LOGITECH MODULA-2

Software Development System

LOGITECH

LOGITECH

SOFTWARE ENGINEERING LIBRARY

MODULA-2/86
USER'S MANUAL

Third Edition   March 1986

LU-GU101-3

Initial issue: February 1984

Current revision: April 1986

This edition applies to Release 2.00 of the software.

Printed: April 1986

LOGITECH'S POLICIES AND SERVICES

Congratulations on the purchase of your LOGITECH MODULA-2/86 Software Development System. Please refer to the following information for details about LOGITECH's policies and services.

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

(415) 364-7057

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

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

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

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

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

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

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

To join BIX contact:

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

We look forward to hearing from you on BIX!

This manual is designed to meet your particular needs, depending upon your level of programming experience. Although the manual assumes you are familiar with basic programming concepts, it provides an overview of the Modula-2 language and a bibliography of relevant texts for more detailed information. For novice programmers this manual may be used as an introduction to the Modula-2 language and eventually as a step-by-step guide to the specific implementation of LOGITECH MODULA-2/86. For more experienced programmers, especially those familiar with Modula-2's predecessor, Pascal, it may be used as an introduction to the implementation specific features of MODULA-2/86 and later as a reference manual for specific problems or questions which may arise.

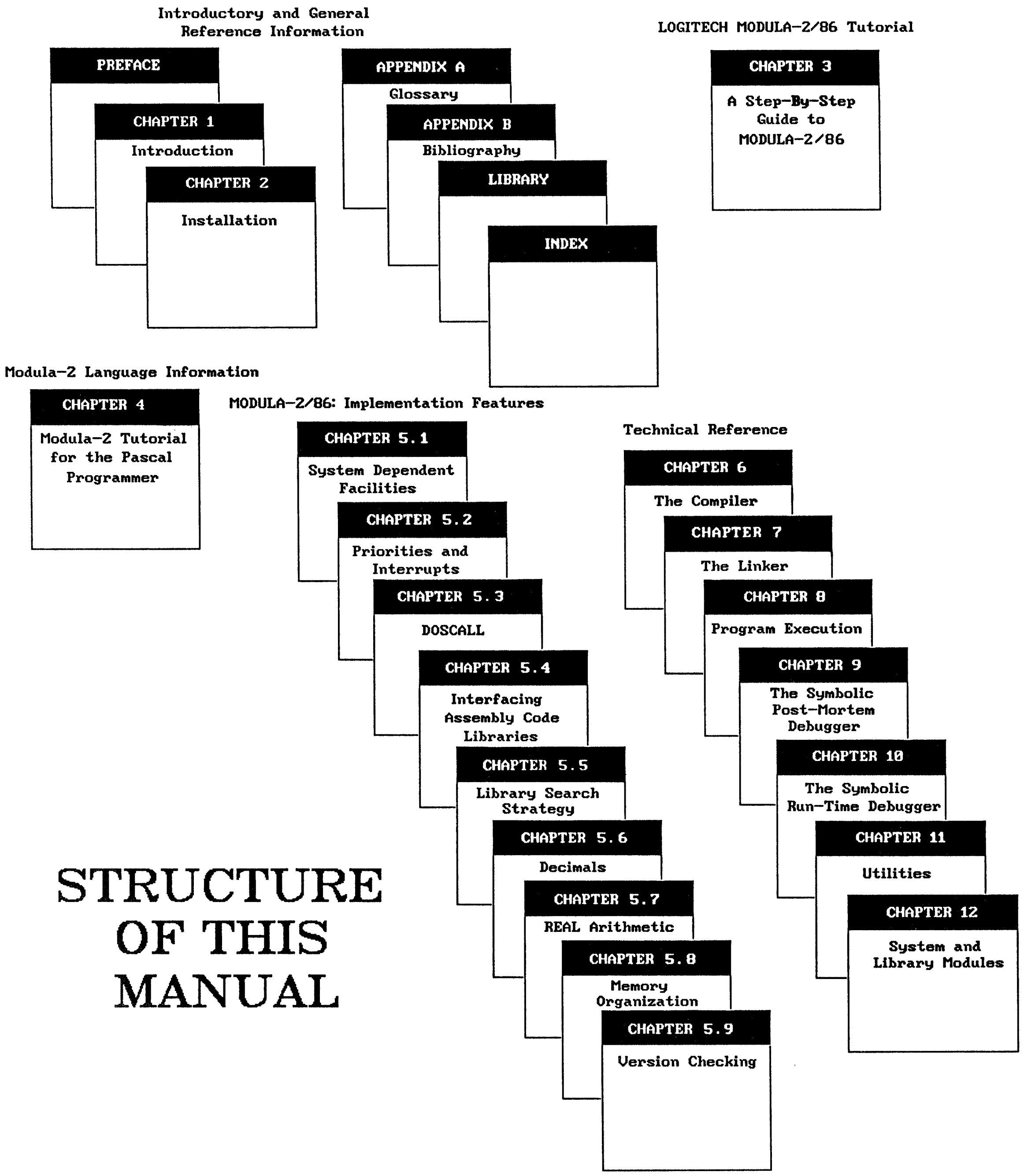
As illustrated in the following diagram, this manual contains five kinds of information:

- Introductory and general reference information, including system requirements, installation instructions, and a global index which lists all key concepts and words you may need for future reference.
- A step-by-step tutorial to guide you through the LOGITECH MODULA-2/86 world, focusing on the Mod Editor as the thread which joins, and the point from which you can control, the various components of your system.
- An overview of the Modula-2 language which explains how Modula-2 differs from, and is similar to, Pascal, in addition to describing the primary features of the language.
- A section which describes the implementation features of MODULA-2/86.
- A reference section which explains the details of the various components of LOGITECH MODULA-2/86, including the Compiler, Linker, Debuggers, Utilities, and Library.

If you are an inexperienced programmer, you can follow the manual sequentially -- working through the tutorial to familiarize yourself with the LOGITECH MODULA-2/86 system, and then reading the overview of Modula-2 to better understand the language. Experienced programmers can skim the introduction to the Modula-2 language and should concentrate on the tutorial and the system dependent facilities of MODULA-2/86, eventually using the manual as a reference guide.

NOTE:
PLEASE REFER TO THE READ.ME FILE FOR THE LATEST INFORMATION ABOUT THE SYSTEM

STRUCTURE OF THIS MANUAL



The following syntactic conventions are used in this manual:

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

  A>m2programname <CR>

- Output the program displays on the screen, for example listings, looks like this:

  ```
  Program Not Found
  ```

- Program source code looks like this:

  ```
  VAR
          a:ADDRESS;
          w:word
          off,seg:CARDINAL;
  ```

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

  <ESC>, <CR>.

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

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

- Alt characters, characters entered while the key marked 'Alt' is depressed, are preceeded by 'Alt' and enclosed in brackets, for example:

  <Alt-C>.

TABLE OF CONTENTS

Section	Contents	Page

Section **Contents** **Page**

Section	Contents	Page

1 INTRODUCTION

Welcome to MODULA-2/86!

Modula-2 is a modern language suitable for system design. It benefits from a decade of experience with Pascal by building on its strengths and correcting many of its deficiencies. The 'standard' language is powerful enough to prevent incompatible dialects from arising. Low level routines may be implemented efficiently without sacrificing the benefits of a high level modular programming language. Modula-2 allows true modular programming with strong type checking while incorporating the flexibility of routines for transfer of data between variables of different types, interrupt handling, and access to underlying hardware and operating software.

The LOGITECH MODULA-2/86 system is a full standard implementation of Modula-2 on 8088/8086 based microcomputers. Very large programs may be compiled in efficient native machine code. Features of the LOGITECH MODULA-2/86 system include:

- Native code compiler
- Extensive library of standard modules
- Support for the 8087 for fast, accurate REAL math
- Support for 80186/80286 advanced instruction set
- Support for REAL emulation
- Support for the full one Megabyte address space of the 8086/8088
- Access to underlying hardware and operating system functions
- Support for the creation of overlays for very large systems
- A syntax assisted, fully integrated editor

- Two symbolic debuggers: post-mortem and run-time
- A set of utilities
- Capability to generate ROMable code

1.1 System Requirements

To develop programs using the LOGITECH MODULA-2/86 system, the following minimum system configuration is required:

IBM PC or 'compatible' with:

- 256K or more of RAM memory (512K if you want to run the fully linked version of the compiler)
- Two double sided disk drives (300K each or more), one of them may be a hard disk drive
- PC-DOS or MS-DOS 2.x, 3.x operating system

Other systems or configurations supported:

- 8086/8088 based microcomputers running MS-DOS.

1.2 Memory Requirements

The MODULA-2 system has the following memory requirements:

- To Run the Editor
  The editor takes up less than 128KB and DOS needs approximately 32KB. Therefore, we suggest a 256KB or 320KB system.

- To Run the Editor and Call the Compiler and Linker
  If you are compiling a small program, we suggest a 398KB system. However, to be safe, a 512KB system would be best.

If you run out of memory, the editor will recover gracefully. A small low-fuel box appears when only a few KBytes remain. At this time you can save the texts that have been modified without losing them.

1.3 Other Requirements

A printer is not required but strongly recommended. Software developers will find a hard disk system useful.

Compiled MODULA-2/86 programs may be executed on any 8086 or 8088 system assuming that the target system's memory is large enough to hold the executable program and data. No references to a particular underlying operating system are generated by the compiler.

The numeric data processor 8087 is also supported. However, it is not required to run the compiler and the other program development utilities.

2 INSTALLATION

This chapter describes the installation of MODULA-2/86 under DOS. The following section provides more complete instructions on how to install your MODULA-2/86 system, and on how to make the best use of it. Some command (batch) files are provided on the MODULA-2/86 system diskette that can help you perform the necessary operations. These files assume a standard configuration - a floppy disk drive A, and either a floppy disk drive B or a hard disk drive C. You might have to modify these files or do the installation step-by-step if your system configuration differs from the assumed standard.

2.1 Configuring Your Operating System

You need to configure your operating system to run MODULA-2/86. This must be done by setting up the file CONFIG.SYS on the disk from which you start (boot) your operating system. A sample CONFIG.SYS file is provided on your MODULA-2/86 System diskette. After you have changed or installed your CONFIG.SYS file, you must restart your operating system for these changes to become effective. It is recommended that this file contain the following three commands:

- FILES=12
- BUFFERS=13
- DEVICE=ANSI.SYS

These commands have the following meaning:

- FILES=12

  Defines the number of files that can be open at the same time. A value of twelve or more is required in order to operate the MODULA-2/86 compiler, linker, editor, debugger, and utilities properly.

