANSFER.
   For instance, it may be called right at the beginning of the code of the
   process. Except for the call to 'InstallHandler', the code of a permanently
   installed interrupt handler is identical to the code of a regular interrupt
   handler.
*)

PROCEDURE UninstallHandler(process: PROCESS);
(*
 - Uninstall an interrupt handler which has been installed permanently

   in:    process    process associated with the interrupt handler

   In general, there is no need to call this procedure. The LOGITECH MODULA-2 run­
   time support automatically uninstalls interrupt handlers upon termination
   of a (sub-)program.
*)

END RTSDevice.
RTSIntPROC

DEFINITION MODULE RTSIntPROC;

(*
  Constraints and Limitations:

NOTE : Each interrupt PROC must be removed before a new one
is installed within the save vector !!

    The interrupt PROC must not be in a priority module !!
    nor call a priority module procedure.

    The PROC is executed interrupt off, so it must be as short as possible.

IMPLEMENTATION :

All registers are saved; thus the Modula-2 code can be executed without
taking care of this.
The End of Interrupt is also sent to the Interrupt Controller before the
entry of the Modula-2 interrupt procedure.

Sets a PROC as interrupt service routine, all registers are preserved except the
stack, that remains the stack of the interrupted process.

NOTE: this interrupt PROC MUST BE as short as possible, and use as little stack
as possible. So will it be fast and reliable.
*)

PROCEDURE SetIntPROC(p: PROC; vector: CARDINAL);

(*
  Removes the interrupt PROC previously installed and restores the old value of the
  interrupt vector.
*)

PROCEDURE RemoveIntPROC(vector: CARDINAL);

(*
  Removes all interrupt PROC installed previously
*)

PROCEDURE FreeIntPROC;

END RTSIntPROC.
DEFINITION MODULE RTSM87;

EXPORT QUALIFIED co87Present;

VAR
    co87Present : BOOLEAN;

END RTSM87.
RTSMain

(*$A+$*)

DEFINITION MODULE RTSMain;

FROM SYSTEM IMPORT ADDRESS, BYTE, PROCESS;

EXPORT QUALIFIED
Status, GetMessage,
ProcPtr, ProcDescriptor, freeList,
OverlayKey, OverlayName, OverlayPtr, OverlayDescriptor, overlayList,
RegisterBlock, ProcessDescriptor, ProcedureKind, ActivationBlock,
PSPAddress, blockList,
Process, ProcessPtr, curProcess, activProcess, errorCode,
Terminate, InstallTermProc, CallTermProc, InstallInitProc, CallInitProc,
RTDProc, DebuggerRecord, debuggerRecord, Execute,
overlayInitProc, overlayTermProc;

(* Type definition above shall imperatively correspond to the structures *)
(* defined in RTS.INC *)

CONST
CheckValue = 0FA50H;

(* ***** Errors ***** *)

TYPE
Status = ( Normal, Warning, Stopped, Fatal,
Halt, CaseErr, StackOvf, HeapOvf,
FunctionErr, AddressOverflow, RealOverflow, RealUnderflow,
BadOperand, CardinalOverflow, IntegerOverflow, RangeErr,
DivideByZero, CoroutineEnd, CorruptedData, FileStructureErr,
IllegalInstr, IllErrorCode, TooManyInterrupts, TermListFull,
InitListFull, NoCOprocessor87 );

TYPE
Process = POINTER TO ProcessDescriptor;
ProcessPtr = POINTER TO Process;

VAR
curProcess : ProcessPtr; (* always points to activProcess *)
activProcess : Process; (* points to the ProcessDescriptor *)

(* ***** Internal informations ***** *)

VAR
PSPAddress : ADDRESS;
blockList : ADDRESS;

PROCEDURE GetMessage(status: Status; VAR message: ARRAY OF CHAR);

(* ***** Termination procedures ***** *)
TYPE

ProcPtr = POINTER TO ProcDescriptor;
ProcDescriptor = RECORD
  next : ProcPtr;
  termProc : PROC;
END;

VAR
  freeList: ProcPtr;

PROCEDURE InstallTermProc( p: PROC );
PROCEDURE CallTermProc;
PROCEDURE InstallInitProc( p: PROC );
PROCEDURE CallInitProc;

(* ***** Overlays and drivers ***** *)

TYPE

OverlayName = ARRAY [0..39] OF CHAR;
OverlayKey = ARRAY [0..2] OF CARDINAL;
OverlayPtr = POINTER TO OverlayDescriptor;
OverlayDescriptor = RECORD
  overlayKey : OverlayKey;
  overlayName : OverlayName;
  checkWord : CARDINAL;
  memoryAddr : ADDRESS;
  memorySize : CARDINAL; (* in paragraphs *)
  codeSegment : CARDINAL;
  programLevel: CARDINAL;
  termProc : ProcPtr;
  initProc : ProcPtr;
  freeList : ProcPtr;
  next,
  prev : OverlayPtr;
CASE overlay : CARDINAL OF
  0 : notUsed : ARRAY [0..14] OF CARDINAL;
  | 1,2 : loaderProcess: Process;
    priorityMask : CARDINAL;
    SP, SS, BP : CARDINAL;
    overlayStatus: Status;
    father,
    parent : OverlayPtr;
    processList : Process;
    resource : ADDRESS;
END;
layer : ADDRESS;
dummy : ARRAY [1..7] OF ADDRESS;
END(* OverlayDescriptor*);
VAR
overlayList : OverlayPtr;

(* ***** Overlay Interface procedures ***** *)

VAR
overlayInitProc : PROC;
overlayTermProc : PROC;

(* ***** Process descriptor ***** *)

TYPE
RegisterBlock = RECORD
  ES   : CARDINAL;
  DS   : CARDINAL;
  DI   : CARDINAL;
  SI   : CARDINAL;
  BP   : CARDINAL;
  dummy : CARDINAL;
  BX   : CARDINAL;
  DX   : CARDINAL;
  CX   : CARDINAL;
  AX   : CARDINAL;
  IP   : CARDINAL;
  CS   : CARDINAL;
  flag : CARDINAL;
END;

ProcedureKind = (FarProcedure, NearProcedure, NestedProcedure);
ActivationBlock = RECORD
dynamicLink: ADDRESS;
IP     : CARDINAL;
CASE ProcedureKind OF
  NearProcedure:
    | FarProcedure:
      | CS: CARDINAL;
    | NestedProcedure:
           staticLink: ADDRESS
END;
END;

ProcessDescriptor = RECORD
topStack   : POINTER TO RegisterBlock;
progStatus : Status; (* alignment mandatory *)
priorityMask : BITSET;
programLevel : CARDINAL;
heapDesc   : ADDRESS;
unused    : ADDRESS;
checkWord : CARDINAL;
bottomStack : CARDINAL; (* still used ??? *)
currOverlay : OverlayPtr;
interruptDesc : CARDINAL;
processList : Process;
dummy     : ARRAY [1..3] OF ADDRESS;
END;

(* ***** Debugger interface ***** *)
TYPE

   RTDProc = PROCEDURE(PROCESS, ADDRESS);
   (* active process and overlay list *)

DebuggerRecord = RECORD
   (* The debugger ID is initialized with the CheckValue *)
   (* The RTD initialize it to 0 *)
   debuggerId : CARDINAL;
   beforeInitCode : RTDProc;
   beforeMainCode : RTDProc;
   beforeTermProc : RTDProc;
   beforeExit : RTDProc;
END;

VAR
   debuggerRecord : DebuggerRecord;

(* ***** Program termination ***** *)

VAR
   errorCode : BYTE;

PROCEDURE Terminate( st : Status );
PROCEDURE Execute;
   (* Warning: upon entry, ES:DI is a pointer to the address of the code *)
   (* to execute !!! *)

END RTSMain.


**SimpleTerm**

```
DEFINITION MODULE SimpleTerm;

(* All the procedures above use the standard console device from MS-DOS *)
(* and thus can be redirected as DOS allows it *)

EXPORT QUALIFIED
  Read, KeyPressed, ReadAgain, ReadString,
  Write, WriteString, WriteLn;

PROCEDURE WriteString( s : ARRAY OF CHAR );
(* Displays the string s on DOS standard output *)

PROCEDURE WriteLn;
(* Displays an end of line on DOS standard output *)

PROCEDURE Write( ch: CHAR );
(* Displays the character ch on DOS standard output *)

PROCEDURE Read( VAR ch: CHAR );
(* Reads a character from DOS standard input *)

PROCEDURE KeyPressed(): BOOLEAN;
(* Tests if any character is ready from DOS standard input *)

PROCEDURE ReadString( VAR s: ARRAY OF CHAR );
(* Gets a string from DOS standard input : ESC or RETURN ends the input *)

PROCEDURE ReadAgain;

END SimpleTerm.
```
DEFINITION MODULE Sounds;

EXPORT QUALIFIED
    Sound, NoSound;

PROCEDURE Sound(hertz: INTEGER);
    (*
        Accesses the PC speaker with the frequency of hertz Hertz.
        The specified frequency will be emitted until you call the procedure NoSound.
        Frequencies between 21 and 32767 Hertz can be produced.
    *)

PROCEDURE NoSound;
    (*
        Turns off the PC speaker.
    *)

END Sounds.
DEFINITION MODULE Storage;
(*
    Standard dynamic storage management

    Storage management for dynamic variables. Calls to the Modula-2 standard
    procedures NEW and DISPOSE are translated into calls to ALLOCATE and DEALLOCATE.
    The standard way to provide these two procedures is to import them from
    this module 'Storage'.
*)
FROM SYSTEM IMPORT ADDRESS;
EXPORT QUALIFIED
    ALLOCATE, DEALLOCATE, Available,
    InstallHeap, RemoveHeap;
PROCEDURE ALLOCATE (VAR a: ADDRESS; size: CARDINAL);
(*
    - Allocate some dynamic storage (contiguous memory area).

    in:    size    number of bytes to allocate,
    out:   a       ADDRESS of allocated storage.

    The actual number of bytes allocated may be slightly greater
    than 'size', due to administrative overhead.

    Errors: If not enough space is available, or when attempting
    to allocate more than 65520 (0FF0H) bytes at once, then the calling program is
    terminated with the status 'heapovf'.
*)
PROCEDURE DEALLOCATE (VAR a: ADDRESS; size: CARDINAL);
(*
    - Release some dynamic storage (contiguous memory area).

    in:    a       ADDRESS of the area to release,
            size    number of bytes to be released,
    out:   a       set to NIL.

    The storage area released is made available for subsequent calls to ALLOCATE.
*)
PROCEDURE Available (size: CARDINAL) : BOOLEAN;
(*
    - Test whether some number of bytes could be allocated.

    in:    size    number of bytes
    out:   TRUE if ALLOCATE (p, size) would succeed.
*)
PROCEDURE InstallHeap;
(*
   - Used internally by the loader
*)
PROCEDURE RemoveHeap;
(*
   - Used internally by the loader
*)
END Storage.
Chapter 9

Strings

DEFINITION MODULE Strings;

(*
   Variable-length character strings handler.

NOTE: For most of these string handling procedures, there is the possibility of the
user not providing a variable large enough to contain the result of a string
operation. Should this possibility arise, truncation may result, as there will be
no other error notification. The implementation of this module does not cause a
range error, instead, it truncates silently.