If the proper number of files is not set in DOS, an error message 'file not found' will appear. While the file in question may be present, it cannot be opened due to a lack of file descriptors in the operating system.

- BUFFERS=13

  It is recommended to set the number of buffers to at least thirteen. An appropriate value will increase the performance of the MODULA-2/86 system. However, this is not a requirement and you may omit this command.

- DEVICE=ANSI.SYS

  This command gives access to the 'Extended Screen and Keyboard Control' provided by DOS. Some parts of MODULA-2/86 assume that this driver is used. If this command is omitted in CONFIG.SYS, certain control characters written to the display may not have the effect specified in the definition module 'Terminal'.

  Because of the command 'DEVICE=ANSI.SYS' in CONFIG.SYS you must also put a copy of the file ANSI.SYS onto the disk that contains CONFIG.SYS. The file ANSI.SYS is provided on one of the original diskettes of your operating system.

2.2 Systems Equipped with Floppy Disks Only

If you use the MODULA-2/86 system on a floppy-based installation, we recommend the following organization of your disks:

1 First prepare a Modula-2 work disk. Insert a copy of the MODULA-2/86 system disk into drive B, and an empty, formatted diskette into drive A. Copy the contents of the Modula-2/86 system disk onto your working diskette.

2 Prepare a copy of the MODULA-2/86 compiler disk. Use this disk when compiling Modula-2 programs. Your compiler disk contains:

   - The MODULA-2/86 compiler (.LOD).
   - The symbol files (.SYM) of the library modules.

3 Prepare a copy of the MODULA-2/86 linker disk. Use this disk when linking Modula-2 programs. Your linker disk contains:

- The MODULA-2/86 linker (.LOD).
- The link files (.LNK) of the library modules.
- The reference files (.REF) of the library modules.

While you are working with MODULA-2/86, drive A holds your Modula-2 work disk with:

- The MODULA-2/86 run-time support (M2.EXE).
- The MODULA-2/86 editor (MOD.EXE).
- Your Modula-2 source files.
- Any other files you created with your MODULA-2/86 system which are needed for compiling, linking or debugging.

While you prepare your programs, drive B holds a disk with your operating system and its utilities.

- To compile, you insert a copy of the compiler disk into drive B.
- To link, you insert a copy of the linker disk into drive B.
- To run the debuggers insert a copy of the debuggers disk into drive B.
- To run the utilities insert a copy of the utility disk into drive B.

Note that, depending on the capacity of your disks, you can combine two or more of the above described disks in one disk.

If your diskettes have a large capacity, it may be worthwhile for you to study the following section on hard disk systems, on the environment variables used by MODULA-2/86, and also the section on library search strategy.

Fully Linked Version

The compiler and linker are distributed in overlay version. If you have the Base Language System/512k that contains the fully linked version of compiler and linker, you will have an additional diskette containing the files M2C.EXE and M2L.EXE.

Prepare a copy of the MODULA-2/86 fully linked compiler disk. From the additional diskette you received with the Base Language System, copy in an empty, formatted diskette the file M2C.EXE. From the MODULA-2/86 compiler disk copy all .SYM files. Your fully linked compiler disk will contain:

- The MODULA-2/86 compiler (M2C.EXE).
- The symbol files (.SYM) of the library modules.

Prepare a copy of the MODULA-2/86 fully linked linker disk. From the additional diskette you received with the Base Language System, copy in an empty, formatted diskette the file M2L.EXE. From the MODULA-2/86 linker disk copy all .LNK files. Your fully linked linker disk will contain:

- The MODULA-2/86 linker (M2L.EXE).
- The link files (.LNK) of the library modules.

PATH and Environment Variables

To allow the MODULA-2/86 system to work properly, you must insert some additional DOS commands into your AUTOEXEC.BAT file. This file must be in the root directory of your boot disk. The commands it contains are executed automatically every time you start, or boot, your operating system. If you do not yet have such a file create it in the root directory, using your text editor. If you keep your M-2 Work Disk in drive A and the compiler/linker /debuggers/utilities disk in drive B, include the following commands in your AUTOEXEC.BAT file:

- SET M2SYM=A:\;B\; ...
- SET M2LNK=A:\;B\; ...
- SET M2REF=A:\;B\; ...
- SET M2MOD=A:\;B\; ...
- SET M2MAP=A:\;B\; ...

These commands will set up the environment variables for MODULA-2/86 for a dual floppy configuration. In this way your MODULA-2/86 system can take full advantage of the DOS features. More information on the environment variables used by MODULA-2/86 can be found in the section of this manual on the library search strategy.

You must also set the DOS environment variable 'PATH' (refer to the DOS Manual). This variable is used by DOS to search for .EXE files. If you keep your M-2 Work Disk in drive A and the compiler/linker/debuggers/utilities disk in drive B, the environment variable PATH should contain:

- PATH=A:\;B:\; ...

Before you start using MODULA-2/86, be sure to re-start, re-boot, your system, so the commands of the AUTOEXEC.BAT file will be executed.

2.3 Systems Equipped with a Hard Disk

If you use the MODULA-2/86 system on an installation equipped with a hard disk, we recommend that you copy all the files on the distribution disks to your hard disk. This is the most convenient way to use Modula-2.

You can copy all the files into the same directory where you intend to write your Modula-2 programs. However, this is not very convenient. It is better to take advantage of the structured directory system (assuming that the version of DOS you are running supports this). This will reduce the number of files in your directories and, at the same time, it will allow you to use the MODULA-2/86 system from any directory you choose. We recommend the following organization:

1 Copy the file M2.EXE from the MODULA-2/86 system diskette to the directory where you usually keep public executable programs. This should be (one of) the directory(ies) where DOS searches for files to be executed. (Under DOS you can set the command search directories using the 'PATH' command.) You can also copy the file M2.EXE to the directory that you will use for developing your Modula-2 software. In this case, however, you need a copy of M2.EXE in every directory where you want to run Modula-2 programs - including your own Modula-2 programs, as well as the MODULA-2/86 compiler, linker, and debugger.

2 For the rest of the installation of your MODULA-2/86 system, command (batch) files are provided on the MODULA-2/86 system diskette. They can be used to perform the rest of the installation automatically. Only if you do not have a standard system configuration will you need do it step-by-step. The installation command files assume that your hard disk is the current drive and that the MODULA-2/86 diskettes are being inserted into drive A of your system.

Insert a copy of the MODULA-2/86 system diskette into drive A and type:

```
a:install1<CR>
```

This command file creates two new directories 'm2lod' and 'm2lib' in the root directory ('\') of the current disk (your hard disk). Then, in the directory 'm2lib', it creates the sub-directories 'def', 'mod', 'sym', 'lnk', 'ref', and 'map'.

For a step-by-step installation, perform these DOS commands:

- C>cd \
- C>mkdir m2lod
- C>mkdir m2lib
- C>cd m2lib
- C>mkdir def
- C>mkdir mod
- C>mkdir sym
- C>mkdir lnk
- C>mkdir ref
- C>mkdir map

After these directories have been created - by running the command file or step-by-step - you are ready for the next step. With the MODULA-2/86 system diskette in drive A execute the second command file. Type:

```
a:install2<CR>
```

and insert the compiler, linker, and debugger diskettes as requested by the command file.

For a step-by-step installation, copy all the files with extension 'LOD' from the three distribution diskettes to the directory '\m2lod'. Then copy all the files according to their extension into the corresponding directory. For example, copy the files with extension 'DEF' to the directory '\m2lib\def', copy the files with extension 'MOD' to the directory '\m2lib\mod', and so on.

Fully Linked Version

The compiler and linker are distributed in overlay version and to run them you type 'm2 comp' or 'm2 link'. If you have the Base Language System that contains the fully linked version of compiler and linker, you will have an additional diskette containing the files M2C.EXE and M2L.EXE.

Copy M2C.EXE and M2L.EXE from the additional diskette you received with the Base Language System to the drive and directory where you usually keep public executable programs.

PATH and Environment Variables

To allow the MODULA-2/86 system to work properly, you must insert some additional DOS commands into your AUTOEXEC.BAT file. This file must be in the root directory of your hard disk. The commands it contains are executed automatically every time you start, or boot, your operating system. If you do not yet have such a file create it in the root directory, using your text editor. Include the following commands in your AUTOEXEC.BAT file:

- SET M2SYM=C:\M2LIB\SYM; ...
- SET M2LNK=C:\M2LIB\LNK; ...
- SET M2REF=C:\M2LIB\REF; ...
- SET M2MOD=C:\M2LIB\MOD; ...
- SET M2MAP=C:\M2LIB\MAP; ...

These commands will set up the environment variables for MODULA-2/86. In this way your MODULA-2/86 system can take full advantage of the DOS features and your hard disk. More information on the environment variables used by MODULA-2/86 can be found in the section of this manual on the library search strategy.

You must also set the MS DOS environment variable 'PATH' (refer to the MS DOS Manual). This variable is used by DOS to search for .EXE files. For example, if you keep your public executable file in drive 'C:\commands', the environment variable PATH should contain:

- PATH=C:\commands; ...

Before you start using MODULA-2/86, be sure to re-start, re-boot, your system, so the commands of the AUTOEXEC.BAT file will be executed.

A STEP-BY-STEP GUIDE TO MODULA-2/86

The best way to learn a new language is to speak it. The same is true for programming languages and with the LOGITECH MODULA-2 Editor, you'll be writing Modula-2 programs in no time. The syntax assisted editor is fully integrated with your LOGITECH MODULA-2/86 system. That means you can control and call on all the tools which make up your system from within the editor.

Therefore, before we begin our typical programming session with LOGITECH MODULA-2/86, it is helpful to familiarize yourself with the features and general concepts of the Modula-2 Editor, 'mod'.