String variables have the following characteristics:
- They are of type ARRAY OF CHAR.
- The array lower bound must be zero.
- The length of the string is the size of the string variable,
  unless a null character (0C) occurs in the string to indicate end of string.
*)

EXPORT QUALIFIED
   Assign, Insert, Delete,
   Pos, Copy, Concat, Length, CompareStr;

PROCEDURE Assign (VAR source, dest: ARRAY OF CHAR);

(*
 - Assign the contents of string variable source into string variable dest
   in: source
   out: dest
*)

PROCEDURE Insert (substr : ARRAY OF CHAR;
                  VAR str : ARRAY OF CHAR;
                  inx   : CARDINAL);

(*
 - Insert the string substr into str, starting at str[inx].
   in: substr
       str
       inx
   out: str

If inx is equal or greater than Length(str)
then substr is appended to end of dest.
*)
PROCEDURE Delete (VAR str : ARRAY OF CHAR;
    inx : CARDINAL;
    len : CARDINAL);
(*
   - Delete len characters from str, starting at str[inx].
  *)
in:   str
    inx
    len
out:  str

If inx >= Length(str) then nothing happens. If there are
not len characters to delete, characters to the end of string are deleted.
*)

PROCEDURE Pos (substr, str: ARRAY OF CHAR): CARDINAL;
(*
   - Return the index into str of the first occurrence of the substr.
  *)
in:   substr
    str

Pos returns a value greater then HIGH(str) if no
occurrence of the substring is found
*)

PROCEDURE Copy (str : ARRAY OF CHAR;
    inx : CARDINAL;
    len : CARDINAL;
    VAR result : ARRAY OF CHAR);
(*
   - Copy at most len characters from str into result.
  *)
in:   str source string,
    inx starting position in 'str',
    len maximum number of characters to copy,
out:  result copied string
*)

PROCEDURE Concat (s1, s2 : ARRAY OF CHAR;
    VAR result : ARRAY OF CHAR);
(*
   - Concatenate two strings.
  *)
in:   s1 left string,
    s2 right string,
out:  result receives left string followed by right string.
*)

PROCEDURE Length (VAR str: ARRAY OF CHAR): CARDINAL;
(*
   - Return the number of characters in a string.
  *)
in:   str
*)
PROCEDURE CompareStr (s1, s2: ARRAY OF CHAR): INTEGER;
(*
  - Compare two strings.
    in:   s1
          s2
 Returns an integer value indicating the comparison result:
    -1 if s1 is less than s2;
    0 if s1 equals s2;
    1 if s1 is greater than s2
*)
END Strings.
DEFINITION MODULE Termbase;

(*
   Terminal input/output with redirection hooks
   [Private module of the MODULA-2/86 system]

   Derived from the Lilith Modula-2 system developed by the group of Prof. N. Wirth
   at ETH Zurich, Switzerland.
*)

EXPORT QUALIFIED
   ReadProcedure, StatusProcedure, WriteProcedure,
   AssignRead, AssignWrite, UnAssignRead, UnAssignWrite,
   Read, KeyPressed, Write;

TYPE
   ReadProcedure = PROCEDURE (VAR CHAR);
   (*
    - To assign a private read procedure (for redirection of input)
      a procedure of type 'ReadProcedure' must be provided.
      This procedure returns a character from the input device.
      It waits until a character has been entered.
   *)

   StatusProcedure = PROCEDURE (): BOOLEAN;
   (*
    - To assign a private status-procedure (for redirection of input)
      a procedure of type 'StatusProcedure' must be provided.
      This procedure returns TRUE, if a character is available to read,
      FALSE otherwise.
   *)

   WriteProcedure = PROCEDURE (CHAR);
   (*
    - To assign a private write procedure (for redirection of output)
      a procedure of type 'WriteProcedure' must be provided. This is typically
      used to redirect output to a file or to the screen and a file (log file).
      Special interpretation of characters sent to the screen can be performed
      in such a private driver procedure.
   * )
PROCEDURE AssignRead (rp: ReadProcedure;  
    sp: StatusProcedure;  
    VAR done: BOOLEAN);

(*  
  - Install read and status routines for terminal input.  

  in:   rp    read-a-character procedure,  
         sp    is-character-available function,  

  out:  done TRUE if the installation was done.  

Initially the corresponding procedures of 'Keyboard' are installed.  

Subsequent assignments will be valid until the next 'UnAssignRead' is executed  
or until the (sub-)program which has installed the procedures terminates. Upon  
termination of a program, the read and status procedures allocated by that  
program are removed. Read procedures are non-sharable resources  
(see module 'Program').  

The assignments are implemented in a stack manner. When a read procedure is  
removed, the previously valid procedure becomes valid again. Up to six levels  
of re-assignment are allowed. Done = FALSE if this depth is exceeded. During  
exection of a read or status procedure, this assignments-stack is decremented,  
which allows an installed routine to call recursively Terminal.Read and/or  
Terminal.KeyPressed to activate the previously installed routine. At the lowest  
level however, the stack is not decremented.  
*)

PROCEDURE AssignWrite (wp: WriteProcedure;  
    VAR done: BOOLEAN);

(*  
  - Install write routine for terminal output.  

  in:   wp    character output procedure,  

  out:  done set TRUE if the installation was done.  

[See AssignRead above.]  
Initially the procedure Display.Write is assigned.  
*)

PROCEDURE UnAssignRead (VAR done: BOOLEAN);

(*  
  - Undo the last AssignRead by the current program.  

  out:  done set TRUE if there was something to unassign.  

The previously valid procedures become active again.  
*)

PROCEDURE UnAssignWrite (VAR done: BOOLEAN);

(*  
  - Undo the last AssignWrite by the current program.  

  out:  done set TRUE if there was something to unassign.  

The previously valid procedure becomes active again.  
*)
PROCEDURE Read (VAR ch: CHAR);
(*
* - Read a character using the current input procedure.

    out: ch the character read.

    Uses the current read-procedure, as assigned by AssignRead.
*)

PROCEDURE KeyPressed (): BOOLEAN;
(*
* - Test if a character is available from the current input.

    Uses the current status-procedure, as assigned by AssignRead.
*)

PROCEDURE Write (ch: CHAR);
(*
* - Write a character to the current output.

    in: ch character to write.

    Uses the current write-procedure as assigned by AssignWrite.
*)

END Termbase.
DEFINITION MODULE Terminal;
(*

Terminal Input/Output

This module uses the read and write procedures from module TermBase, which allows to redirect the i/o.

Derived from the Lilith Modula-2 system developed by the group of Prof. N. Wirth at ETH Zurich, Switzerland.
*)

EXPORT QUALIFIED
Read, KeyPressed, ReadAgain, ReadString,
Write, WriteString, WriteLn;

PROCEDURE Read (VAR ch: CHAR);
(*
- Read a character from the terminal.

out: ch character that was read.

The character is not echoed.
The character ASCII.cr is transformed into ASCII.EOL.
*)

PROCEDURE KeyPressed (): BOOLEAN;
(*
- Test if a character is available to Read from terminal.
*)

PROCEDURE ReadAgain;
(*
- Undo the last read: Make the last character be re-read.
*)

PROCEDURE ReadString(VAR string: ARRAY OF CHAR);
(*
- Read (with echo) a line from the terminal.

out: string receives the text of the line

Characters are accepted (and echoed) from the keyboard until <cr> is entered. The <cr> is not returned or echoed. <del> and <bs> can be used for editing. Tabs may be entered, but are expanded into blanks immediately. No other control characters may be entered.
*)
PROCEDURE Write (ch: CHAR);
(*
 - Write a character to the terminal.

 in: ch character to be written.

 If terminal output has not been redirected,
 the following interpretations are made:

 ASCII.EOL (36C) = go to beginning of next line
 ASCII.ff (14C) = clear screen and set cursor home
 ASCII.del (177C) = erase the last character on the left
 ASCII.bs (10C) = move 1 character to the left
 ASCII.cr (13C) = go to beginning of current line
 ASCII.lf (12C) = move 1 line down, same column
 *)

PROCEDURE WriteString (string: ARRAY OF CHAR);
(*
 - Write a string to the terminal.

 in: string string to be written.

 The string is terminated by its physical length or by a null character (0C).
 *)

PROCEDURE WriteLn;
(*
 - Write a new-line to the terminal.
 [Equivalent to Write(ASCII.EOL)]
 *)

END Terminal.
TIME

DEFINITION MODULE TimeDate;
(*
 * Access to the system’s date and time
 *)

EXPORT QUALIFIED
 Time,
 SetTime, GetTime,
 CompareTime, TimeToZero,
 TimeToString;

TYPE
 Time = RECORD day, minute, millisec: CARDINAL; END;
 (*
 - date and time of day
 'day' is : Bits 0..4 = day of month (1..31),
        Bits 5..8 = month of the year (1..12),
        Bits 9..15 = year - 1900.
 'minute' : is hour * 60 + minutes.
 'millisec' : is second * 1000 + milliseconds,
             starting with 0 at every minute.
 *)

PROCEDURE GetTime (VAR curTime: Time);
(*
 - Return the current date and time.
 out: curTime record containing date and time.
      On systems which do not keep date or time, 'GetTime'
      returns a pseudo-random number.
 *)

PROCEDURE SetTime (curTime: Time);
(*
 - Set the current date and time.
 in: curTime record containing date and time.
      On systems which do not keep date or time, this call has no effect.
 *)

PROCEDURE CompareTime(t1, t2: Time): INTEGER;
(*
 - compare two dates and time
 in : t1, t2 two time structures to compare
 out: return integer value indicating result of comparison
      -1 t1 < t2
      0 t1 = t2
      +1 t1 > t2
 *)
PROCEDURE TimeToZero(VAR t: Time);
(*
    - initialize time and date to zero
    out:   t      zero time  00-00-00 00:00:00
*)

PROCEDURE TimeToString(t: Time; VAR s: ARRAY OF CHAR);
(*
    - convert time into a string
    in:  t  time structure to be convert to a string
    out: s  string containing description of date and time given in t.

    The length of s should be at least 17 characters and the time will be of format:
    yy-mm-dd hh:nn:ss

    where
    yy is year (last 2 digits only)
    mm is month (1..12)
    dd is day of month (1..31)
    hh is hours (0..23)
    nn is minutes (0..59)
    ss is seconds (0..59)
*)

END TimeDate.
Notes:
9.3 Library Cross Reference

The following procedures with referenced modules are in alphabetical order — first, by procedures whose names begin in upper case — then, by those in lower case.