FEATURES OF MOD

- Fast on-line Modula-2 syntax check
- Ability to call the Modula-2 compiler, linker, other Modula-2 programs and utilities from the editor.
- Easy positioning and correcting of compile-time errors
- Full screen capacity
- Window based
- Multiple windows
    - variable size
    - overlapping or split windows
    - horizontal and vertical scroll
- Edits more than one file at a time
- Pop-up menu and single keystroke commands
- On-line help information
- User definable templates for Modula-2 syntactical constructs
- User definable help file
- Can be used with a mouse (LOGITECH Logimouse)

The mod editor edits standard ASCII files. It displays the text associated with these files in windows.

There is always a current position indicated on the screen by a blinking underscore, which is the locus of action for most commands.

Most commands can be invoked either from the keyboard or from a menu. Function keys, Fl through Fl0, and control characters allow you to invoke various commands. The cursor keys allow you to move up and down in a menu. To execute a command from a menu, type either <CR>, or F8, or the first letter of the command.

When the editor needs special information from you, such as the name of a file, it will put up a Prompt Box. For example, if you type F10, mod generates a Prompt Box. When entering or editing text in a Prompt Box, you can use many of the basic motion and editing commands.

When the editor needs to tell you some information, such as the result of an operation or the answer to a request, and does not require an answer from you, it will put up a Dialogue Box. For example, when you type <Alt-F1>, mod will generate a Dialogue Box indicating the release number, the amount of memory available, and on which drive your compiler and linker are located.

Text you select will be highlighted on the screen. You can either delete it or copy it into a temporary holding area called the scratchpad. You can insert contents into the scratchpad at any time, which allows text to be copied within a file or between files.

If you type a character mod does not understand, it will beep. For example, if you are trying to execute a particular command and you type something other than the first character of the command, mod will beep.

The escape command is very important in mod. The <ESC> key is used to exit a menu, exit Help, and to resume editing after a dialogue box has been displayed. The <ESC> key is also used to exit mod. If any texts have been modified and not saved when you try to exit mod by typing <ESC>, mod will ask you if you want to save the files before exiting. If you have entered a menu and do not want to select any of the choices, you can leave the menu by typing <ESC>.

Mod has an online help feature which describes all of the mod commands. At any point you can type F1 to get a help screen. Make sure the file MOD.HLP is in the current directory or in a directory specified by the MSDOS Path command. The file MOD.HLP is a standard ASCII file so you can modify it to suit your needs.

Additional information about the amount of available memory and the release number of mod you are using is displayed when you type <Alt-F1>.

That's enough general information about mod to get us started. In the following tutorial you'll learn the basics of your MODULA-2 system: how to load a program, edit it, compile, link and debug it. As you proceed through the tutorial, take note of any questions that may come up or areas in which you'd like further information. You can then refer to the appropriate section of this manual for more detailed explanations.

Before you start this tutorial make back-up copies of all the MODULA-2/86 master diskettes and then store the masters in a safe place. This tutorial assumes that you have successfully installed the compiler, the linker, and mod on your system. This User's Guide explains the proper installation procedures. Make sure that you have specified the MS-DOS 'PATH' and 'SET' commands correctly.

To evoke the editor you only need to type 'mod' and then <CR> to execute the file MOD.EXE. You can also specify a file name on the command line when mod is called and this file will be loaded into the editor. Just one more note before we begin -- unless otherwise specified, the information you must type into your computer is underlined (so you don't enter the drive name and prompt sign). Let's begin. Type:

```
        A> mod<CR>
```

MENUS

There are three important menus within mod -- the Main menu, the File menu, and the Window menu. The Main menu is evoked by typing F9. This menu provides editing functions, the ability to call the syntax checker, and access to the File and Window menus. To see the Main menu, type F9. Most of the menu choices will be described later in this tutorial. The cursor keys followed by <CR>, or the first character of a command can be used to select choices within a menu. To leave the menu without selecting anything, type <ESC>.

The File menu is called either from the Main menu or by typing <Alt-F9>. This menu provides file related commands. Type <Alt-F9> to see the File menu. Most of the commands available in the File menu can be executed using the function keys except for 'name', 'run' and 'write'. Type <ESC> to leave this menu.

The Window menu is called either from the Main menu or by typing <Alt-F10>. All the commands needed for creating and using windows are included on the Window menu. Type <Alt-F10> to see the Window menu. We will explain this menu later in the tutorial.

Now let's try mod's online help feature. Type F1 to display the help text. Type <ESC> to exit from the help text.

BASIC COMMANDS

Now you are ready to begin editing a file. To load a file into mod, type F3. A prompt box will appear on the screen asking you which file to get. For this tutorial we have provided a file called MODEX1.MOD. The default file name extension when getting a file is .MOD so you can simply type MODEX1<CR>. For files with different extensions such as .DEF or .TXT the complete file name and extension must be entered.

The text of the file EXAMP1.MOD should now be displayed on your screen. There are several ways to move around the file. Try playing with the cursor keys to position the cursor at different places in the file. The End key will move to the end of the text; the Home key will return you to the top of the file. For a more complete description of the various ways to move within a file refer to the commands listed in the section on "Keystrokes and their Definitions".

Now let's modify the Module ModEx1 which is currently displayed on your screen. If you have already changed it at all please reload it by typing F3 and then typing MODEX1<CR>.

CUT AND PASTE

Text can be selected to cut or to delete by typing F8. Selected text will be highlighted. Position the cursor at the beginning of the fifth line which begins with WriteString. The cursor should be under the "W". Now type F8 to begin selecting text. Using the cursor key, move the cursor to the right and notice how each character is highlighted showing that it is now selected text. Go all the way to the end of the line including the semicolon so that the entire line is highlighted. If you decide that you really don't want to select this particular text, the selection can be cancelled by typing F8 again -- try it.

Text which is selected can be deleted or cut and then reinserted at a different place in the file. Move back to the beginning of the fifth line and again select the entire line using the F8 key. Once the line is selected press the <DEL> key to delete the line. This line has been written to the Scratchpad. The Scratchpad holds information which has been cut from a file so that it may be copied somewhere else.

There is only one Scratchpad so whenever you cut a portion of a file, the previous contents of the Scratchpad are destroyed. The text Scratchpad can also store text for cutting and pasting between different files. Now type the <INS> key and the line will reappear in the file. Move the cursor to the end of the word BEGIN of the previous line and type <CR> to create a blank line. Position the cursor at the beginning of this new blank line and type <INS>. The WriteString statement should have been inserted on the blank line. On this new line delete the string 'The program worked! (Hit a key)' typing F8 to select the text and <DEL> to delete it. Replace the string with something simple like This is my first program!

WINDOWING

Using windows, you can edit more than one file at a time with mod. To open a new window you invoke the Window menu. You can display this menu either by typing F9 for the Main menu and then selecting Window, or by typing <Alt-F10>. The Window menu provides several choices of windows, such as horizontal and vertical windows, as well as the ability to close, move and reshape windows which display text read from files.

Each window shows a part of some text, and the name of the file associated with that text is visible on the lower right corner of the window. If a text has been changed so that it might differ from its associated file, a small 'delta' symbol appears before the name of the file.

There are two types of windows mod can create. The first type are split-screen windows which are created with either the horizontal-split or the vertical-split commands. The second type of windows are overlapping windows which are created using the open command. Windows behave differently depending upon how they were created. With practice you can develop combinations of windows which are suited for different tasks.

To begin, let's open a horizontal window. Position the cursor about half way down the screen, the window will be created from this cursor position. If the cursor is in the home or upper left corner position when a horizontal window is created, the new window will automatically split the screen in half. Now type <Alt-F10> to get the Window menu. To select horizontal split you can either type h, or select the horizontal option and type <CR>. There should be a horizontal line across the screen which is the window border. The cursor is now in the new window which is the active window. To switch back to the upper window type F7. The cursor should now be in the upper window marking it as the active window. The F7 key is used to switch between different windows.

Type F7 to activate the lower window. To close this window type <Alt-F10> to display the Window menu and then select 'close'. The window is now gone but the space which it occupied on the screen will not be reclaimed. To allow the current window access to the full screen you must reshape the window. The reshape command in the Window menu will reshape a window by defining a new lower right corner of the window. Display the Window menu <Alt-F10> and select 'reshape', the cursor should now be in the lower right corner of the window. Use the cursor keys to move the cursor somewhere down towards the bottom of the screen and in toward the left margin, to redefine the lower right corner of the window. Now type <CR> and the window will assume this new shape. If you simply want the half window to become a full screen window, select the full screen option from the Window menu <Alt-F10>.

Another type of window in mod is an overlapping window. Recall that with the horizontal-split window both windows were displayed at the same time and you could switch between the windows using the F7 key. Overlapping windows differ by actually overlapping one another on the screen. They are created using the open command from the Window menu. The cursor position at the time the open command is executed will define the upper left corner of the overlapping window.

Position the cursor at approximately the center of the screen. Type <Alt-F10> to call the Window menu and select the open command. A new window should be created using the cursor position as the upper left corner. Notice how this window overlaps the full screen window. Type F7 to switch back to the first window. The new window disappears because the full screen window completely overlaps the smaller one. Type F7 again to activate the smaller overlapping window. Close this window by selecting the close command from the Window menu. When an overlapping window is closed, the space which it occupied is now available to other windows without having to reshape them.

You can open an unlimited number of windows, however, if you open more than six windows at a time you will dramatically hinder the performance of the editor.

You should now have one window open which takes up the full screen. Type F3 to load a new file called MODEX2.MOD into this window. Mod will ask you if it is okay to abandon the current text. Answer yes to this prompt and Mod will load the file MODEX2.MOD.

SEARCH AND REPLACE

Mod allows you to search for a specified string and to replace occurrences of a specified string with a new string. To search for a string, you type F10 to specify the string you want to find. All searches wil begin from the current cursor position. The two commands to initiate a search are <Ctrl-S> for forward search and <Ctrl-R> for a reverse or backward search through a file.

Type F10 and enter the string Read. Because the cursor is at the beginning of the file start the search with <Ctrl-S>. After the first occurrence you can continue to search by typing <Ctrl-S> as many times as you need. Now position the cursor at the end of the file using the End key. Type F10 to select a new search string. Type <Ctrl END> to delete the word 'Read' and enter the word Lines. You can now do a reverse search for the word 'Lines' using <Ctrl-R>.

In addition to a simple search, you can also do a search and replace operation. The command <Ctrl-Q> initiates the search and replace feature. Like the search commands, the search and replace will start from the current cursor position and goes forward only, so if you want to search and replace through an entire file you must first position the cursor at the top of the file.

Position the cursor at the beginning of the Module ModEx2 and type <Ctrl-Q> to start the search and replace. The file MODEX2.MOD which you are currently editing has errors in it which need to be changed. Mod will prompt you for the old string. Type the word string for the Old String. Mod will then prompt for the New String. Enter the word String as the New String. Mod will now give you the choice to have a query or non-query search and replace. A query replace means that at each occurrence of the Old String, mod will stop and prompt the user before it replaces the Old String with the New String. At this point select query search. The error you will now fix will be to replace the word string with the proper spelling, String, in the identifier WriteString. Begin the search and answer yes at the replace prompts. Notice how mod searches for the Old String even when it is only part of a longer string.

SYNTAX CHECKING

Now that you have modified the file, you can quickly check for syntax errors from within mod using the F2 function key. The syntax checker will check the entire file, so the current cursor position is not important. Note: The syntax checker requires proper indentation of nested Modula-2 statements (i.e. an indent factor of at least one). The syntax checker will either position you at the first error and display the error message in a box or put up a box saying that no errors were found. In either case, the box will disappear as soon as you type <ESC> or one of the cursor keys. Try checking your syntax now by positioning the cursor at the end of the file and typing F2. There are syntax errors in the file MODEX2.MOD so the syntax checker should find them. Use the regular editing commands to fix any errors that are detected and then recheck until no errors are found. Once the program is syntactically correct you are ready to compile and link the program.

CONFIGURATION FILE

Mod is a syntax assisted editor. This means that with only one keystroke you can program mod to create commonly used structures such as IF..THEN statements and PROCEDURE declarations. There is a file called MOD.CON which is included on the mod diskette. This is the file which contains the template information mod needs to build the Modula-2 statements.

When you start mod, it will search the current directory, and those directories specified by the Path command for the standard file MOD.CON. The example MOD.CON file provided on the mod diskette is a default configuration file which we created to get you started. MOD.CON is a standard ASCII file so you can modify this file to better fit your programming style. If you edit the MOD.CON and want to reload the new version you can save it first and then, type <Alt-F2> which will find and load the modified configuration file.

To demonstrate the syntax assisted power of mod, let's open a small practice window. To do this move the cursor to the center of the screen and type <Alt-F10> to display the Window menu. Then select 'open' either by typing o or by using the cursor keys. You should now have an overlapping window in the lower right corner of the screen.

The configuration file defines structures that are created using the keys <Alt-a> through <Alt-z>. To start, type <Alt-h> for a program header. Mod will now write to the screen the outline for a program header and then position the cursor at the first entry which is 'Title'. Go ahead and fill in a title, date, and your name. Now skip a few lines and try some of the other built-in statements like <Alt-i> for an IF..THEN statement or <Alt-w> for a WHILE loop. Once you get accustomed to using this feature you can add to the configuration file and modify it so that the templates are displayed exactly how you want them.

To call the compiler and the linker from mod it is necessary to provide mod with information about where to find the compiler and linker files. This compiler/linker drive specification information is also included in the MOD.CON configuration file. You should still be in the overlapping window that you created to practice the syntax assisted features of mod. Now type F3 and load the file called MOD.CON. Mod will ask you if it should abandon any unsaved text before it allows you to load a new file -- answer yes.

The first line of this file is '@:c' which is the compiler/linker drive specification for a hard disk system. It tells mod to look for the compiler and linker files on the C drive. If you are using a dual floppy system you should change this line to '@:b'. If you modify MOD.CON remember to save the new one by typing F4. Once the new MOD.CON is saved you can load it by typing <Alt-F2>. You can now close the overlapping window you have been using by choosing the close command from the Window menu. Mod will ask you if it should abandon any unsaved text before it lets you close a window. Once the window is closed the Module ModEx2 should still be displayed on the screen in a full screen window. Now you are ready to try compiling Module ModEx2.

HOW TO COMPILE

Before you try to call the compiler or linker from mod make sure that you have properly installed both the compiler and the linker, properly set the MS DOS variables PATH and SET as explained in the Installation chapter, and that you have a drive specification in the file MOD.CON telling mod where to look for compiler/linker files. If you are using a dual floppy system, remember to put the compiler disk into drive B.

To start the compiler type F5. Mod will display a prompt box asking what file to compile, with the file MODEX2.MOD as the default. Type <CR> to begin compiling Module ModEx2.MOD. There are errors in this file so don't worry when the compiler tells you that an error is detected. The errors were placed in the file so that you can learn how to use the GoToError feature of mod.

Before the compiler terminates it will quickly display the errors it found on the screen. After the compilation is complete you will have to correct the errors in the Module ModEx2.MOD. Type <ESC> to return to the editor. To correct the errors in the module, type <Alt-F5> which will put you in GoToError mode. Mod will position you near the first error and display a box which tells you what type of error was detected. In this case the error is 'Identifier not declared' refering to the identifier LineCopied. The correct identifier is LinesCopied. Use the cursor keys to move to this error and correct it by inserting the 's' into the identifier name.

To proceed to the next error type <Alt-F5> again. The error here is the same as the last one. Go ahead and fix this one by changing 'LineCopied' to 'LinesCopied'. Try typing <Alt-F5> again and mod will tell you that there are no more errors which need to be fixed. Type F5 to compile again. This time the compilation should complete without any errors. If an error is found try fixing it the same way you did before; if no errors are found you are ready to link the program.

HOW TO LINK

If you are using a dual floppy system remember to take the compiler disk out of drive B and insert the linker disk in drive B. Now you can call the linker by typing F6. As with compilation, mod will display a dialogue box asking which file to link with the file MODEX2.LNK as the default. Type <CR> to start linking the program. The linker produces a .LOD file.

The linker will display its messages to the screen along with any linker error messages. Once the linkage is complete you can return to mod by typing <ESC>.

HOW TO RUN A MODULA-2 PROGRAM FROM MOD

You can execute MODULA-2/86 programs (.LOD files) from mod. The file you try to run must be a .LOD file. In the previous section you produced a file called MODEX2.LOD by linking the file MODEX2.LNK. To run this program, first display the Files menu <Alt-F9> and select the run command. Type the name of the file you want to run, MODEX2, and press <CR>. The default file extension is .LOD so it is not necessary to include this extension when you type the file name.

The program MODEX2 copies a specified number of lines from one file to another. The first prompt will ask you to enter an input file. You can type MODEX2.MOD for the input file. The second prompt will ask you for the lines to copy. Enter 10 at this prompt. The third prompt will ask for an output file into which MODEX2 will copy the 10 lines. Type the file name TEST.TXT for the output file. When the program terminates type <ESC> to return to mod.

MODEX2 should have created a file called TEST.TXT. To see this file, open a new window by typing <Alt-F10> to display the Windows menu and select horizontal split. Now get the file TEST.TXT by typing F3 and then the file name. The first 10 lines of the file MODEX2.MOD were copied to the file TEST.TXT.

EXITING MOD

To exit mod simply type <ESC>. If any files have been modified and not saved, mod will give you a chance to save the modifications before it terminates.

USING MODULA-2/86 WITHOUT THE EDITOR

All the components of MODULA-2/86 function as stand alone parts. Therefore, once you have written your MODULA-2 program, as long as you use your editor to create standard ASCII files, you can evoke the Compiler, Linker and all the Utilities from the command line.

To run a MODULA-2 program you can still use the runtime system and run your program the same way you were running it from within MOD. Or, if you use the utility LOD2EXE, you can create a standar .EXE file and run your program without the runtime support.

Once you have completed the sample program, exit from MOD and try to compile, link and run your program!

MODULA-2/86 is easy, flexible and extremely powerful.

HOW TO ANALYZE AND SOLVE PROGRAM EXECUTION ERRORS

There are three possible ways to analyze the execution error of a program. The simplest method is the use of module 'Debug' which provides you with limited debug information upon unsuccessful program termination.

The next 'level' of debugging is the use of the symbolic post-mortem debugger which allows you to analyze the program status (content of variables) from a memory dump file produced upon unsuccessful termination of the program.

The best tool to study the execution of a program is the symbolic run-time debugger which allows you to follow the program execution in steps, analyze at certain breakpoints the variable contents, and slowly approach the statement producing the execution error.

For further description of both symbolic debuggers, please refer to the corresponding chapters. The following text explains in more detail the use of module 'Debug':

If the module 'Debug' is linked to the application program, no dump file is generated. Instead, the system generates a list of the call sequence within the current process that lead to the termination displayed on the terminal. If the necessary reference files are available in the current directory, each line displays the name of the procedure, its module and the source line number, where the next procedure is called, or (in the case of the first line listed) where the program was stopped.

If there are missing reference files, the procedure names are replaced by procedure numbers within the module, and the line numbers are replaced by the value of the instruction pointer. The module body (module initialization code) is referred to as procedure zero. It is followed, in the order of their declaration, by the procedures declared in the definition module, if any, and then by the other procedures of the module. The Utility M2DECOD can be helpful to associate procedures numbers with your code.

Sample Terminal Output

```
---> halt called
MODULE : Demo  PROCEDURE : LastOne      LINE : 48
MODULE : Demo  PROCEDURE : RecursiveOne LINE : 37
MODULE : Demo  PROCEDURE : RecursiveOne LINE : 38
MODULE : Demo  PROCEDURE : FirstOne     LINE : 24
MODULE : Demo  PROCEDURE : Demo         LINE : 57
---> halt called
```

To be able also to debug all initialization codes of imported modules, module 'Debug' should be forced to be initialized as the first module. This can be accomplished by importing it as the first module in the main module of the application program.

KEYSTROKES AND THEIR DEFINITIONS

The following section lists all the MODULA-2 Editor keystrokes and their definitions.