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDRESS</td>
<td>SYSTEM</td>
</tr>
<tr>
<td>ADR</td>
<td>SYSTEM</td>
</tr>
<tr>
<td>ALLOCATE</td>
<td>Storage</td>
</tr>
<tr>
<td>AX</td>
<td>SYSTEM</td>
</tr>
<tr>
<td>ActivationBlock</td>
<td>RTSMain</td>
</tr>
<tr>
<td>AddDec</td>
<td>Decimals</td>
</tr>
<tr>
<td>Again</td>
<td>FileSystem</td>
</tr>
<tr>
<td>Alloc</td>
<td>DynMem</td>
</tr>
<tr>
<td>Arc</td>
<td>Graphics</td>
</tr>
<tr>
<td>Assign</td>
<td>Strings</td>
</tr>
<tr>
<td>AssignRead</td>
<td>Termbase</td>
</tr>
<tr>
<td>AssignWrite</td>
<td>Termbase</td>
</tr>
<tr>
<td>Avail</td>
<td>DynMem</td>
</tr>
<tr>
<td>Available</td>
<td>Storage</td>
</tr>
<tr>
<td>Awaited</td>
<td>Processes</td>
</tr>
<tr>
<td>BASE</td>
<td>NumberConversion</td>
</tr>
<tr>
<td>BP</td>
<td>SYSTEM</td>
</tr>
<tr>
<td>BX</td>
<td>SYSTEM</td>
</tr>
<tr>
<td>BYTE</td>
<td>SYSTEM</td>
</tr>
<tr>
<td>BackgroundColor</td>
<td>Graphics</td>
</tr>
<tr>
<td>BackgroundPattern</td>
<td>Graphics</td>
</tr>
<tr>
<td>Procedure</td>
<td>Module</td>
</tr>
<tr>
<td>-------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>Black</td>
<td>Graphics</td>
</tr>
<tr>
<td>Blue</td>
<td>Graphics</td>
</tr>
<tr>
<td>Brown</td>
<td>Graphics</td>
</tr>
<tr>
<td>BusyRead RS232Code</td>
<td></td>
</tr>
<tr>
<td>BusyRead RS232Int</td>
<td></td>
</tr>
<tr>
<td>BusyRead RS232Polling</td>
<td></td>
</tr>
<tr>
<td>Button</td>
<td>Mouse</td>
</tr>
<tr>
<td>ButtonSet</td>
<td>Mouse</td>
</tr>
<tr>
<td>CODE</td>
<td>SYSTEM</td>
</tr>
<tr>
<td>CR</td>
<td>LogiFile</td>
</tr>
<tr>
<td>CS</td>
<td>SYSTEM</td>
</tr>
<tr>
<td>CURSORSHAPE</td>
<td>Graphics</td>
</tr>
<tr>
<td>CURSORSHAPEPOIN</td>
<td>Graphics</td>
</tr>
<tr>
<td>CX</td>
<td>SYSTEM</td>
</tr>
<tr>
<td>CallInitProc</td>
<td>RTSMain</td>
</tr>
<tr>
<td>CallOverlay</td>
<td>Overlay</td>
</tr>
<tr>
<td>CallOverlayInLayer</td>
<td>Overlay</td>
</tr>
<tr>
<td>CallTermProc</td>
<td>RTSMain</td>
</tr>
<tr>
<td>CancelRedirection</td>
<td>DOS31</td>
</tr>
<tr>
<td>CardToString</td>
<td>NumberConversion</td>
</tr>
<tr>
<td>ChangeDirectory</td>
<td>DiskDirect</td>
</tr>
<tr>
<td>Circle</td>
<td>Graphics</td>
</tr>
<tr>
<td>Clear</td>
<td>DurationOps</td>
</tr>
<tr>
<td>ClearWindow</td>
<td>Graphics</td>
</tr>
<tr>
<td>ClipDot</td>
<td>Graphics</td>
</tr>
<tr>
<td>ClipLine</td>
<td>Graphics</td>
</tr>
<tr>
<td>Close</td>
<td>FileSystem</td>
</tr>
<tr>
<td>Close</td>
<td>LogiFile</td>
</tr>
<tr>
<td>CloseInput</td>
<td>InOut</td>
</tr>
<tr>
<td>CloseOutput</td>
<td>InOut</td>
</tr>
<tr>
<td>Procedure</td>
<td>Module</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>ColorTable</td>
<td>Graphics</td>
</tr>
<tr>
<td>Command</td>
<td>FileSystem</td>
</tr>
<tr>
<td>CompareDec</td>
<td>Decimals</td>
</tr>
<tr>
<td>CompareStr</td>
<td>Strings</td>
</tr>
<tr>
<td>CompareTime</td>
<td>TimeDate</td>
</tr>
<tr>
<td>Concat</td>
<td>Strings</td>
</tr>
<tr>
<td>ConditionalOff</td>
<td>Mouse</td>
</tr>
<tr>
<td>ConvertCardinal</td>
<td>Conversions</td>
</tr>
<tr>
<td>ConvertHex</td>
<td>Conversions</td>
</tr>
<tr>
<td>ConvertInteger</td>
<td>Conversions</td>
</tr>
<tr>
<td>ConvertLongInt</td>
<td>Conversions</td>
</tr>
<tr>
<td>ConvertOctal</td>
<td>Conversions</td>
</tr>
<tr>
<td>Copy</td>
<td>Strings</td>
</tr>
<tr>
<td>Create</td>
<td>FileSystem</td>
</tr>
<tr>
<td>Create</td>
<td>LogiFile</td>
</tr>
<tr>
<td>CreateMedium</td>
<td>FileSystem</td>
</tr>
<tr>
<td>CreateNewFile</td>
<td>DOS3</td>
</tr>
<tr>
<td>CreateTemporary</td>
<td>DOS3</td>
</tr>
<tr>
<td>CurrentDirectory</td>
<td>DiskDirectory</td>
</tr>
<tr>
<td>CurrentDrive</td>
<td>DiskDirectory</td>
</tr>
<tr>
<td>CursorColor</td>
<td>Graphics</td>
</tr>
<tr>
<td>CursorShape</td>
<td>Graphics</td>
</tr>
<tr>
<td>CursorShow</td>
<td>Graphics</td>
</tr>
<tr>
<td>CursorVisible</td>
<td>Graphics</td>
</tr>
<tr>
<td>CursorWrap</td>
<td>Graphics</td>
</tr>
<tr>
<td>Cyan</td>
<td>Graphics</td>
</tr>
<tr>
<td>Procedure</td>
<td>Module</td>
</tr>
<tr>
<td>----------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>DEALLOCATE</td>
<td>Storage</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>Decimals</td>
</tr>
<tr>
<td>DI</td>
<td>SYSTEM</td>
</tr>
<tr>
<td>DOSAlloc</td>
<td>DOSMemory</td>
</tr>
<tr>
<td>DOSAvail</td>
<td>DOSMemory</td>
</tr>
<tr>
<td>DOSCALL</td>
<td>SYSTEM</td>
</tr>
<tr>
<td>DOSDeAlloc</td>
<td>DOSMemory</td>
</tr>
<tr>
<td>DOSGetMaxSize</td>
<td>DOSMemory</td>
</tr>
<tr>
<td>DOSSetSize</td>
<td>DOSMemory</td>
</tr>
<tr>
<td>DS</td>
<td>SYSTEM</td>
</tr>
<tr>
<td>DX</td>
<td>SYSTEM</td>
</tr>
<tr>
<td>DarkGray</td>
<td>Graphics</td>
</tr>
<tr>
<td>Date</td>
<td>Calendar</td>
</tr>
<tr>
<td>DeAlloc</td>
<td>DynMem</td>
</tr>
<tr>
<td>DeInstallOverlay</td>
<td>Overlay</td>
</tr>
<tr>
<td>DecCur</td>
<td>Decimals</td>
</tr>
<tr>
<td>DecDigits</td>
<td>Decimals</td>
</tr>
<tr>
<td>DecPoint</td>
<td>Decimals</td>
</tr>
<tr>
<td>DecRepr</td>
<td>Decimals</td>
</tr>
<tr>
<td>DecSep</td>
<td>Decimals</td>
</tr>
<tr>
<td>DecState</td>
<td>Decimals</td>
</tr>
<tr>
<td>DecStatus</td>
<td>Decimals</td>
</tr>
<tr>
<td>DecToStr</td>
<td>Decimals</td>
</tr>
<tr>
<td>DecValid</td>
<td>Decimals</td>
</tr>
<tr>
<td>Delay</td>
<td>Delay</td>
</tr>
<tr>
<td>Delete</td>
<td>Directories</td>
</tr>
<tr>
<td>Delete</td>
<td>FileSystem</td>
</tr>
<tr>
<td>Delete</td>
<td>LogiFile</td>
</tr>
<tr>
<td>Delete</td>
<td>Strings</td>
</tr>
<tr>
<td>DeltaDate</td>
<td>Calendar</td>
</tr>
</tbody>
</table>
Procedure Module

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diff</td>
<td>DurationOps</td>
</tr>
<tr>
<td>DirQuery</td>
<td>Directories</td>
</tr>
<tr>
<td>DirQueryProc</td>
<td>Directories</td>
</tr>
<tr>
<td>DirResult</td>
<td>Directories</td>
</tr>
<tr>
<td>DirectoryProc</td>
<td>FileSystem</td>
</tr>
<tr>
<td>DisableBreak</td>
<td>Break</td>
</tr>
<tr>
<td>DiskDirProc</td>
<td>DiskFiles</td>
</tr>
<tr>
<td>DiskFileProc</td>
<td>DiskFiles</td>
</tr>
<tr>
<td>DisplayCursor</td>
<td>Graphics</td>
</tr>
<tr>
<td>DisposeLayer</td>
<td>Overlay</td>
</tr>
<tr>
<td>DivDec</td>
<td>Decimals</td>
</tr>
<tr>
<td>DoIo</td>
<td>FileSystem</td>
</tr>
<tr>
<td>Done</td>
<td>InOut</td>
</tr>
<tr>
<td>Done</td>
<td>RealInOut</td>
</tr>
<tr>
<td>DosCommand</td>
<td>Exec</td>
</tr>
<tr>
<td>DosShell</td>
<td>Exec</td>
</tr>
<tr>
<td>Dot</td>
<td>Graphics</td>
</tr>
<tr>
<td>DriverInstalled</td>
<td>Mouse</td>
</tr>
<tr>
<td>Duration</td>
<td>DurationOps</td>
</tr>
<tr>
<td>EOF</td>
<td>LogiFile</td>
</tr>
<tr>
<td>EOL</td>
<td>ASCII</td>
</tr>
<tr>
<td>EOL</td>
<td>InOut</td>
</tr>
<tr>
<td>EOL</td>
<td>LogiFile</td>
</tr>
<tr>
<td>ES</td>
<td>SYSTEM</td>
</tr>
<tr>
<td>EmptyUnitSet</td>
<td>DurationOps</td>
</tr>
<tr>
<td>EnableBreak</td>
<td>Break</td>
</tr>
<tr>
<td>EndFile</td>
<td>LogiFile</td>
</tr>
<tr>
<td>Equal</td>
<td>DurationOps</td>
</tr>
<tr>
<td>EraseCursor</td>
<td>Graphics</td>
</tr>
<tr>
<td>Procedure</td>
<td>Module</td>
</tr>
<tr>
<td>----------------</td>
<td>----------------</td>
</tr>
<tr>
<td>ErrorCode</td>
<td>Overlay</td>
</tr>
<tr>
<td>Event</td>
<td>Mouse</td>
</tr>
<tr>
<td>EventHandler</td>
<td>Mouse</td>
</tr>
<tr>
<td>EventSet</td>
<td>Mouse</td>
</tr>
<tr>
<td>Execute</td>
<td>Exec</td>
</tr>
<tr>
<td>Execute</td>
<td>RTSMain</td>
</tr>
<tr>
<td>ExitToOS</td>
<td>ErrorCode</td>
</tr>
<tr>
<td>FNPartSet</td>
<td>FileNames</td>
</tr>
<tr>
<td>FNParts</td>
<td>FileNames</td>
</tr>
<tr>
<td>File</td>
<td>FileSystem</td>
</tr>
<tr>
<td>File</td>
<td>LogiFile</td>
</tr>
<tr>
<td>FileNameAndOptions</td>
<td>Options</td>
</tr>
<tr>
<td>FileNameChar</td>
<td>FileSystem</td>
</tr>
<tr>
<td>FileProc</td>
<td>FileSystem</td>
</tr>
<tr>
<td>FillRect</td>
<td>Graphics</td>
</tr>
<tr>
<td>Flag</td>
<td>FileSystem</td>
</tr>
<tr>
<td>FlagReset</td>
<td>Mouse</td>
</tr>
<tr>
<td>FlagSet</td>
<td>FileSystem</td>
</tr>
<tr>
<td>Float</td>
<td>FloatingUtilities</td>
</tr>
<tr>
<td>FloodFill</td>
<td>Graphics</td>
</tr>
<tr>
<td>ForegroundColor</td>
<td>Graphics</td>
</tr>
<tr>
<td>Format</td>
<td>DurationOp</td>
</tr>
<tr>
<td>FormatOf</td>
<td>DurationOp</td>
</tr>
<tr>
<td>Frac</td>
<td>FloatingUtilities</td>
</tr>
<tr>
<td>Procedure</td>
<td>Module</td>
</tr>
<tr>
<td>----------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>GETREG</td>
<td>SYSTEM</td>
</tr>
<tr>
<td>GetButPres</td>
<td>Mouse</td>
</tr>
<tr>
<td>GetButRel</td>
<td>Mouse</td>
</tr>
<tr>
<td>GetCursorPosition</td>
<td>Graphics</td>
</tr>
<tr>
<td>GetDeviceStatus</td>
<td>Devices</td>
</tr>
<tr>
<td>GetDeviceStatus</td>
<td>RTSDevice</td>
</tr>
<tr>
<td>GetDotColor</td>
<td>Graphics</td>
</tr>
<tr>
<td>GetErrorCode</td>
<td>ErrorCode</td>
</tr>
<tr>
<td>GetErrorCode</td>
<td>Overlay</td>
</tr>
<tr>
<td>GetExtendedError</td>
<td>DOS3</td>
</tr>
<tr>
<td>GetFileDate</td>
<td>LogiFile</td>
</tr>
<tr>
<td>GetMachineName</td>
<td>DOS31</td>
</tr>
<tr>
<td>GetMessage</td>
<td>RTSMain</td>
</tr>
<tr>
<td>GetOption</td>
<td>Options</td>
</tr>
<tr>
<td>GetPos</td>
<td>FileSystem</td>
</tr>
<tr>
<td>GetPos</td>
<td>LogiFile</td>
</tr>
<tr>
<td>GetPosBut</td>
<td>Mouse</td>
</tr>
<tr>
<td>GetPrinterSetup</td>
<td>DOS31</td>
</tr>
<tr>
<td>GetProgramSegmentPrefix</td>
<td>DOS3</td>
</tr>
<tr>
<td>GetRedirectionListEntry</td>
<td>DOS31</td>
</tr>
<tr>
<td>GetScreenExt</td>
<td>Graphics</td>
</tr>
<tr>
<td>GetScreenMode</td>
<td>Graphics</td>
</tr>
<tr>
<td>GetTime</td>
<td>TimeDate</td>
</tr>
<tr>
<td>GetWindow</td>
<td>Graphics</td>
</tr>
<tr>
<td>GraphicCursor</td>
<td>Mouse</td>
</tr>
<tr>
<td>Greater</td>
<td>DurationOps</td>
</tr>
<tr>
<td>GreaterOrEqual</td>
<td>DurationOps</td>
</tr>
<tr>
<td>Green</td>
<td>Graphics</td>
</tr>
<tr>
<td>HideCursor</td>
<td>Mouse</td>
</tr>
<tr>
<td><strong>Procedure</strong></td>
<td><strong>Module</strong></td>
</tr>
<tr>
<td>---------------</td>
<td>------------</td>
</tr>
<tr>
<td>INBYTE</td>
<td>SYSTEM</td>
</tr>
<tr>
<td>IOTRANSFER</td>
<td>SYSTEM</td>
</tr>
<tr>
<td>Init</td>
<td>Processes</td>
</tr>
<tr>
<td>Init</td>
<td>RS232Code</td>
</tr>
<tr>
<td>Init</td>
<td>RS232Int</td>
</tr>
<tr>
<td>Init</td>
<td>RS232Polling</td>
</tr>
<tr>
<td>InitDiskSystem</td>
<td>DiskFiles</td>
</tr>
<tr>
<td>Insert</td>
<td>Strings</td>
</tr>
<tr>
<td>InstallBreak</td>
<td>Break</td>
</tr>
<tr>
<td>InstallDynMem</td>
<td>DynMem</td>
</tr>
<tr>
<td>InstallHandler</td>
<td>Devices</td>
</tr>
<tr>
<td>InstallHandler</td>
<td>RTSDevice</td>
</tr>
<tr>
<td>InstallHeap</td>
<td>Storage</td>
</tr>
<tr>
<td>InstallInitProc</td>
<td>RTSMain</td>
</tr>
<tr>
<td>InstallOverlay</td>
<td>Overlay</td>
</tr>
<tr>
<td>InstallOverlayInLayer</td>
<td>Overlay</td>
</tr>
<tr>
<td>InstallTermProc</td>
<td>RTSMain</td>
</tr>
<tr>
<td>Int</td>
<td>FloatingUtilities</td>
</tr>
<tr>
<td>IntToString</td>
<td>NumberConversion</td>
</tr>
<tr>
<td>KeyPressed</td>
<td>Keyboard</td>
</tr>
<tr>
<td>KeyPressed</td>
<td>SimpleTerm</td>
</tr>
<tr>
<td>KeyPressed</td>
<td>Termbase</td>
</tr>
<tr>
<td>KeyPressed</td>
<td>Terminal</td>
</tr>
<tr>
<td>LF</td>
<td>LogiFile</td>
</tr>
<tr>
<td>LayerId</td>
<td>Overlay</td>
</tr>
<tr>
<td>Length</td>
<td>FileSystem</td>
</tr>
<tr>
<td>Length</td>
<td>Strings</td>
</tr>
<tr>
<td>LightBlue</td>
<td>Graphics</td>
</tr>
<tr>
<td>LightCyan</td>
<td>Graphics</td>
</tr>
<tr>
<td>Procedure</td>
<td>Module</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>LightGray</td>
<td>Graphics</td>
</tr>
<tr>
<td>LightGreen</td>
<td>Graphics</td>
</tr>
<tr>
<td>LightMagenta</td>
<td>Graphics</td>
</tr>
<tr>
<td>LightPenOff</td>
<td>Mouse</td>
</tr>
<tr>
<td>LightPenOn</td>
<td>Mouse</td>
</tr>
<tr>
<td>LightRed</td>
<td>Graphics</td>
</tr>
<tr>
<td>Line</td>
<td>Graphics</td>
</tr>
<tr>
<td>LockUnlockFileA</td>
<td>DOS3</td>
</tr>
<tr>
<td>LongToStr</td>
<td>NumberConversion</td>
</tr>
<tr>
<td>Lookup</td>
<td>FileSystem</td>
</tr>
<tr>
<td>LookupFile</td>
<td>Lookup</td>
</tr>
<tr>
<td>Magenta</td>
<td>Graphics</td>
</tr>
<tr>
<td>MakeDir</td>
<td>DiskDirectory</td>
</tr>
<tr>
<td>MaxBase</td>
<td>NumberConversion</td>
</tr>
<tr>
<td>MediumType</td>
<td>FileSystem</td>
</tr>
<tr>
<td>ModifyByte</td>
<td>LogiFile</td>
</tr>
<tr>
<td>ModifyWord</td>
<td>LogiFile</td>
</tr>
<tr>
<td>MoveCursor</td>
<td>Graphics</td>
</tr>
<tr>
<td>MulDec</td>
<td>Decimals</td>
</tr>
<tr>
<td>NEWPROCESS</td>
<td>SYSTEM</td>
</tr>
<tr>
<td>NUL</td>
<td>LogiFile</td>
</tr>
<tr>
<td>NamePartSet</td>
<td>Options</td>
</tr>
<tr>
<td>NameParts</td>
<td>Options</td>
</tr>
<tr>
<td>NegDec</td>
<td>Decimals</td>
</tr>
<tr>
<td>NewLayer</td>
<td>Overlay</td>
</tr>
<tr>
<td>NoSound</td>
<td>Sounds</td>
</tr>
<tr>
<td>NumToString</td>
<td>NumberConversion</td>
</tr>
</tbody>
</table>
## Chapter 9