KEYSTROKE	DEFINITION
Esc	Escapes from prompt boxes and dialogue boxes and from the editor itself. If you try to escape from the editor with unsaved modifications, you will be asked if you really want to do this.
Enter, <CR>	Inserts a line-break, moves down one line, and indents. Functions the same as <Ctrl-O>, <Ctrl-N>, and <Tab>.
Home	Moves to the beginning of a text. (Also valid in menus)
Ctrl-Home	Moves the cursor to the top left corner of the window.
End	Moves to the end of a text. (Also valid in menus)
PgUp	Slides the window upward over the text, approximately the height of the window.
PgDn	Slides the window down over the text, approximately the height of the window.
Del	If a block of text is selected, deletes it and places it in the scratchpad. Otherwise deletes the character at the cursor position. If this is done beyond the end of a line, it will join the following line to the current one.
Ins	Inserts the contents of the scratch pad at the current cursor position.
Tab	(Note: Tab, in mod, does not function as it does in most standard editors). Moves the cursor to an appropriate beginning column for the current line. If the current line has something on it, Tab moves the cursor to the first non-blank. Otherwise, it looks backward for keywords or preceding text and positions the cursor to an appropriate indentation according to Modula-2 syntax. If a block of text is selected with F8, Tab will indent (move to the right) all the lines by the indent factor.

KEYSTROKE	DEFINITION
Alt-Tab	If a block of text is selected with F8, Alt-Tab will 'unindent' (move to the left) all the lines by the indent factor.
Shift Tab	Moves to the end of the current line.
Ctrl-->	Go to the next word.
Ctrl<--	Go to the previous word.
Ctrl-PgUp	Scrolls the text up one line in the window.
Ctrl-PgDn	Scrolls the text down one line in the window.
Ctrl-End	Deletes the entire line including the line-break and loads it into the scratchpad, replacing the old contents.
Ctrl-A	Moves to the beginning of the current line.
Ctrl-B, <--	Moves the cursor Backward one character.
Ctrl-D	See Del.
Ctrl-E	Moves the cursor to the end of the current line.
Ctrl-F, -->	Moves the cursor Forward one character.
Ctrl-K	Kills the rest of the line, to the right and loads it in the scratchpad, replacing the old contents.
Ctrl-L	Redisplays the screen.
Ctrl-N, ↓	Moves the cursor to the Next line.
Ctrl-O	Opens a line: Breaks the line, or inserts a new line.
Ctrl-P, ↑	Moves the cursor to the Preceding line.
Ctrl-Q	Query replace: starting from the current position Replaces old string with new string. If you answer yes to the Query mode, the replace will stop after each old string found and ask for a confirmation; otherwise, it will replace all occurrences.
Ctrl-R	Searches in the Reverse direction for the current pattern.
Ctrl-S	Searches in the Forward direction for the current pattern.

KEYSTROKE	DEFINITION
F1	Help - Displays a very brief help text in a temporary window. (The help text is in the file MOD.HLP, so you can edit it.) Type <ESC> to go back to your text. All cursor movement keys are allowed to browse through the help text.
Alt-F1	Gives system information about mod. Type <ESC> to go back to your text.
F2	Syntax - Checks the syntax of the text as a Modula-2 module. It will either position you to the first error and display the error message in a box, or put up a box indicating that no errors were found. In either case, the box will go away as soon as you type something. The syntax checker requires that you properly indent your Modula-2 program.
Alt-F2	Reloads the Configuration File.
F3	Get - Prompts for a file name and loads the file in the current window. If the file you edit is a .MOD file, you don't need to specify the extension 'MOD' because it's used by default.
Alt-F3	Splits the current window and asks for a file to load.
Ctrl-F3	Opens a full window and asks for a file to load.
F4	Save - Saves the current text in its associated file.
Alt-F4	Go to the specified line.
Ctrl-F4	Tells you at what line and column the cursor is.
F5	Compile - Invokes the MODULA-2 Compiler. Note: mod can only run a Compiler 2.0 or later. If the file has been modified, it will ask if you want you want to save it.
Alt-F5	Go to next error after compilation with errors. It loops on the error list (maximum 30 errors). Allows you to work on an existing error listing file (.LST) even if you didn't evoke the compiler from the editor. This is very useful to correct all the compiler errors after a batch compilation. Note: This keystroke is supported only with Compiler 2.0 or later.

KEYSTROKE	DEFINITION
F6	Link - Invokes the MODULA-2 Linker. Note: mod can only run a Linker 2.0 or later.
Alt-F6	Converts a number (eg: 123, -123, 0b00h, 867B, 45H) or a string (eg: 'ab', "Ab", 'a', "a") into a cardinal, integer, octal, hex, binary, or string.
Ctrl-F6	Displays the ASCII table (file mod.asc)
F7	Next Window - Switches to another window. If repeated, this command will cycle through all windows.
Ctrl-F7	Enters any byte value [00H..FFH] at the current cursor position. This command is useful to enter a page break (=0CH) or special characters which don't exist on the keyboard.
F8	Select - Starts selecting a block of text. All direct motion commands (Home, End, Arrows, Page Up, and Page Down) can be used to establish the other end of the selected region. The selected text is highlighted. (F8 acts like a switch -- if you make a selection and type F8 again, the second F8 cancels the selection.)
F9	Main Menu - Calls up the Main Menu with the following commands: ▪ files Calls up the File Menu. Functions the same as <Alt F9>. ▪ delete Cuts a selected block of text from your program into the scratchpad; removes the text from the program. Using F8, first select a block of text. 'Delete' this text and put it in the scratchpad. ▪ copy Copies a selected block of text from your program into the scratchpad; leaves the text in the program. Using F8, first select a block of text. 'Copy' into a scratchpad the text you've selected. ▪ paste Pastes the text from the scratchpad into the window at the current cursor position.

KEYSTROKE	DEFINITION
	▪ quit Exits the editor. Functions the same as <Esc>. If you have modified your program, the editor will ask if you really want to exit without saving your modifications. Y = yes, N or <Esc> = no.
	▪ syntax Checks to see if your program is syntactically correct. Functions the same as F2.
	▪ windows Calls up the Window Menu. Functions the same as Alt-F10.
Alt-F9	File Menu - Calls up the File Menu with the following commands:
	▪ compile Calls the MODULA-2 Compiler and prompts for the program you wish to have compiled. Note: mod can only run a Compiler 2.0 or later. When finished compiling, type <ESC> to return to the editor. If errors are detected by the compiler, the errors are displayed on the screen and a listing file with the error message is generated with the name, <prog>.LST. You can open a window in this file and check the errors, or better, use <Alt-F5> to have an automatic positioning where the error occurred. Note: this feature requires a system with at least 398K bytes of main memory. Only the overlay version of the Compiler can be called from mod. You cannot call 'M2C.EXE', the fully linked version of the Compiler.
	▪ get Prompts for the file name and reads in the file. Functions the same as F3.
	▪ link Calls the MODULA-2 Linker and prompts for the program you wish to have linked. Note: mod can only run a Linker 2.0 or later. When finished linking, type <ESC> to return to the editor. Only the overlay version of the Linker can be called from mod. You cannot call 'M2L.EXE', the fully linked version of the linker.

KEYSTROKE	DEFINITION
	▪ name Allows you to change the name of a file. Especially useful if you want to retain a copy of an old file as well as the modified version of this file. You would rename the modified file and thus keep both files (.MOD is the default extension).
	▪ run Executes any MODULA-2/86 program that fits in memory (extension .LOD). Note: mod can only execute a program generated with a Compiler and Linker 2.0 or later. It prompts you for the name of the program you wish to execute. When the program has finished, type <ESC> to return to the editor.
	▪ save Saves the current text in its associated file. Functions the same as F4.
	▪ write Also saves text, but you specify the file name you wish to save. The default proposes the current file name. (.MOD is the default extension).
F10	Pattern - Opens a window for editing the search pattern for editing. Note: '<ESC>' inside the pattern window closes the pattern window and restores the old pattern. 'Enter' sets the new pattern and closes the window. Any other command saves and closes the pattern, and is then executed, including Search forward and Search backwards.
Alt-F10	Window Menu - Calls up the Window Menu with the following commands:
	▪ close Closes the selected window. If more than one window is open, the one in which the cursor is positioned disappears. The window that remains open will retain its current shape, for example, half the screen if two windows were opened. Use the reshape command to modify it.

KEYSTROKE	DEFINITION
	▪ hor. split Splits the current window horizontally. First, define where you want to split the window by positioning the cursor. Then select 'hor. split' from the menu to split the current window. If the cursor is in the upper left corner position when the hor. split is invoked, the current window will automatically be split in half. You then invoke F3 to get another file. This allows you to have two or more windows open at the same time.
	▪ move Moves the current window. Select 'move' from the menu. Moving the cursor defines the new position of the upper left corner of the window. Enter <CR> to confirm this position and the window will move to this position.
	▪ open Creates a new window with the upper left corner at the current cursor position. The size of the screen determines the size of the new window.
	▪ reshape Reshapes the current window. Select 'reshape' from the menu. Moving the cursor defines the new position of the lower right corner of the window. Enter <CR> to confirm this position and the window will reshape with the new lower right corner.
	▪ vert. split Splits the current window vertically. First, define where you want to split the window by positioning the cursor. Then select 'vert. split' from the menu to split a current window. If the cursor is in the upper left corner position when the vert. split is invoked, the current window will automatically be split in half. You then invoke F3 to get another file. This allows you to have two or more windows open at the same time.
	▪ full screen Makes the current window a full screen window which overlaps the other windows. All other windows remain unchanged and can be visited using F7. It is not the reverse operation of vertical or horizontal split.
<Alt-a>..<Alt-z>	Inserts a template in the current window of the current cursor position (please refer to the Configuration File section).

CONFIGURATION FILE

The file 'mod.con' is an ASCII file that contains system information and user definable templates for the MODULA-2 Editor.

Select the Compiler/Linker Drive

The MODULA-2 Editor is compatible with MODULA-2/86 Release 2.0 or later. If you try to invoke the MODULA-2 Compiler or Linker or run any MODULA-2/86 .LOD file, and you are using a version earlier than 2.0, you will get the following error:

```
Bad File Structure
```

Type <ESC> to return to the editor.

If mod is in a different drive than the MODULA-2/86 Compiler and Linker, before you run the Compiler or Linker from within mod, you must specify in the configuration file the drive in which to search.

To tell mod that the Compiler and Linker are in drive C you type the following:

```
@:C
```

If you use a different drive-id, substitute your drive-id for 'C'. If you fail to indicate the drive in which you keep the compiler or linker, when you try to invoke the compiler or linker you will get the following error message:

```
Program Not Found
```

The same message is displayed when you try to run a MODULA-2/86 program (.LOD) and you do not specify the correct drive and directory to tell where the .LOD is.