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUTBYTE</td>
<td>SYSTEM</td>
</tr>
<tr>
<td>Open</td>
<td>LogiFile</td>
</tr>
<tr>
<td>OpenInput</td>
<td>InOut</td>
</tr>
<tr>
<td>OpenMode</td>
<td>LogiFile</td>
</tr>
<tr>
<td>OpenOutput</td>
<td>InOut</td>
</tr>
<tr>
<td>OverlayDescriptor</td>
<td>RTSMain</td>
</tr>
<tr>
<td>OverlayId</td>
<td>Overlay</td>
</tr>
<tr>
<td>OverlayKey</td>
<td>RTSMain</td>
</tr>
<tr>
<td>OverlayName</td>
<td>RTSMain</td>
</tr>
<tr>
<td>OverlayPtr</td>
<td>RTSMain</td>
</tr>
<tr>
<td>PROCESS</td>
<td>SYSTEM</td>
</tr>
<tr>
<td>PSPAddress</td>
<td>RTSMain</td>
</tr>
<tr>
<td>Palette</td>
<td>Graphics</td>
</tr>
<tr>
<td>Pattern</td>
<td>Graphics</td>
</tr>
<tr>
<td>Pos</td>
<td>Strings</td>
</tr>
<tr>
<td>ProcDescriptor</td>
<td>RTSMain</td>
</tr>
<tr>
<td>ProcPtr</td>
<td>RTSMain</td>
</tr>
<tr>
<td>ProcedureKind</td>
<td>RTSMain</td>
</tr>
<tr>
<td>Process</td>
<td>RTSMain</td>
</tr>
<tr>
<td>ProcessDescriptor</td>
<td>RTSMain</td>
</tr>
<tr>
<td>ProcessPtr</td>
<td>RTSMain</td>
</tr>
<tr>
<td>RTDProc</td>
<td>RTSMain</td>
</tr>
<tr>
<td>Read</td>
<td>InOut</td>
</tr>
<tr>
<td>Read</td>
<td>Keyboard</td>
</tr>
<tr>
<td>Read</td>
<td>RS232Code</td>
</tr>
<tr>
<td>Read</td>
<td>RS232Int</td>
</tr>
<tr>
<td>Read</td>
<td>RS232Polling</td>
</tr>
<tr>
<td>Read</td>
<td>SimpleTerm</td>
</tr>
<tr>
<td>Read</td>
<td>Termbase</td>
</tr>
<tr>
<td>Read</td>
<td>Terminal</td>
</tr>
</tbody>
</table>

360
### Libraries

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReadAgain</td>
<td>SimpleTerm</td>
</tr>
<tr>
<td>ReadAgain</td>
<td>Terminal</td>
</tr>
<tr>
<td>ReadByte</td>
<td>FileSystem</td>
</tr>
<tr>
<td>ReadByte</td>
<td>LogiFile</td>
</tr>
<tr>
<td>ReadCard</td>
<td>InOut</td>
</tr>
<tr>
<td>ReadCardinal</td>
<td>CardinalIO</td>
</tr>
<tr>
<td>ReadChar</td>
<td>FileSystem</td>
</tr>
<tr>
<td>ReadChar</td>
<td>LogiFile</td>
</tr>
<tr>
<td>ReadFileName</td>
<td>FileNames</td>
</tr>
<tr>
<td>ReadHex</td>
<td>CardinalIO</td>
</tr>
<tr>
<td>ReadInt</td>
<td>InOut</td>
</tr>
<tr>
<td>ReadLongInt</td>
<td>LongIO</td>
</tr>
<tr>
<td>ReadMotionCount</td>
<td>Mouse</td>
</tr>
<tr>
<td>ReadNBytes</td>
<td>FileSystem</td>
</tr>
<tr>
<td>ReadNBytes</td>
<td>LogiFile</td>
</tr>
<tr>
<td>ReadProcedure</td>
<td>Termbase</td>
</tr>
<tr>
<td>ReadReal</td>
<td>RealInOut</td>
</tr>
<tr>
<td>ReadString</td>
<td>InOut</td>
</tr>
<tr>
<td>ReadString</td>
<td>SimpleTerm</td>
</tr>
<tr>
<td>ReadString</td>
<td>Terminal</td>
</tr>
<tr>
<td>ReadWord</td>
<td>FileSystem</td>
</tr>
<tr>
<td>ReadWord</td>
<td>LogiFile</td>
</tr>
<tr>
<td>ReadWrd</td>
<td>InOut</td>
</tr>
<tr>
<td>RealToString</td>
<td>RealConversion</td>
</tr>
<tr>
<td>Red</td>
<td>Graphics</td>
</tr>
<tr>
<td>RedirectDevice</td>
<td>DOS31</td>
</tr>
<tr>
<td>RegisterBlock</td>
<td>RTSMain</td>
</tr>
<tr>
<td>Remainder</td>
<td>Decimals</td>
</tr>
<tr>
<td>RemoveDir</td>
<td>DiskDirectory</td>
</tr>
<tr>
<td>RemoveHeap</td>
<td>Storage</td>
</tr>
<tr>
<td>RemoveMedium</td>
<td>FileSystem</td>
</tr>
</tbody>
</table>
## Procedure | Module
--- | ---
Rename | Directories
Rename | FileSystem
Reset | FileSystem
Reset | LogiFile
ResetDiskSys | DiskDirectory
ResetDrive | DiskDirectory
Response | FileSystem
RestoreInterrupt | Devices
RestoreInterrupt | RTSDevice
RestorePicture | Graphics
Round | FloatingUtilities
Run | Exec
SEND | Processes
SETREG | SYSTEM
SI | SYSTEM
SIGNAL | Processes
SIZE | SYSTEM
SP | SYSTEM
SWI | SYSTEM
SaveInterruptVector | Devices
SaveInterruptVector | RTSDevice
SavePicture | Graphics
ScreenMode | Graphics
SelectDrive | DiskDirectory
SetCursorPos | Mouse
SetDeviceStatus | Devices
SetDeviceStatus | RTSDevice
SetErrorCode | ErrorCode
SetEventHandler | Mouse
SetGraphicCursor | Mouse
SetHorizontalLimits | Mouse
<table>
<thead>
<tr>
<th>Procedure</th>
<th>Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetMickeyPerPixel</td>
<td>Mouse</td>
</tr>
<tr>
<td>SetModify</td>
<td>FileSystem</td>
</tr>
<tr>
<td>SetOpen</td>
<td>FileSystem</td>
</tr>
<tr>
<td>SetPos</td>
<td>FileSystem</td>
</tr>
<tr>
<td>SetPos</td>
<td>LogiFile</td>
</tr>
<tr>
<td>SetPrinterSetup</td>
<td>DOS31</td>
</tr>
<tr>
<td>SetRead</td>
<td>FileSystem</td>
</tr>
<tr>
<td>SetSpeedThreshold</td>
<td>Mouse</td>
</tr>
<tr>
<td>SetTextCursor</td>
<td>Mouse</td>
</tr>
<tr>
<td>SetTime</td>
<td>TimeDate</td>
</tr>
<tr>
<td>SetVerticalLimit</td>
<td>Mouse</td>
</tr>
<tr>
<td>SetWrite</td>
<td>FileSystem</td>
</tr>
<tr>
<td>ShowCursor</td>
<td>Mouse</td>
</tr>
<tr>
<td>Sound</td>
<td>Sounds</td>
</tr>
<tr>
<td>StartProcess</td>
<td>Processes</td>
</tr>
<tr>
<td>StartReading</td>
<td>RS232Code</td>
</tr>
<tr>
<td>StartReading</td>
<td>RS232Int</td>
</tr>
<tr>
<td>Status</td>
<td>RTSMain</td>
</tr>
<tr>
<td>StatusProcedure</td>
<td>Termbase</td>
</tr>
<tr>
<td>StopReading</td>
<td>RS232Code</td>
</tr>
<tr>
<td>StopReading</td>
<td>RS232Int</td>
</tr>
<tr>
<td>StrToDec</td>
<td>Decimals</td>
</tr>
<tr>
<td>StringToCard</td>
<td>NumberConv</td>
</tr>
<tr>
<td>StringToInt</td>
<td>NumberConv</td>
</tr>
<tr>
<td>StringToLongInt</td>
<td>NumberConv</td>
</tr>
<tr>
<td>StringToNum</td>
<td>NumberConv</td>
</tr>
<tr>
<td>StringToReal</td>
<td>RealConversions</td>
</tr>
<tr>
<td>SubDec</td>
<td>Decimals</td>
</tr>
<tr>
<td>Sum</td>
<td>DurationOps</td>
</tr>
<tr>
<td>Procedure</td>
<td>Module</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>TRANSFER</td>
<td>SYSTEM</td>
</tr>
<tr>
<td>TSIZE</td>
<td>SYSTEM</td>
</tr>
<tr>
<td>Terminate</td>
<td>RTSMain</td>
</tr>
<tr>
<td>Termination</td>
<td>Options</td>
</tr>
<tr>
<td>Text</td>
<td>Graphics</td>
</tr>
<tr>
<td>Time</td>
<td>TimeDate</td>
</tr>
<tr>
<td>TimeToDate</td>
<td>Calendar</td>
</tr>
<tr>
<td>TimeToString</td>
<td>TimeDate</td>
</tr>
<tr>
<td>TimeToZero</td>
<td>TimeDate</td>
</tr>
<tr>
<td>Trunc</td>
<td>FloatingUtilities</td>
</tr>
<tr>
<td>UnAssignRead</td>
<td>Termbase</td>
</tr>
<tr>
<td>UnAssignWrite</td>
<td>Termbase</td>
</tr>
<tr>
<td>UnInstallBreak</td>
<td>Break</td>
</tr>
<tr>
<td>UninstallHandle</td>
<td>Devices</td>
</tr>
<tr>
<td>UninstallHandle</td>
<td>RTSDevice</td>
</tr>
<tr>
<td>Unit</td>
<td>DurationOps</td>
</tr>
<tr>
<td>UnitSet</td>
<td>DurationOps</td>
</tr>
<tr>
<td>WAIT</td>
<td>Processes</td>
</tr>
<tr>
<td>WORD</td>
<td>SYSTEM</td>
</tr>
<tr>
<td>White</td>
<td>Graphics</td>
</tr>
<tr>
<td>Window</td>
<td>Graphics</td>
</tr>
<tr>
<td>Write</td>
<td>Display</td>
</tr>
<tr>
<td>Write</td>
<td>InOut</td>
</tr>
<tr>
<td>Write</td>
<td>RS232Code</td>
</tr>
<tr>
<td>Write</td>
<td>RS232Int</td>
</tr>
<tr>
<td>Write</td>
<td>RS232Polling</td>
</tr>
<tr>
<td>Write</td>
<td>SimpleTerm</td>
</tr>
<tr>
<td>Write</td>
<td>Termbase</td>
</tr>
<tr>
<td>Write</td>
<td>Terminal</td>
</tr>
</tbody>
</table>

364
<table>
<thead>
<tr>
<th>Procedure</th>
<th>Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>WriteByte</td>
<td>FileSystem</td>
</tr>
<tr>
<td>WriteByte</td>
<td>LogiFile</td>
</tr>
<tr>
<td>WriteCard</td>
<td>InOut</td>
</tr>
<tr>
<td>WriteCardinal</td>
<td>CardinalIO</td>
</tr>
<tr>
<td>WriteChar</td>
<td>FileSystem</td>
</tr>
<tr>
<td>WriteChar</td>
<td>LogiFile</td>
</tr>
<tr>
<td>WriteHex</td>
<td>CardinalIO</td>
</tr>
<tr>
<td>WriteHex</td>
<td>InOut</td>
</tr>
<tr>
<td>WriteInt</td>
<td>InOut</td>
</tr>
<tr>
<td>WriteLn</td>
<td>InOut</td>
</tr>
<tr>
<td>WriteLn</td>
<td>SimpleTerm</td>
</tr>
<tr>
<td>WriteLn</td>
<td>Terminal</td>
</tr>
<tr>
<td>WriteLongInt</td>
<td>LongIO</td>
</tr>
<tr>
<td>WriteNBytes</td>
<td>FileSystem</td>
</tr>
<tr>
<td>WriteNBytes</td>
<td>LogiFile</td>
</tr>
<tr>
<td>WriteOct</td>
<td>InOut</td>
</tr>
<tr>
<td>WriteProcedure</td>
<td>Termbase</td>
</tr>
<tr>
<td>WriteReal</td>
<td>RealInOut</td>
</tr>
<tr>
<td>WriteRealOct</td>
<td>RealInOut</td>
</tr>
<tr>
<td>WriteResponse</td>
<td>FileMessage</td>
</tr>
<tr>
<td>WriteString</td>
<td>InOut</td>
</tr>
<tr>
<td>WriteString</td>
<td>SimpleTerm</td>
</tr>
<tr>
<td>WriteString</td>
<td>Terminal</td>
</tr>
<tr>
<td>WriteWord</td>
<td>FileSystem</td>
</tr>
<tr>
<td>WriteWord</td>
<td>LogiFile</td>
</tr>
<tr>
<td>WriteWrd</td>
<td>InOut</td>
</tr>
</tbody>
</table>

Yellow          | Graphics     |
## Chapter 9

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>ack</td>
<td>ASCII</td>
</tr>
<tr>
<td>activProcess</td>
<td>RTSMain</td>
</tr>
<tr>
<td>addProcess</td>
<td>RTSCoroutine</td>
</tr>
<tr>
<td>arctan</td>
<td>MathLib0</td>
</tr>
<tr>
<td>bel</td>
<td>ASCII</td>
</tr>
<tr>
<td>blockList</td>
<td>RTSMain</td>
</tr>
<tr>
<td>bs</td>
<td>ASCII</td>
</tr>
<tr>
<td>can</td>
<td>ASCII</td>
</tr>
<tr>
<td>co87Present</td>
<td>RTSM87</td>
</tr>
<tr>
<td>cos</td>
<td>MathLib0</td>
</tr>
<tr>
<td>cr</td>
<td>ASCII</td>
</tr>
<tr>
<td>curProcess</td>
<td>RTSMain</td>
</tr>
<tr>
<td>cursorHeight</td>
<td>Graphics</td>
</tr>
<tr>
<td>cursorWidth</td>
<td>Graphics</td>
</tr>
<tr>
<td>dc1</td>
<td>ASCII</td>
</tr>
<tr>
<td>dc2</td>
<td>ASCII</td>
</tr>
<tr>
<td>dc3</td>
<td>ASCII</td>
</tr>
<tr>
<td>dc4</td>
<td>ASCII</td>
</tr>
<tr>
<td>debuggerRecord</td>
<td>RTSMain</td>
</tr>
<tr>
<td>del</td>
<td>ASCII</td>
</tr>
<tr>
<td>dle</td>
<td>ASCII</td>
</tr>
<tr>
<td>em</td>
<td>ASCII</td>
</tr>
<tr>
<td>enq</td>
<td>ASCII</td>
</tr>
<tr>
<td>entier</td>
<td>MathLib0</td>
</tr>
<tr>
<td>eot</td>
<td>ASCII</td>
</tr>
<tr>
<td>Procedure</td>
<td>Module</td>
</tr>
<tr>
<td>---------------</td>
<td>--------------</td>
</tr>
<tr>
<td>errorCode</td>
<td>RTSMain</td>
</tr>
<tr>
<td>esc</td>
<td>ASCII</td>
</tr>
<tr>
<td>etb</td>
<td>ASCII</td>
</tr>
<tr>
<td>etx</td>
<td>ASCII</td>
</tr>
<tr>
<td>exp</td>
<td>MathLib0</td>
</tr>
<tr>
<td>ff</td>
<td>ASCII</td>
</tr>
<tr>
<td>freeList</td>
<td>RTSMain</td>
</tr>
<tr>
<td>fs</td>
<td>ASCII</td>
</tr>
<tr>
<td>gphCMedRes</td>
<td>Graphics</td>
</tr>
<tr>
<td>gphHiRes</td>
<td>Graphics</td>
</tr>
<tr>
<td>gphMedRes</td>
<td>Graphics</td>
</tr>
<tr>
<td>gs</td>
<td>ASCII</td>
</tr>
<tr>
<td>ht</td>
<td>ASCII</td>
</tr>
<tr>
<td>in</td>
<td>InOut</td>
</tr>
<tr>
<td>if</td>
<td>ASCII</td>
</tr>
<tr>
<td>ln</td>
<td>MathLib0</td>
</tr>
<tr>
<td>nak</td>
<td>ASCII</td>
</tr>
<tr>
<td>nul</td>
<td>ASCII</td>
</tr>
<tr>
<td>out</td>
<td>InOut</td>
</tr>
<tr>
<td>overlayInitProc</td>
<td>RTSMain</td>
</tr>
<tr>
<td>overlayList</td>
<td>RTSMain</td>
</tr>
<tr>
<td>overlayTermProc</td>
<td>RTSMain</td>
</tr>
<tr>
<td>Procedure</td>
<td>Module</td>
</tr>
<tr>
<td>-----------</td>
<td>--------------</td>
</tr>
<tr>
<td>real</td>
<td>MathLib0</td>
</tr>
<tr>
<td>rs</td>
<td>ASCII</td>
</tr>
<tr>
<td>si</td>
<td>ASCII</td>
</tr>
<tr>
<td>sin</td>
<td>MathLib0</td>
</tr>
<tr>
<td>so</td>
<td>ASCII</td>
</tr>
<tr>
<td>soh</td>
<td>ASCII</td>
</tr>
<tr>
<td>sqrt</td>
<td>MathLib0</td>
</tr>
<tr>
<td>stx</td>
<td>ASCII</td>
</tr>
<tr>
<td>sub</td>
<td>ASCII</td>
</tr>
<tr>
<td>syn</td>
<td>ASCII</td>
</tr>
<tr>
<td>termCH</td>
<td>InOut</td>
</tr>
<tr>
<td>txtCHiRes</td>
<td>Graphics</td>
</tr>
<tr>
<td>txtCMedRes</td>
<td>Graphics</td>
</tr>
<tr>
<td>txtHiRes</td>
<td>Graphics</td>
</tr>
<tr>
<td>txtMedRes</td>
<td>Graphics</td>
</tr>
<tr>
<td>us</td>
<td>ASCII</td>
</tr>
<tr>
<td>vt</td>
<td>ASCII</td>
</tr>
</tbody>
</table>
Appendix A
Modula-2 Bibliography

Books

Wirth, Niklaus
*Programming in Modula-2*, Third Edition
Springer-Verlag, New York, Berlin, 1985

Adams, J. Mack, Phillippe J. Gebrini, and Barry L. Kurtz
*An Introduction to Computer Science with Modula-2*
D. C. Heath and Co., Lexington, MA, 1988

Beidler, John, and Paul Jackowitz
*Modula-2*
PWS, Boston, MA 1986

Chirlian, Paul M.
*Introduction to Modula-2*
Matrix Publishing, Beaverton, OR, 1984

Christian, Kaare
*A Guide to Modula-2*
Springer-Verlag, New York, Berlin, 1984
Appendix A

Ford, Gary A., and Richard Wiener
*Modula-2 A Software Development Approach*
John Wiley and Sons, New York, 1985

Gleaves, Richard
*Modula-2 for PASCAL PROGRAMMERS*
Springer-Verlag, New York, 1985

Greenfield, S.B.
*Invitation to Modula-2*
Petrocelli, Princeton, NJ, 1985

Joyce, Edward
*Modula-2: A Seafarer's Guide and Shipyard Manual*
Addison Wesley, Reading, MA, 1985

Kaplan, I., and M. Miller
*Modula-2 Programming*
Hayden, Hasbrouck Heights, NJ, 1986

Kaplan, I., and M. Miller
*Programming in Modula-2*

Kelly-Boole, Stan
*Modula-2 Primer*
Howard Sams, Indianapolis, IN, 1987

King, K. N.
*Modula-2: A Complete Guide*
D. C. Heath and Co., Lexington MA, 1988

Knepley, E., and R. Platt
*Modula-2 Programming*
Reston, Reston, VA, 1985

Moore, J.B., and K.N. McKay
*Modula-2: Text and Reference*

Messer, P. and I. Marshall
*Modula-2: Constructive Program Development*
Blackwell, 1986
Ogilvie, J.W.L.  
*Modula-2 Programming*  

Pinson, Lewis  
*Introduction to Computer Science with Modula-2*  
John Wiley and Sons, New York

Sale, Arthur  
*Modula-2: Discipline & Design*  
Addison-Wesley, Sydney, Australia, 1986

Sawyer, B. and D. Foster  
*Programming Expert Systems in Modula-2*  
Wiley, New York, 1986

Schildt, H.  
*Modula-2 Made Easy*  
Osborne McGraw-Hill, Berkeley, CA, 1986

Sincovec, R., and R. Wiener  
*Data Structures with Modula-2*  
John Wiley & Sons, 1986

Sutcliffe, Richard J.  
*Introduction to Programming Using Modula-2*  
Merrill, Columbus, OH, 1987