Select Indentation

The indent factor (the number of columns used to indent) is user programmable. The default is 2, so if you wish to change the indent factor, simply type any number 0 to 9 after @:C. (The value 0 disables the indentation mechanism.)

Select Backup Option

You can also specify if you want to keep the backup file by typing b in the configuration file (mod.con). For example, to specify drive C, to define an indent factor of 4, and to indicate true for the backupFlag type:

<u>@:C4b</u>

If you accidently try to invoke the compiler or linker, for example with F5 or F6, you can exit by typing <u><ESC></u>. If, however, you begin compiling or linking, you must type <u><Ctrl-Break</u>> to stop them.

User Definable Template

A template is a sequence of characters inserted in the currently open window when you press the ALT key and a character [a..z] associated with the template.

The current version of the Configuration File contains templates for some frequently used Modula-2 constructs; for example, if you type <u><Alt-i></u> you will get the following template:

IF THEN

END

You can redefine the templates as you wish. The format of a template is:

```
...
@x
1st line of the template

Nth line of the template
...
```

When you type <u><Alt-x></u>, the lines following '@x' will be inserted in the current cursor position. The first underscore character, '_', found in the template lines will be the new cursor position after the template is inserted. If there is no underscore the first character of the first line of the template will be the new current cursor position. User definable templates are always called by typing <Alt-x> where x is in the set [a..z].

You can modify the Configuration File with the MODULA-2 Editor itself. After you have saved the Configuration File you can reload it by typing Alt-F2.

The maximum size for the Configuration File is 3000 bytes.

USING LOGITECH MODULA-2 EDITOR WITH A MOUSE

The LOGITECH MODULA-2 Editor can be used with a LOGITECH Logimouse. There is a file in CLICK -- the control center of the mouse -- called mod.mnu which enables you to use pop-up menus in mod. Please refer to your LOGIMOUSE documentation for more information about using mod with LOGIMOUSE.

MODULA-2 TUTORIAL FOR THE PASCAL PROGRAMMER

This tutorial is aimed mainly at Pascal programmers in transition to Modula-2. The fact that Modula-2 evolved from Pascal makes it easy for Pascal programmers to familiarize themselves with the new language.

There are two levels of differences 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.

Types of Modules

There are three types of modules in Modula-2. These are program modules, definition modules and implementation modules. Program modules contain the source code for a user's main program. Program libraries are created from matched pairs of definition modules and implementation modules. The source code for all types of modules is stored as standard text files and may be modified by any text editor capable of working with these files. The naming convention for program and implementation modules is .MOD. Definition modules have the file extension .DEF.

Note: Most of the common word processors have program file modes which work with standard text files.

Program Modules

A program module is the main module of a user 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:

```
MODULE <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.

Definition Modules

Definition modules are used to define the interfaces 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 <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>;
        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>.
```

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 is a language aimed at enhancing software production. The real advantage of using the language is that it shortens development time. While the real time speed of a Pascal or Modula-2 compilers are important, we must look at the overall time involved in software production cycle. This includes the time for software updates and alterations. With Pascal this often translates into a significant additional programming effort. This extra effort ends up offsetting the benefit of short Pascal compiling sessions. With Modula-2 such wasteful "domino effect" is minimized by using highly independent module libraries.

The core of the Modula-2 language is smaller than that of Pascal. This is due to the fact that Modula-2 has no predefined I/O statements, math functions or string manipulation routines, to name a few. Instead, they are all imported from a variety of library modules. In a sense, Modula-2 really practices what it preaches!

There are two main sections to this tutorial. The first section will allow a Pascal programmer to write a Modula-2 program or to convert a Pascal program to Modula-2. The second section explains the concept of the module.

The following section will discuss the 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.

The 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:

```
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 are new concepts specific to Modula-2 and demonstrated 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, published by Springer Verlag, you will find that it exports many procedures. In our example we chose to "import" the four required procedures only.

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, by appending an 'H'. Similarly, 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 the following basic data types: integer, boolean, characters, real, cardinal and bitsets. The first three types are used identically as in Pascal. Using reals has the 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.

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.

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 realted pointer type ChemicalPointer, we proceed to define the following pointer-typed variables:

```
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,Element,Solid) ;
(* Water is a compound and a liquid at normal conditions *)
NEW(Water,Compound,Liquid);
(* Oxygen is an element and a gas at normal conditions *)
NEW(Oxygen,Element,Gas);
```

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

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

```
VAR Screen[B800H: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 that have 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 are shown in the following function to 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.

```
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 := 9.5;
    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 is an example of using the CASE statement to translate numeric school grades into letters:

```
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*X 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) * 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:

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

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

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

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.

```
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 ");
    WriteReal(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.

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. Modula-2 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.

```
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>)].

The following is an example for a routine to calculate the mean value of an array of reals.

```
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). The example 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:

```
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.

Modula-2 provides a number of predefined functions and procedures. They are listed in Wirth's book, Programming in Modula-2.

USE OF MODULES

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.

The implementation module is:

```
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. Imported objects are placed in either the definition or implementation module. If there are imported data objects that are included in the definition module, the import list is located in the latter 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.

Let us return to our 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. Using an opaque complex type, the above transition is a smooth one.

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

```
DEFINITION MODULE ComplexOps;

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.
```

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 a mandatory requirement in Modula-2.

```
IMPLEMENTATION MODULE ComplexOps;

(* 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.
```

Below is the implementation module version that uses polar coordinates:

```
IMPLEMENTATION MODULE ComplexOps;

FROM MathLib0 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 MathLib0.
- 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.

It is worthwhile pointing out that procedure MakeComplex in the very first implementation seemed an extravagant export. After all, given the definition of Complex, client programs can assign values to the record fields effortlessly. The situation is quite the reverse 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.

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.

```
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.WriteLn;
   REPEAT
      InOut.WriteString("Enter number of lines ");
      InOut.ReadCard(NumLines); InOut.WriteLn;
   UNTIL NumLines > 0;

   MyFileIO.Assign(F,Filename);
   MyFileIO.Reset(F);
   InOut.WriteString("Enter text "); InOut.WriteLn;
   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.

Standard Library Modules

Modula-2/86 comes with a variety of versatile library modules. They supply users' programs with a wide gamut of capabilities. This includes string manipulation, file I/O, disk directory access, data conversions, mathematical functions, DOS and low level access, coroutines, just to name a few. Many of the above routines are part of Pascal, but not Modula-2. Thus Modula-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. The 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] <> 0C THEN
      i := 0;
      WHILE (i <= HIGH(A)) AND (A[i] <> 0C) DO
         Ptr :=S^.Element + i * TSIZE(CHAR);
         Ptr^ := A[i];
         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.
```

The next module we examine is SYSTEM. It exports three data types: 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.

```
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.

Let us present a simple procedure to perform a generic linear list search on an array:

```
PROCEDURE GenericLookUp(VAR SearchArray : ARRAY OF WORD;
                        SampleScalarType,
                        SearchValue    : ARRAY OF WORD;
                        IsItEqual : SuppliedPROC;
                        VAR Found : BOOLEAN) : CARDINAL;

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;
   RETURN SearchIndex;
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.

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 Ptr1, Ptr2 : POINTER TO RECORD
            DayNumber, MonthNumber : CARDINAL;
                       END;
BEGIN
    (* Get pointers addresses *)
    Ptr1 := ADR(Element1);
    Ptr2 := ADR(Element2);
    RETURN ((Ptr1^.DayNumber = Ptr2^.DayNumber) AND
            (Ptr1^.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, Coroutine1, Coroutine2, Coroutine3 : PROCESS;
  WorkSpace1, WorkSpace2,
  WorkSpace3 : ARRAY [1..200] OF WORD; (* Workspace *)

PROCEDURE Message1;
BEGIN
  LOOP
    WriteString("In Coroutine # 1");
    WriteLn;
    TRANSFER(Coroutine1, 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, Coroutine1);
  END;
END Message3;
```

```
BEGIN (* main *)
   (* Create the new Coroutines *)
   NEWPROCESS(Message1, ADR(WorkSpace1),
         SIZE(WorkSpace1), Coroutine1);
   NEWPROCESS(Message2, ADR(WorkSpace2),
         SIZE(WorkSpace2), Coroutine2);
   NEWPROCESS(Message3, ADR(WorkSpace3),
         SIZE(WorkSpace3), Coroutine3);
   TRANSFER(main, Coroutine1);

END ConcurrentDemo.
```

The above program shows that each coroutine 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 TRANFER. 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 module Processes 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. To demonstrate how the above data type and procedures are used, consider the following program. It 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 us present the listing and then resume our discussion.

```
MODULE Synchronicity;

FROM Processes IMPORT SIGNAL, Init, SEND, WAIT,
               StartProcess;
FROM InOut IMPORT WriteString, WriteLn, ReadCard,
             WriteCard;
FROM RealInOut IMPORT ReadReal, WriteReal;
FROM MathLib0 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];
      SEND(GoAheadMakeMyDay);
   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;
   StartProcess(GetSumSquare,400);
   SEND(GoAheadMakeMyDay);
   LOOP
      WAIT(GoAheadMakeMyDay);
      WriteString('Summing 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;
   Average := SumX / FLOAT(NumData);
   StdDeviation := sqrt((SumXX - SumX * SumX /
                  FLOAT(NumData)) /
                  (FLOAT(NumData - 1));
   WriteString('Average = ');
   WriteReal(Average,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.

MODULA-2/86: IMPLEMENTATION FEATURES

5.1 SYSTEM DEPENDENT FACILITIES

This chapter gives an overview of the MODULA-2/86 specific low-level features. The chapter Priorities and Interrupts gives additional information on hardware dependencies.

The differences in programming for various implementations can be attributed to the following:

- Changes to the language proper.
- 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 -- the set of standard library modules that allow access, for example, to the file system, the keyboard and the screen.

A description of the MODULA-2/86 library is given in the library section of this manual.

WARNING

All features described in this chapter should be applied with utmost care, since their use might conflict with the basic software of the operating system, and the MODULA-2/86 system.

5.1.1 Language Extensions

Constants of type ADDRESS may be declared as <segment:offset>, where the segment and offset are CARDINAL numbers. The segment and offset may not be constant expressions.