Thalmann, Daniel  
*Modula-2, An Introduction*  
Springer Verlag, New York, Berlin, 1985

Ural, S  
*Introduction to Programming with Modula-2*  
Harper & Row, New York, 1987

Walker, B.K.  
*Modula-2 Programming with Data Structures*  
Wadsworth, Belmont, CA, 1986

Ward, Terry A.  
*Advanced Programming Techniques in Modula-2*  
Scott, Foresman, Glenview, IL, 1987
Appendix A

Wiatrowski, Calude A. and Richard S. Wiener
*From C to Modula-2 and Back ... Bridging the Language Gap*
John Wiley and Sons, New York, 1987

Wiener, Richard
*Data Structures Using Modula-2*
John Wiley and Sons, New York

Wiener, Richard
*Modula-2 Wizard: A Programmer's Reference*
John Wiley and Sons, New York, 1986

Wiener, Richard, and Richard Sincovec
*Data Structure Components in Modula-2*
John Wiley and Sons, New York, 1986

Wiener, Richard, and Richard Sincovec
*Software Engineering with Modula-2 and ADA*
John Wiley and Sons, New York, 1984

Wirth, Niklaus
*Algorithm and Data structures*
Magazines

Journal of Pascal, Ada & Modula-2
John Wiley & Sons
Subscription Department
605 Third Avenue
New York, NY 10158

The MODUS Quarterly (formerly Modula-2 News)
Modula-2 Users Association (MODUS)
P.O. Box 51778
Palo Alto, CA 94303

Structured Language World
Springer-Verlag New York, Inc.
175 Fifth Avenue
New York, NY 10010
User Groups

Modula-2 Users Association (MODUS)
c/o Pacific Systems Group
P.O. Box 51778
Palo Alto, CA 94303

Modula-2 Special Interest Group (SIG)
USUS (USCD Pascal System Users' Society)
P.O. Box 1148
La Jolla, CA 92038
Appendix B
Memory Organization & Run Time Description

B.1 Global Memory Organization

Global memory organization when executing a LOGITECH Modula-2 program is shown in Figure B-1.

After loading a program, run-time support creates the main process which will then execute the program. It then transfers control to this main process.
Appendix B

The memory location of START_MEMORY in Figure B-1 is the address of the first free paragraph after the code and data of a Modula-2 program. From this address on, chunks of memory can be allocated, through DOS, to the application program in order to implement the heap. Such chunks may not be contiguous.

Figure B-1: Global Memory Organization
B.2 Subprograms and Resident Overlays

The following schema is supported by the LOGITECH Modula-2 Development System only if the LOGITECH Linker is used to bind the application.

An overlay is a piece of executable code that can be loaded and executed. An overlay can in turn load another overlay on top of itself.

Two different kinds of overlay are supported: subprogram overlay and resident overlay.

After execution, a subprogram overlay is automatically unloaded from the memory, while a resident overlay gets unloaded from the memory — either upon explicit request, or when the parent overlay subprogram gets loaded. The parent overlay subprogram is the first overlay subprogram found going back in the loading chain. (We will refer to an overlay as "layer" from time to time.)

- The main program is a subprogram overlay; it is also called the BASE LAYER.
- A Library module called Overlay is the overlay manager. Overlay contains procedures that can load and execute subprogram overlays or install and de-install resident overlays.
- The number of layers that can be loaded on top of each other is limited only by the available memory.
- The programmer is not concerned where an overlay is loaded. The overlay manager takes care of finding the place and of loading the new layer.
**Figure B-2** shows a possible memory organization when a base layer and two overlays are loaded.

<table>
<thead>
<tr>
<th>Interrupt Vector Table</th>
<th>0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating System</td>
<td>0400</td>
</tr>
<tr>
<td>Modula-2 Program Code</td>
<td></td>
</tr>
<tr>
<td>Modula-2 Program Data</td>
<td></td>
</tr>
<tr>
<td>Stack</td>
<td>START_MEMORY</td>
</tr>
<tr>
<td>Overlay 1</td>
<td></td>
</tr>
<tr>
<td>Heap Block</td>
<td></td>
</tr>
<tr>
<td>Free Memory</td>
<td></td>
</tr>
<tr>
<td>Overlay 2</td>
<td></td>
</tr>
<tr>
<td>Heap Block</td>
<td></td>
</tr>
<tr>
<td>Free memory</td>
<td>TOP_OF_MEMORY</td>
</tr>
</tbody>
</table>

**Figure B-2**: Memory Organization for Overlays
B.3 Program Execution

To run a LOGITECH Modula-2 program, you enter the name of the program on the DOS command line, concluding it with \[Cd\]. The program name can be preceded by a prefix with the drive and/or directory name, in which case the program is loaded from the specified drive or directory.

For example:

```
C:\TEMP> \otherdir\example\  
The program worked! (Hit a key)  
C:\TEMP>
```
B.4 Processes

When starting a Modula-2 program, LOGITECH Modula-2 Run-Time Support automatically creates the main process. This default process gets the stack as its workspace. Figure B-1 and Figure B-2 above, illustrate the organization of the workspace of the main process.

When a new process is created by a call to procedure NEWPROCESS from module SYSTEM, it must be assigned a workspace. This region of memory must be explicitly defined by the programmer. It is usually a variable, owned by the parent process. Such a variable can be global, for example, an ARRAY declared at the level of a module. It can be a dynamic variable, created on the heap by a call to NEW, or it can also be a variable declared local to a procedure, which is allocated on the stack. If a non-global variable is used, make sure the process does not have a longer lifetime than its workspace!

The heap of the process is allocated by DOS in DOS free memory and is consequently shared by any other process.

![Process Workspace Diagram](image)

Figure B-3: Process Workspace
The process descriptor of a process created by NEWPROCESS starts at the first paragraph boundary in the workspace of the process. Approximately 200 bytes are needed for the process descriptor plus the initial stack. In addition, at any point in time, there should be approximately 200 bytes of free memory in the workspace of the process. This memory may be needed when an interrupt occurs during the execution of the process, because the standard interrupt handlers of the operating system use the current stack. This is true for any program and is not related in any way to the use of IOTRANSFER in Modula-2. This brings the minimum size of a workspace to approximately 400 bytes, assuming that the corresponding process does nothing at all!

---NOTE---

If the workspace of the new process is too small, and does not allow a reasonable initialization, the process that calls NEWPROCESS will be terminated with a stack overflow.

For any procedure call, some space on the stack is needed. Also, any call to the operating system, needs approximately 100 bytes of stack space. The standard LOGITECH Modula-2 library implements all input and output functions by means of calls to the operating system. Taking everything into account, even the most simple process that does terminal or file I/O, requires a workspace of at least 2K. For more complicated processes, a larger workspace is required.

The workspace of a process must be large enough to hold its stack. If the process stack grows too much, the program containing this process is aborted with a stack overflow. The maximum size of a process workspace is approximately 64K.
Local variables are declared inside a procedure. They are allocated with the procedure activation record on the stack of the process that executes the procedure call. The variables of modules which are declared local to a procedure are allocated at the same place. The procedure parameters are also allocated inside the procedure activation record.

Because the procedure activation record only exists while the procedure is being executed, the lifetime of local variables and of procedure parameters is limited to the duration of the procedure call. Every time a procedure is called, a new instance of its activation record is created. If the procedure is recursive, or is called by more than one process at the same time, several instances of its procedure activation record will exist.

Global variables that are declared in modules which are not local to a procedure come into existence when the program or subprogram that contains this module is loaded. Their lifetime is limited by the lifetime of the program or subprogram to which they belong.

There is only one instance of the global variables of a module. Global variables are shared by all procedures and processes of a program. A program that calls a subprogram also shares its global variables with the subprogram.

Variables are allocated in the order of their declaration: the first variable has the lowest address. Alignment of variables depends on the setting of the alignment option. If alignment is on, variables with a size greater than one byte are allocated on even addresses. Byte sized variables can be allocated on odd addresses. Alignment can improve the performance of the program for an 8086 based system.

The maximum allowance for all local variables of a procedure is approximately 32K bytes. The same limit exists for the total size of all parameters of a procedure. In practice, however, these sizes are much more limited by the size of the stack, which cannot exceed 64K bytes. The limit for the total size of all global variables declared in one program module or in one implementation module, including those declared in the corresponding definition module, is approximately 64K bytes. The total size of the global variables in all modules of a program is only limited by DOS.
B.6 The Heap

The library module Storage implements the heap management. On each chunk of memory (typically 10 - 40 K) it does its own memory management. This is transparent and you can directly get and release memory by means of ALLOCATE and DEALLOCATE.

Modula-2 provides the standard procedures NEW and DISPOSE to allocate and deallocate dynamic memory. The compiler maps calls to these procedures to calls of the procedures ALLOCATE and DEALLOCATE. When using NEW or DISPOSE in a module, procedures ALLOCATE or DEALLOCATE must be imported or declared in that module. The standard way is to import these procedures from the library module Storage. However, a program may declare and use its own versions of ALLOCATE or DEALLOCATE. In this way, a program can implement its own heap management. In general, the strategy for allocation and deallocation of dynamic memory will then differ from the default strategy provided by module Storage.

B.7 The Stack

The stack holds different kinds of data:

- Procedure activation records
- Temporary values during the evaluation of an expression
- Other temporary data

Every process owns its private stack which is part of its workspace. Upon creation of a process by a call to NEWPROCESS the stack is set such that the first word pushed onto the stack occupies the last word at the highest even address in the workspace. The stack grows from the end of the workspace toward lower addresses.

Maximum stack size is 64K bytes. However, in most applications the workspace needed by a process is less than 64K. Therefore, the stack size is usually limited by the size of the workspace and the occupation of the heap.

The default size of the main process stack is a fixed value (8000 bytes); you can change this at link time by using a different size. Refer to the section on linker options.
B.8 The Procedure Activation Record

Each time there is a procedure call, a new procedure activation record is created on the stack of the current process. Depending on whether 8086/8088 or 80186/80286 code is generated, the activation record format differs slightly. The procedure activation record contains the following information (see also Figure B-4 and Figure B-5 below):

- **Procedure parameters**: Are pushed, if they exist, onto the stack in the order they are declared. Because the stack grows toward lower addresses, the last parameter is found at the lowest address.

- **Static link**: A pointer, within the same stack, to another procedure activation record which constitutes the static environment of the procedure. The static link can find variables or parameters in the static environment of the procedure. The static link is only for procedures which are declared nested inside of another procedure. The static environment consists of the parameters and variables which are declared in the embedding procedure(s). When code is generated for 80186/80286, the static link does not exist, but is implemented as a display.

- **Return address**: If the procedure was activated by a near procedure call, the return address is an offset value only, which corresponds to the instruction pointer. If the procedure was activated by a far call, there is also a segment value which corresponds to the code segment of the calling procedure.

- **Dynamic link**: Points to the previous procedure activation record within the same stack.

- **Display (186/286 only)**: A table of pointers, within the same stack, to the other procedure activation records which make up the static environment of the procedure. The number of table entries corresponds to the lexical nesting level of the current procedure. The display table is used to find variables or parameters in the static environment of the procedure. The display is only generated if the code generation option for 80186/80286 was selected. For 8086/8088 code, access to the static environment is implemented by the static link.

- **Local data**: All the variables declared inside the procedure.
<table>
<thead>
<tr>
<th>Memory Organization</th>
</tr>
</thead>
</table>

### Figure B-4: Procedure Activation Record for 8086/8088

- **Local Data of Procedure**
- **Dynamic Link**
- **Return Offset**
- **Return Code Segment**
- **Static Link**
- **Last Parameter**
  - ...
- **First Parameter**

<table>
<thead>
<tr>
<th>Low Addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack Pointer</td>
</tr>
<tr>
<td>Base Pointer</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>High Addresses</th>
</tr>
</thead>
</table>

---

387
### Local Data of Procedure

<table>
<thead>
<tr>
<th>Display</th>
<th>Dynamic Link</th>
<th>Return Offset (IP)</th>
<th>Return Code Segment (CS)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>First Parameter</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Last Parameter</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Low Addresses
- Stack Pointer
- Base Pointer

#### High Addresses

---

**Figure B-5: Procedure Activation Record for 80186/80286**
B.9 Procedure Calling Conventions

A procedure is called with a far intersegment call if at least one of the following conditions is true:

- It is imported from another separately compiled module.
- It is exported from a definition module.
- It is used in an assignment to a procedure variable or as a procedure parameter.
- It is used as the body (starting point) of a process upon a call to NEWPROCESS.

If none of these conditions is true, the procedure is called with a near intrasegment call.

Before a procedure call occurs, this prologue is executed in the calling procedure:

- Parameters, if any, are pushed on the stack in the same order as they are declared. A value parameter on one byte occupies two bytes on the stack, with the value in the low byte and an undefined high byte.
- for 8086/8088 only:
  - If the called procedure is declared nested inside of the calling procedure, the static link is pushed on the stack.

This sets up the first part of the procedure activation record. The remainder is set up inside the called procedure.

Now, the procedure is called and gains control. It executes the following procedure prologue, to prepare the rest of the procedure activation record:

- An optional call to the run-time support routine stack check is executed. BX contains the number of bytes on the stack needed by the current procedure. This amount includes the size of local variables and the stack space needed to pass parameters to called procedures.
Appendix B

The following steps are executed for 8086/8088:

- The current value of the base pointer BP is pushed on the stack. This sets up the dynamic link.
- The value of base pointer BP is set to the current value of stack pointer SP.
- Space is reserved on top of the stack for the local variables of the procedure, if any exist, by reducing the current value of the stack pointer SP by the total size of the procedure variables.

This is the code generated for 80186/80286:

- The instruction ENTER size, level is executed where size is the total size of the procedure variables, and level is the lexical nesting level of the procedure. This instruction automatically sets up the dynamic link, the display, the space for the local variables on the stack, and the values for BP and SP.

The statements of the procedure body are then executed. The local variables and the parameters of the procedure are accessed with an offset relative to the base pointer BP.

Upon termination of the procedure body, the procedure epilogue is executed, performing the following operations:

The following steps are executed for 8086/8088:

- The stack pointer SP is reset to the current value of the base pointer BP. This removes the local variables from the stack.
- The dynamic link is popped to restore the old value of the base pointer BP.

This is the code generated for the 80186/80286:

- The instruction LEAVE is called. LEAVE automatically removes local variables, display, and dynamic link and resets BP and SP.
- A return instruction passes control back to the calling procedure. A far or near return is used, according to the type of call that was used to activate the procedure. The parameters and the static link are discarded automatically with the return instruction.
B.10 Function Results

A function result is returned as follows, depending on the size of the function type:

- One byte values are passed back in register AL.
- Two byte values are passed back in register AX.
- Four byte values are passed back in register DX and in register AX.
- REAL values are always passed back on top of the stack.

---NOTE---

In the current release, arrays and record types are not allowed as function types.

SET types bigger than a word are treated as structures.
B.11 Symbols in .OBJ Files

Here are the exact symbol definitions that the LOGITECH Modula-2 compiler puts in the object file. These can be used for symbolic debugging, or correcting linker symbol errors. The LOGITECH debuggers show you the names of variables or procedures you declared, without prefix and suffix. Symbols are truncated to 31 characters (the limit of the DOS linker and of some assemblers). The generated symbols are case-sensitive.

<table>
<thead>
<tr>
<th>Type of Symbol</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exported procedure</td>
<td>L_&lt;procnam&gt;_&lt;modname&gt;</td>
</tr>
<tr>
<td>Local procedure</td>
<td>S_&lt;procnam&gt;_&lt;...&gt;...</td>
</tr>
<tr>
<td>Nested procedure</td>
<td>N_&lt;procnam&gt;_&lt;...&gt;...</td>
</tr>
<tr>
<td>Global variable</td>
<td>&lt;varnam&gt;_&lt;...&gt;...</td>
</tr>
<tr>
<td>Beginning of module</td>
<td>$BM_&lt;modname&gt;</td>
</tr>
<tr>
<td>End of module</td>
<td>$EM_&lt;modname&gt;</td>
</tr>
<tr>
<td>Beginning of data of a module</td>
<td>$BD_&lt;modname&gt;</td>
</tr>
<tr>
<td>End of data of a module</td>
<td>$ED_&lt;modname&gt;</td>
</tr>
<tr>
<td>Beginning of initialization code</td>
<td>$BI_&lt;modname&gt;</td>
</tr>
<tr>
<td>if in different segment</td>
<td></td>
</tr>
<tr>
<td>End of initialization code</td>
<td>$EI_&lt;modname&gt;</td>
</tr>
<tr>
<td>Note: These symbols are neither used nor generated, but are reserved.</td>
<td></td>
</tr>
<tr>
<td>Initialization entrypoint</td>
<td>$INIT_&lt;modname&gt;</td>
</tr>
<tr>
<td>Module entrypoint after initialization</td>
<td>$BODY_&lt;modname&gt;</td>
</tr>
<tr>
<td>Code of a local module</td>
<td>$BODY_&lt;modname&gt;</td>
</tr>
<tr>
<td>Key for version checking at link time</td>
<td>KEY_&lt;dateSYMfile&gt; OF_&lt;modname&gt;</td>
</tr>
<tr>
<td>Key for version checking at run time</td>
<td>$OK_&lt;dateOBJfile&gt; OF_&lt;modname&gt;</td>
</tr>
<tr>
<td>Beginning of a Modula-2 program</td>
<td>Start Modula</td>
</tr>
<tr>
<td>Description of a Modula-2 program</td>
<td>$DD</td>
</tr>
</tbody>
</table>

KEY__ uses two underline characters with no break;
<date of file> uses the position-sensitive format ddmmyy_hhmm where;

dd = the day; mmm = month; yy = year;
"_" is a separator; hh = the hour; mm = the minute.
B.12 Aborting LOGITECH Modula-2 Programs

When you type (Ctrl-Break) or (Ctrl-C), the operating system usually aborts the program that is currently running. (Ctrl-Break) and (Ctrl-C) have the same effect in LOGITECH Modula-2. However, depending on the circumstances, there are some restrictions on their use.

In general, (Ctrl-C) only has an effect when the program is waiting for keyboard input. (Ctrl-Break) cannot be used when the program is waiting for input, but can be used any other time. (Ctrl-Break) is immediately effective — it is acted upon as soon as you use it. The effect of (Ctrl-C) is delayed until the program reads the (Ctrl-C) character. By typing (Ctrl-Break) it is possible to stop a Modula-2 program that is running in an infinite loop. However, under certain circumstances, the whole system might crash if (Ctrl-Break) was accepted. LOGITECH Modula-2 tries to prevent this from happening. Therefore, typing (Ctrl-Break) will sometimes have no effect at all.

In LOGITECH Modula-2, the Break library module lets you define how a program will behave when you press (Ctrl-Break) or (Ctrl-C) is typed. If the Break and the DebugPMD modules are linked into a program, a memory dump (file MEMORY.PMD) will be generated when you press (Ctrl-Break) or (Ctrl-C). To debug a program with the symbolic post-mortem debugger, a memory dump is needed. To be linked with a Modula-2 program, you must explicitly import the Break module into one of the program modules. Normally, you will import it in the main module of the program. If the Break or DebugPMD modules are not linked with a program, no memory dump will be generated when you use (Ctrl-Break) or (Ctrl-C) to terminate the program, but the program will stop anyway, if possible.

With the Break module, you can also keep a program from aborting by having it ignore (Ctrl-Break) and (Ctrl-C). You can also install a Break procedure which will be called when you press (Ctrl-Break) or (Ctrl-C). With a Break procedure, a dump will not be generated automatically. When the Break module is used, pressing (Ctrl-Break) once, in almost all cases, stops the program or calls the installed break procedure.
B.13 Command Line Arguments

When a *LOGITECH Modula-2* program is executed using the executable file name prefix and the `GD`, any text which follows the file name is taken as keyboard input. This means that you can type, for example:

```
M2COMP MY_PROG/BATCH/NOAQUERY
```

This works for any *LOGITECH Modula-2* program that does keyboard input using the *Terminal* or *InOut* modules. You can also include this facility in your own program. When you use a read routine like `ReadString`, it will automatically read the command line.

This lets you use *LOGITECH Modula-2* programs more easily with the *DOS* Batch files, which only recognize program input on the command line. Because the compiler, linker and debugger accept either a space or a `GD` to terminate an argument, multiple arguments may be given on the command line. For example:

```
M2L OVERLAY1 (MAINLINE)
```
Appendix C
Technical Tips
C.1 Print Time and Date

The following is an example of how to get the Time and Date from the system in a legible format. Some math manipulation has to be done to extract the information from the Time record.

```pascal
MODULE PrintTimeDate;

FROM TimeDate IMPORT
  Time, GetTime;
FROM InOut IMPORT
  WriteCard, WriteLn, WriteString;

VAR
  curtime : Time;
  tday, tmonth, tyear, ttemp,
  hr, min : CARDINAL;

BEGIN

  GetTime(curtime);
  tday := curtime.day MOD 32;
  ttemp := curtime.day DIV 32;
  tmonth := ttemp MOD 16;
  ttemp := curtime.day DIV 512;
  tyear := ttemp MOD 128;
  hr := curtime.minute DIV 60;
  min := curtime.minute MOD 60;

  WriteString('The Time is ');
  WriteCard(hr,2);
  WriteString(':');
  WriteCard(min,2);
  WriteLn;
  WriteLn;
  WriteString('The Date is ');
  WriteCard(tmonth,2);
  WriteString('/');
  WriteCard(tday,2);
  WriteString('/');
  WriteCard(tyear,2);
  WriteLn;

END PrintTimeDate.

(*
  The printout has the following format:

  The Time is 14:25
  The Date is 8/5/87
*)
```
C.2 Printing

This is one way of getting output on the printer

MODULE Printing;

FROM FileSystem IMPORT
  Lookup, Close, File, WriteChar, Response;

FROM InOut IMPORT
  WriteString, WriteLn;

VAR
  printer : File;
  str : ARRAY [0..9] OF CHAR;

PROCEDURE PrintString(str: ARRAY OF CHAR);
VAR
  i: CARDINAL;
BEGIN
  i:=0;
  WHILE (i<= HIGH(str)) AND (str[i]<>'0c') DO
    WriteChar(printer,str[i]);
    INC(i);
  END;
END PrintString;

PROCEDURE OpenPrinter;
BEGIN
  Lookup(printer,'PRN', FALSE);
  IF printer.res <> done THEN
    WriteString("cannot open 'PRN'");
    WriteLn;
    RETURN;
  END;
END OpenPrinter;

BEGIN
  str := 'It works!';
  OpenPrinter;
  PrintString(str);
  Close(printer);
END Printing.
C.3 The Screen

This is one way of writing to the screen memory. The location of the screen memory depends on your Video Adaptor Card.

For Monochrome Adaptors it is OB000H:OH
For Color Graphic Adaptors it is OB800H:OH]

MODULE Screen;

CONST
  MAXCOL = 79;
  MAXROW = 23;

TYPE ScreenType =
  ARRAY[0..MAXROW] OF
  ARRAY[0..MAXCOL] OF
  RECORD char, attr : CHAR END;

VAR
  screen[OB000H:OH]: ScreenType;
  PROCEDURE WriteToScreen(str: ARRAY OF CHAR);

VAR i : CARDINAL;

BEGIN
  i := 0;
  WHILE (i <= HIGH(str)) AND (str[i] <> 0c) DO
    screen[12, 37 + i].char := str[i]; (* Starts writing approximately in the center of the screen *)
    INC(i);
  END;
  END WriteToScreen;

BEGIN
  WriteToScreen('Hello');
END Screen.
C.4 Redirect Input

This is an example of how to use the redirection capability of Termbase module. Refer to a textbook on Modula-2 if you don’t understand the mechanism of procedure parameters as used by AssignRead.

DEFINITION MODULE RedirectInput;
 EXPORT QUALIFIED SetInput;
 PROCEDURE SetInput(str: ARRAY OF CHAR);
 END RedirectInput.

IMPLEMENTATION MODULE RedirectInput;
 FROM InOut IMPORT OpenInput;
 FROM Termbase IMPORT UnAssignRead,
 AssignRead, ReadProcedure, StatusProcedure;
 FROM ASCII IMPORT EOL;
 VAR
   InputString: ARRAY [0..80] OF CHAR;
   StringIndex : CARDINAL;
 PROCEDURE Read(VAR ch:CHAR);
 BEGIN
   IF (StringIndex<=HIGH(InputString)) AND
       (InputString[StringIndex]<>0c) THEN
     ch:=InputString[StringIndex];
     INC(StringIndex);
   ELSE
     ch:=EOL;
   END;
 END Read;

PROCEDURE Status(): BOOLEAN;
 BEGIN
   RETURN TRUE;
 END Status;

PROCEDURE SetInput(str: ARRAY OF CHAR);
 VAR i: CARDINAL; done: BOOLEAN;
BEGIN
    i:=0;
    WHILE (i<=HIGH(str)) AND (i<=HIGH(InputString)) AND (str[i]<>0c) DO
        InputString[i]:=str[i];
        INC(i);
    END;
    IF (i<=HIGH(InputString)) THEN InputString[i]:=0c END;
    StringIndex:=0;
    AssignRead(Read,Status,done);
    OpenInput(".TXT");
    UnAssignRead(done);
    END SetInput;
END RedirectInput.