Examples:

```
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 *)
```

MODULA-2/86 also provides for the declaration of absolute variables. Absolute variables are variables for which the programmer, rather than the MODULA-2/86 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.

5.1.2 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.

5.1.2.1 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:

- TYPE ADDRESS = POINTER TO WORD;
- TYPE ADDRESS =RECORD

```
                    OFFSET: CARDINAL;
                    SEGMENT: CARDINAL;
                END;
```

Example:

If we assume the declarations:

```
VAR
    a: ADDRESS;
    w: WORD;
    off,seg: CARDINAL;
```

then the following statements are legal:

```
a^ := w; w := a^;
a.OFFSET := off; off := a.OFFSET;
a.SEGMENT := seg; seg := a.SEGMENT
WITH a DO SEGMENT := seg END;
```

5.1.2.2 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 'address overflow' will occur upon an overflow of the OFFSET value on an addition or subtraction operation.

The standard procedures INC and DEC can also be used with variables of type ADDRESS. The following declarations are assumed:

- PROCEDURE INC (VAR a:ADDRESS; k:CARDINAL);
- PROCEDURE DEC (VAR a:ADDRESS; k:CARDINAL);

Note that in particular, the second parameter must be assignment compatible with type CARDINAL. If test code is on, negative values will generate a run-time error. Using INC and 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.

Operation	1st Operand	2nd Operand	Result Type
addition	ADDRESS	CARDINAL	ADDRESS
addition	CARDINAL	ADDRESS	ADDRESS
subtraction	ADDRESS	CARDINAL	ADDRESS

MULTIPLICATION AND DIVISION

Multiplication and division operations (*, /, MOD, DIV) are not allowed with operands of type ADDRESS.

COMPARISON

The comparison of two operands, or expressions, of type ADDRESS is allowed. The following explains the restrictions and how the comparison is implemented:

- Operations 'equal', 'not-equal':

  A check on (non-) identity is generated: a1 = a2 <--> (a1.SEGMENT = a2.SEGMENT) AND (a1.OFFSET = a2.OFFSET)

- Operations 'greater-than', 'greater-equal', 'less-than', 'less-equal':

  These operations 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.

5.1.2.3 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 0FFFFH 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 0FFFFH: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.

5.1.3 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 MODULA-2/86 run-time 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 Chapter 12 of the Modula-2 Language Report in Programming in Modula-2.

5.1.3.1 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.

- RTSVECTOR

  This is the number of the interrupt vector which is used to call the MODULA-2/86 run-time support. It is not recommended that application programs call the run-time support directly. If it is necessary, RTSVECTOR can be used to issue a software interrupt.

5.1.3.2 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. The section on 'Address Arithmetic' explains more on the properties and the use of type ADDRESS.

5.1.3.3 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
- TSIZE(type, tag1const, tag2const,...): 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.

5.1.3.4 Procedures Exported from Module SYSTEM

- NEWPROCESS (processBody: PROC;

  ```
  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;

  ```
  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.

- 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 MODULA-2/86 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 MODULA-2/86 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.

- 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.

5.1.4 Data Representation

The data types have the following internal representation in MODULA-2/86:

- 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.

- 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
- BITSET

  Two bytes. If we number the elements of a set from 0 to 15, the representation in a memory word is:

```
 --------------------------------------------
| 7 6 5 4 3 2 1 0 | 15 14 13 12 11 10 9 8 |
 --------------------------------------------
 low byte                high byte
```

- POINTER
- PROC
- PROCEDURE
- ADDRESS
- PROCESS

  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).

- ARRAY

  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).

- RECORD

  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.

- Opaque Types

  Opaque Types are always allocated four bytes, regardless of their actual implementation.

5.1.5 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 provided as built-in, standard procedures or as part of the MODULA-2/86 library. They are as follows:

Standard Functions	Library Functions
CHR	MathLib0.real
ORD	MathLib0.entier
VAL	
FLOAT	
TRUNC	

  Type conversion works by calculating a new value of a new type which corresponds to the value to be converted. Code is 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 coersion. This method is system dependent - it depends on the internal representation of data. Therefore, type transfers must be used with utmost care, and should be avoided 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 provides for use of an identifer 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:

SYSTEM.TSIZE (T1) = SYSTEM.TSIZE (T2)

When using type transfer instead of type conversion the user must be aware of the internal representation of data. Also, his programs probably will not run on other machines or with other implementations of Modula-2.

Example 1:

Interpret the value of a SET as a CARDINAL

```
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

```
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:

```
       i :=  2*i  +  VAL  (INTEGER,  c);
```

5.2 PRIORITIES AND INTERRUPTS

5.2.1 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.

5.2.2 Priority Levels

Both the MODULA-2/86 compiler and the MODULA-2/86 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).

The above range of priority levels is the default for the MODULA-2/86 system. The default may be changed during installation of MODULA-2/86 as follows:

Note: To encorporate the following changes, you need the sources of the MODULA-2/86 run-time support and some compiler modules.

- Limit in the compiler
  The compiler parameter module contains a variable that specifies the upper limit of the legal range. By assigning another value to that variable, this upper limit can be modified.

- Limit in the run-time support
  Please refer to the assembly source program of the run-time support for the changes required to support another range of priority levels.

5.2.3 Interrupt Handling

There are three main ways to handle interrupts in MODULA-2/86:

5.2.3.1 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 MODULA-2/86, 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 MODULA-2/86.

5.2.3.2 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 permanently. 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.

5.2.3.3 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 upon interrupts, but uses the current stack to handle the interrupt. This provides a great improvement in speed because the overhead of two coroutine transfers is removed. One other 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 following code shows a module which allows the installation of a Modula-2 procedure as an interrupt service routine:

```
DEFINITION MODULE ISR;
(*
     Interrupt Service Routines

     Module, to support faster interrupt handling without
     context-switch upon interrupts.
     Allows installation of Modula-2 procedures as interrupt
     service routines.
*)

FROM SYSTEM IMPORT
   ADDRESS, BYTE;

EXPORT QUALIFIED
   ISR, InstallISR, UninstallISR;

(*$A-*)
(*
     make sure, that the following type declaration will
     not have word aligned fields, because they have to
     stay consecutive
*)
   TYPE
     ISR = RECORD (* should be considered a hidden type *)
                  (* the following fields represent the code
                       that will be executed as the actual
                       interrupt service routine. The address
                       of a variable of this data type will
                       be used as the content of the specified
                       interrupt vector
                  *)
                  (* save all registers; BP automatically saved
                       by user procedure
                  *)
                  pushAX, pushCX, pushDX, pushBX,
                  pushSI, pushDI, pushDS, pushES: BYTE;
```

```
            (* call indirectly the user procedure *)
            farCall: BYTE;
            handlerProc: PROC;

            (* send EOI code to interrupt controller 8259 *)
            sendEOI: ARRAY [1..5] OF BYTE;

            (* restore the saved registers *)
            popES, popDS, popDI, popSI,
            popBX, popDX, popCX, popAX: BYTE;

            (* return from interrupt *)
            iret: BYTE;

            (* the next fields hold some additional
            information, needed to set and restore
            the interrupt vector
            *)
            oldInterruptVector: ADDRESS;
            vectorNr: CARDINAL;
          END;

  PROCEDURE InstallISR(VAR isr: ISR; proc: PROC;
                       interruptVectorNr: CARDINAL);
  (* Installs the user procedure 'proc' as an interrupt
     service routine for the specified interrupt vector.
     The procedure 'proc' can be declared in a priority
     module (Monitor). In this case, the interrupt mask
     will be automatically set by the run-time support
     upon entry to the user procedure.
  *)

  PROCEDURE UninstallISR(isr: ISR);
  (* Uninstalls the previously installed interrupt
     service routine
  *)

END ISR.
```

```
IMPLEMENTATION MODULE ISR;
(*
   Module, to support faster interrupt handling without
   context-switch upon interrupts.
   Allows installation of Modula-2 procedures as interrupt
   service routines.
*)

  FROM SYSTEM IMPORT
    ADDRESS, BYTE, ADR;

  IMPORT Devices;

   PROCEDURE InstallISR(VAR isr: ISR; proc: PROC;
                        interruptVectorNr: CARDINAL);
   BEGIN
     WITH isr DO
       pushAX := BYTE(50H);
       pushCX := BYTE(51H);
       pushDX := BYTE(52H);
       pushBX := BYTE(53H);
       pushSI := BYTE(56H);
       pushDI := BYTE(57H);
       pushDS := BYTE(1EH);
       pushES := BYTE(06H);
       farCall := BYTE(9AH);
       handlerProc := proc;

       sendEOI[1] := BYTE(0B8H); (* MOV AX, 20H *)
       sendEOI[2] := BYTE(20H);
       sendEOI[3] := BYTE(00H);
       sendEOI[4] := BYTE(0E6H); (* OUT 20H, AL *)
       sendEOI[5] := BYTE(20H);
```

```
      popES := BYTE(07H);
      popDS := BYTE(1FH);
      popDI := BYTE(5FH);
      popSI := BYTE(5EH);
      popBX := BYTE(5BH);
      popDX := BYTE(5AH);
      popCX := BYTE(59H);
      popAX := BYTE(58H);

      iret := BYTE(0CFH);
      vectorNr := interruptVectorNr;
      Devices.SaveInterruptVector(vectorNr, oldInterruptVector);
      Devices.RestoreInterruptVector(vectorNr, ADR(isr));
    END; (* WITH *)
  END InstallISR;

  PROCEDURE UninstallISR(isr: ISR);
  BEGIN
    Devices.RestoreInterruptVector(isr.vectorNr,  isr.oldInterruptVector);
  END UninstallISR;

END ISR.
```

An even faster method is to use the user procedure directly as interrupt service routine, by initializing the interrupt vector to the entry point of the procedure. In this case, the procedure code must make sure to save all registers, to send an end-of-interrupt to the interrupt controller and to return correctly from the interrupt.

Note that the procedure may not be declared within a priority module (Monitor), and stack test code should be turned off. This is because the compiler-generated calls to the run-time support for stack check and monitor entry are called before any user code, and therefore the saving of the registers comes too late.

To execute the interrupt handler with the correct priority, the user can explicitly include the run-time support function calls for monitor entry (exit) after saving (before restoring) the registers. Another method is to directly set the register mask of the interrupt controller. In this case, it must be guaranteed that the whole system doesn't use priority modules because the run-time support is not aware of the direct modification of the register mask and might change it to another (incorrect) value.