MODULE Example;
(*
This program is an example of redirection and it uses MODULE
RedirectInput. Before you run this program create a file which
has the string(text). This program reads the string from a
file(example.txt) and displays it on the screen.
*)
FROM RedirectInput IMPORT SetInput;
FROM InOut IMPORT ReadString, WriteString;

VAR str: ARRAY [0..80] OF CHAR;

BEGIN
    SetInput("example.txt"); (* example.txt contains the string *)
    ReadString(str); (* Reads the string until a blank is encountered *)
    WriteString(str);
    END Example.
Appendix D
Product Support Plan

Copy Protection

The *LOGITECH Modula-2* disks are not copy-protected. This doesn't mean you can make unlimited copies of them. *LOGITECH Modula-2* software is protected by the copyright laws that pertain to computer software. It is illegal to make copies of the contents of these disks, except for your own backup, without written permission from LOGITECH, Inc. *In particular, it is illegal to give a copy to another person.*

Reminder

Remember to send your registration card, if you haven't done that. It helps us to keep our contact with you, and keeps you up-to-date with important product information.
Appendix D

Technical Support

LMIS

We know that effective communication with our customers is the key to quality service. Therefore we have set up the LMIS (LOGITECH Mouse/Modula-2 Information Service), an electronic bulletin board where you can contact us at your convenience. To logon to the LMIS, dial:

(415) 795-0408

using a 300, 1200 or 2400 baud modem.

The menu of available options is self explanatory.

BIX

LOGITECH also sponsors an electronic conference on BIX, the BYTE INFORMATION EXCHANGE system from Byte magazine. If you have access to BIX, join us in the LOGITECH conference, and communicate with us there.

Getting Help through the Hotline

You should rely on your manual or your dealer to answer questions about using your package. If you do encounter a technical problem with your package, our Technical Support Specialists will be glad to help you.

We ask you to follow these steps before you call or write.

- Read the section of the manual that describes the procedure you are trying to perform.
- If the problem relates to your software, check to make sure that the software is properly configured.

If, after following these steps, you are still not able to solve the problem, give us a call at (415) 795-0427, or write to us. If you write, please include your daytime phone number and the best time to reach you. Also, please add "Attn: Technical Support" somewhere on the envelope.

We want to help you make the most effective use of your package.
A LOGITECH Modula-2 Glossary

In LOGITECH Modula-2, these terms have specific meanings:

**Base layer**

A program which calls a subprogram. For example, the compiler passes made by the overlay version of the LOGITECH Modula-2 compiler are called sequentially by the compiled base which is their base layer.

**Compilation unit**

Part of a program contained in a separate file which can be compiled separately. Modula-2 compilation units are: definition, implementation, and program modules. Modules can be compiled separately only if the imported definition modules are already compiled. Only definition modules can export objects. If an object is exported from a compilation unit, it must be split into definition and implementation modules.

**Definition module**

The definition part of a Modula-2 module. For more information on definition modules, refer to the corresponding sections in Programming in Modula-2 by Niklaus Wirth.

**Development system**

The entire system, both hardware and software, used to develop a program. Software includes the operating system and utility programs and libraries. When used to develop Modula-2 programs, it includes Modula-2 run-time support, as well as the Modula-2 compiler, linker, debuggers, editor, utilities, and library.
Modula-2 Glossary

Language support
A program seen as an extension to the hardware. It gives the target system the ability to execute programs written in the corresponding programming language. The language support for Modula-2 is part of the Modula-2 run-time support.

Library
In general, a library is a set of functions or procedures which can be used by any program. In Modula-2 the library is equal to the set of all available Modula-2 library modules.

Library module
A Modula-2 module, consisting of a definition and an implementation part, which is available for use by any Modula-2 program.

Implementation module
The implementation part of a Modula-2 module. An implementation module contains the code that implements the capabilities provided by this module as they are specified by the corresponding definition module. The section on basic concepts in this manual contains a brief description of implementation modules. For more on the use of implementation modules refer to the corresponding sections in Programming in Modula-2 by Niklaus Wirth.

Main module
The main module of a Modula-2 program is the module that is given to the linker to link the program. The module code of the main module constitutes the main program. The main module must be a program module.

Main program
The term 'main program' has two different definitions depending on the context in which it is used:

When talking about a single program, it refers to a particular part of the code of that program, the main program code. Executing the main program code is equivalent to executing the whole program. In Modula-2, the main program code consists of the program module. The execution of a program starts with the execution of its main program code. When the execution reaches the end of the main program code, the program terminates.

When talking about programs and subprograms, the term main program refers to a program that is the base layer of a (set of) subprogram(s), and that is not a subprogram itself.
Objects
Anything that can be given a name, including constants, variables, procedures, types, and modules.

Overlay
A part of the code of a program is an overlay of that program, if this code is loaded at the same memory location as some other code - the code of another overlay - of the same program. When code that belongs to an overlay is loaded into memory, it overlays the code of the overlay that was loaded previously. Sometimes, not only code but also data is overlayed.

By using overlays, a program that would require a large amount of memory can run on a computer with less memory. Modula-2 provides a simple overlay concept in the form of subprograms.

Program
A Modula-2 program with all the modules which are imported directly or indirectly by its main module. When a program is linked, the resulting load file includes all these modules. A Modula-2 program may also call another Modula-2 program as a subprogram. A program that calls a subprogram, but is not a subprogram itself is also called a main program. In this context, the term program refers to the one main program and the set of all its subprograms.

Program module
A Modula-2 module which does not have a definition module and is not declared in any other module. The code of a program module is a main program. For more information on program modules please refer to the corresponding sections in the book Programming in Modula-2 by Niklaus Wirth.

RTS (Run-time support)
A set of modules which includes language support and other configuration-dependent functions. These include typical operating system features such as setting up the memory configuration, loading programs, and dumping memory to disk.

Separately compiled module (SCM)
A compilation unit which is either an implementation or a program module, and has been compiled separately.
Modula-2 Glossary

Subprogram
A *Modula-2* program called by another *Modula-2* program. A subprogram consists of those modules imported directly or indirectly by its main module which are not part of its base layer. A subprogram can use objects exported by the modules of the calling program. A subprogram may also call other subprograms. Subprograms in *Modula-2* provide a very simple overlay concept. For information on how to call subprograms, please refer to the definition module 'Program' in the library section of this manual. For information on the memory use of subprograms, refer to the appendix on memory organization.

Target system
The system, both hardware and software, on which you execute your application programs. In most cases the target system is the same as the development system. However, this is not a requirement. The hardware configuration of a target system for *Modula-2* or *Modula-2/VX86* does not require a terminal or disks. The software configuration may be reduced to the *Modula-2* run-time support and your program.

Workspace
The memory region allocated to a process for stack, program variables, heap, and subprograms. When a *Modula-2* program is started, it begins execution as the main process. The default stack size of any main program is 8000 bytes. This value can be modified at link time. A subprogram shares the workspace (stack) of its base layer. When a process is created, the workspace is defined as a parameter of SYSTEM NEW PROCESS. It may be located anywhere, (loop, stack, or global data). Just don't let the process have a longer lifetime than its workspace!
INDEX
INDEX

A

Abort A Program, 393
Absolute Variables, 159
Activation Record, 386
Address
  Constant, 159
  Arithmetic, 159
ADDRESS, 57 159-163, 165
ADR Function, 62, 166
Alignment, 84
ALLOCATE, 57
ANSLSYS, 10
ARRAY, 40, 172
Assembly interfacing, 110
Assembly language, 110-114, 120
AUTOEXEC.BAT, 17, 20

Compile
  From Editor, 27
  From DOS Shell, 70
Compiler,
  Configuration, 80, 81
  Directives, 90
  Error Messages, 96
  Options, 80
CONFIG.SYS, 10
Configuration of PMD, 125, 128
Controller Mask, 181
Coroutine, 63
CODE, Procedure,
  Using CODE, 89

B

Base Layer, 379, 401
BITSET, 171
BOOLEAN, 171
BUFFERS Statement, 10
BYTE, 165, 171

D

DEALLOCATE, 57
Debugging Commands
  Keyboard Control, 134
  Mouse Control, 132
DEC With Addresses, 161
Decimals, 201
Default Data Segment, 117
Definition Module, 401
Development System, 401
DISABLE, 169
DOSCALL, 170, 187
Diskettes, 9
DOS 3.1, 11
DOS 3.2, 11
DOS Non Reentrancy, 180
Dynamic Array Parameters, 48
Dynamic Link, 387

C

C Language, 115-122
Call Chain, 123, 140
Calling Convention, 387
CARDINAL, 171
CASE Statement, 43
CHAR, 171
CHR, 173
CODE, 169
Command Line Handling, 126, 394
Compilation Unit, 401
INDEX

E

Enable, 169
Enumeration Type, 171
Examples Technical Tips, 369
Execution, 381
EXTCALL Procedure, 115, 170
Extensions (See Files)

F

Files:
  Declaration, 47
  CFG, 125, 128
  FILES statement, 10
  LST, 73
  OBJ, 73
  REF, 73
  SYM, 73
Foreign Definition, 116-122
Function Results In Register, 391

G

GETREG, 168
GOTO statement, 39

H

Heap, 385

I

Implementation Module, 402
INBYTE, 169
INC, With Addresses, 161
Installation, 7
Interfacing Other Languages, 109
  Assembly Language, 110-114, 120
  C Language, 115-122
INTEGER, 171
Interrupt, 177, 181
INWORD, 169
IOTRANSFER, 167, 175, 177
ISR (Interrupt Service Routine), 178

K

Key, 107, 110
Key Control (PMD), 132

L

Language Qualifier, 116
Language Support, 402
Library Search Strategy, 217-219
Library, 402
Link:
  From Editor, 30
  With DOS Linker, 32
Linking, 101, 119, 120
LISTEN, 168, 183
LONGINT, 171
INDEX

M

Main Module, 402
Mask Interrupt Controller, 181
MathLibO, 173
Memory Dump, 123-126
Memory Organization, 377-380
Memory Requirements, 8
Module,
  Definition, 36, 51, 53, 55, 58, 77
  Implementation, 37, 52, 54, 58, 78
  Program, 36, 56, 74
Monitor, 183
Mouse Control (PMD), 132

N

NEWPROCESS, 167, 182, 382

O

Object Files (.OBJ), 392
Objects, 403
Opaque Type, 172
Options,
  Compiler, 80
  PMD, 129
OUTBYTE, 169
OUTWORD, 169
Overlays, 379, 403

P

Pascal, 35-68
PATH Command, 14, 20, 21
Pictures In Decimals Output, 204

PMD

  Commands, 137-149
  Configuration, 125, 128
  Keyboard Control, 134
  Messages, 152-156
  Mouse Control, 132
  Options, 129
  Windows, 137-149
Pointer Type, 41
POINTER, 172
Priorities, 175, 176, 181
PROC Type, 49
PROC, 172
Procedure Activation, 386-389
PROCEDURE, 172

Procedures
  IOTRANSFER, 64
  TRANSFER, 64
  PROCESS, TYPE, 64
  LISTEN, 64
  NEW, 57
  DISPOSE, 57
  WAIT, 65
  SEND, 65
  PROCESS, 165, 172, 382
Program, 403

R

REAL, 171, 208-213
Record Type, 40
Record Variant, 40
Register Constants Def, 165
Return Address, 387
RTS Run Time Support, 403
RTSIntPROC, 178
Run Time Errors, 124
Run Time Support Modules, 119, 120, 214-215
INDEX

S

Search Strategy, 217-219
Separate Compilation, 403
Separate Compilation, 403
SET, 171
set Type, 39
SETREG, 168
SETREG, 168
SIGNAL, TYPE, 65
SIZE Function, 62
Size, 166
Stack, 378, 385
Stack, 378, 385
Statement
   EXIT, 46
   FOR, 44
   IF, 43
   LOOP, 46
   REPEAT, 44
   WHILE, 45
   WITH, 41
Static Link, 387
Status, 124
Subprogram, 404
Subranges, 171
SWI, 169
Symbols In OBJ file, 392
Symbol
   Defined Twice, 101
   Name Convention, 109, 116
   SYSTEM, Module, 64

T

Target System, 404
Termination of Program, 124
Termination Procedure, 124
Transfer, 167, 175
Trunc, 173
TSIZE Function, 62, 166
TYPE, ADDRESS, 57
TYPE, WORD, 60
Type
   Conversion, 173
   Transfer, 174

V

VAL, 173
Variable Allocation, 384
Version Checking
   At Compile-Time, 105
   At Link-Time, 106
   At Load-Time, 108

W

Word, 165, 171
Workspace, 382, 404
Notes:
USER’S MANUAL

Compiler
Library
Post Mortem Debugger