The following module shows the selection of an interrupt handler using this approach:

```
MODULE SimpleInterruptHandler;

  FROM SYSTEM IMPORT
    ADDRESS, CODE, SETREG, SWI, OUTBYTE;

  FROM Devices IMPORT
    SaveInterruptVector, RestoreInterruptVector;

  CONST
    deviceInterrupt  = ??;  (* interrupt vector number  *)
    priority         = ??;  (* priority level (0..7)    *)
                            (* lowest priority = 0      *)

    (* RTS function calls *)
    monitorEntry = 5;
    monitorExit = 6;

  VAR
    oldInterruptVector: ADDRESS;

(*$S-*)
(* Turn stack test off, because it is called before
   we save the registers, so some registers could be
   destroyed.
*)
```

```
PROCEDURE HandleDeviceInterrupt;
BEGIN
   (* compiler generated procedure prolog:
              PUSH BP
              SUB SP, variable space
   *)
   (* save all the registers *)
   CODE(50H); (* PUSH AX *)
   CODE(51H); (* PUSH CX *)
   CODE(52H); (* PUSH DX *)
   CODE(53H); (* PUSH BX *)
   CODE(56H); (* PUSH SI *)
   CODE(57H); (* PUSH DI *)
   CODE(1EH); (* PUSH DS *)
   CODE(06H); (* PUSH ES *)

   (*  here we could explicitly call the
       monitor entry function

           SETREG(BX, priority);
           SETREG(AX, monitorEntry);
           SWI(RTSCALL);
   *)

   (* actual code to handle the interrupt *)

   OUTBYTE(20H, 20H);
   (* send EOI to 8259 interrupt controller *)

   (* here we could explicitly call the
       monitor exit function

           SETREG(AX, monitorExit);
           SWI(RTSCALL);
   *)
```

```
        (* restore all saved registers *)
        CODE(07H); (* POP ES *)
        CODE(1FH); (* POP DS *)
        CODE(5FH); (* POP DI *)
        CODE(5EH); (* POP SI *)
        CODE(5BH); (* POP BX *)
        CODE(5AH); (* POP DX *)
        CODE(59H); (* POP CX *)
        CODE(58H); (* POP AX *)

        (* call explicitly procedure epilog with an IRET *)
        CODE(89H, 0ECH);  (* MOV SP, BP *)
        CODE(5DH);           (* POP BP *)
        CODE(0CFH);          (* IRET *)

    END HandleDeviceInterrupt;

(*$S=*)
(* reset stack option to previous state *)

BEGIN
    (* install the interrupt handler *)
    SaveInterruptVector(deviceInterrupt,  oldInterruptVector);
    RestoreInterruptVector(deviceInterrupt,  ADDRESS(HandleDeviceInterrupt));

    (* actual program code *)

    (* remove the interrupt handler *)
    RestoreInterruptVector(deviceInterrupt, oldInterruptVector);
END SimpleInterruptHandler.
```

5.2.3.4 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 Modula-2 (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-2/86 Compiler. The following program extract shows you how to get access to this information:

```
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:

```
IF criticalSectionPtr^ 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.

5.2.4 Implementation Notes

MODULA-2/86 implements priorities and device handling through the mask register of the interrupt controller in the 8086 system. The corresponding code is part of the MODULA-2/86 run-time support.

5.2.4.1 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 MODULA-2/86 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 MODULA-2/86 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 MODULA-2/86 program.

5.2.4.2 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.

5.2.4.3 The Interrupt Controller Mask

When a program is not running at any priority, MODULA-2/86 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 'interruptMask' 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.

5.2.4.4 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 MODULA-2/86 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.

5.2.5 The Definition Module '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-2/86.
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.
```

5.2.6 The Implementation Module '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-2/86.

A corresponding scheme can be used for interrupt driven
output devices.
*)

  FROM SYSTEM IMPORT
    PROCESS, NEWPROCESS, TRANSFER, IOTRANSFER,
    ADR, SIZE, BYTE, ADDRESS;

  FROM System IMPORT
    TermProcedure;

  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;


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;
        System.TermProcedure(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.
```

5.3 DOSCALL

The procedure DOSCALL must be imported from module SYSTEM. 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[O...35] OF CHAR;
    returnVal: CARDINAL

DOSCALL(15, ADR(FCB), returnVal);
IF returnVal=...THEN
```

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.

The formats of these functions are as follows:

Function 0H: Program Terminate

```
DOSCALL(0H)
        AH
```

Function 1H: Keyboard Input

```
DOSCALL(1H; VAR char:BYTEWORD);
        AH          AL
```

Function 2H: Display Output

```
DOSCALL(2H; char:BYTEWORD);
        AH  DL
```

Function 3H: Auxiliary Input

```
DOSCALL(3H; VAR char:BYTEWORD);
        AH          AL
```

Function 4H: Auxiliary Output

```
DOSCALL(4H; char:BYTEWORD);
        AH  DL
```

Function 5H: Printer Output

```
DOSCALL(5H; char:BYTEWORD);
        AH  DL
```

Function 6H: Direct Console I/O

```
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

```
DOSCALL(7H; VAR char:BYTEWORD);
        AH          AL
```

Function 8H: Console input without echo

```
DOSCALL(8H; VAR char:BYTEWORD);
        AH          AL
```

Function 9H: Print String

```
DOSCALL(9H; stringaddr:ADDRESS);
        AH  DS:DX
```

Function 0AH: Buffered Keyboard input

```
DOSCALL(0AH; stringaddr:ADDRESS);
        AH     DS:DX
```

Function 0BH: check standard input status

```
DOSCALL(0BH; VAR status:BYTEWORD);
        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.

```
DOSCALL(0CH; 1H; VAR char:BYTEWORD);
        AH   AL          AL

DOSCALL(0CH; 6H; VAR char:BYTEWORD;
        AH   AL        AL
        [DL  =  0FFH implicitly]
        VAR ready:BOOLEAN);
             ZF

DOSCALL(0CH; 7H; VAR char:BYTEWORD);
        AH   AL          AL

DOSCALL(0CH; 8H; VAR char:BYTEWORD);
        AH   AL          AL

DOSCALL(0CH; OAH; stringaddr:ADDRESS);
        AH   AL    DS:DX
```

Function 0DH: Disk reset

```
DOSCALL(0DH)
        AH
```

Function 0EH: Select Disk

```
DOSCALL(0EH; drive:BYTEWORD;
        AH   DL

        VAR nrofdrives: WORD);
             AL
```

Function 0FH: Open File

```
        DOSCALL(0FH; FCBaddr:ADDRESS;
                AH   DS:DX

                VAR returnCode:BYTWORD);
                    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
```

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);
                 AL
```

Function 2AH: Get Date

```
DOSCALL(2AH; VAR year:WORD; VAR monthday:WORD);
        AH   CX                 DX
```

Function 2BH: Set Date

```
DOSCALL(2BH; year:WORD; monthday:WORD;
        AH   CX         DX

        VAR returnCode:BYTEWORD);
            AL
```

Function 2CH: Get Time

```
DOSCALL(2CH; VAR hourminute, secondmillisec:WORD);
        AH       CX          DX
```

Function 2DH: Set Time

```
DOSCALL(2DH; hourminute, secondmillisec:WORD;
        AH   CX          DX

        VAR returnCode:BYTEWORD);
            AL
```

Function 2EH: Set/Reset Verify Switch

```
DOSCALL(2EH; zero:BYTEWORD; onoff:BYTEWORD);
        AH   DL             AL
```

5.3.1 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)

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
```

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
```

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 transferedbytes: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 transferedbytes: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 transferedbytes:BYTEWORD;
            AX

        VAR error:WORD);
            AX,CF
```

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; paramblock:ADDRESS;
        AH  DS:DX                ES:BX

            fctval:BYTEWORD;
            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
```

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
```

5.4 INTERFACING ASSEMBLY CODE LIBRARIES

Note: To encorporate the following changes you need the sources of the MODULA-2/86 run-time system.

MODULA-2/86 uses its own object file formats (LNK and LOD files). Therefore, it is not possible to link standard object files together with MODULA-2/86 programs. However, there are other ways to call routines written in assembly language from a MODULA-2/86 program. This section will explain such a way by means of an example. The sample assembly and Modula-2 programs referenced can be found at the end of this section.

The task of preparing a library of assembly code routines for use with MODULA-2/86 can be divided into five steps:

- Preparing the assembly library itself and verifying that it meets all the requirements
- Adapting the file RTS.ASM of the MODULA-2/86 run-time support
- Creating an executable file by assembling and linking your assembly library and the modified MODULA-2/86 run-time support
- Writing a Modula-2 definition module for the assembly library routines
- Implementing this module in Modula-2 by means of calls to the assembly routines

5.4.1 Sample Assembly Library

For our example we use a library of four assembly routines we would like to call from MODULA2/86 programs. The routines in this assembly library are called 'init', 'fct1', 'fct2', and 'fct3'. The routine 'init' takes no parameters and needs to be called to initialize this assembly library. The other routines take their parameters in registers. The assembly source code of this library is shown in the listing of the sample assembly library at the end of this section.

Note that any routines which will be called from a MODULA-2/86 program must be declared as 'far' procedures. This ensures that the assembler will generate a 'far' return instruction. It is not possible to call routines that are declared 'near' (instead of 'far') from a MODULA-2/86 program, because these routines must be called with an intrasegement call ('far' call) from the MODULA-2/86 program. This is due to the fact that MODULA-2/86 supports the full address space of the 8086 processor (large memory model), where code and data may be anywhere in memory.

For the same reason, any parameter given by an address must be handed over to the assembly routines by means of two registers, one containing the segment part and one containing the offset part of the address. In general, two variables declared in a MODULA-2/86 program will be allocated in different data segments.

5.4.2 Adapting the Run-Time Support

To make our assembly routines accessible from Modula-2, we also need to modify the main part of the MODULA-2/86 run-time support. The assembly sources of the MODULA-2/86 run-time support are provided on the distribution disks. For example, we copy the file RTS.ASM to a file we call NEWRTS.ASM. Then we can start to modify the copy. The necessary modifications are described in the following paragraphs. The names used, refer to the listing of the RTS extensions added at the end of this section.

First, we need to define and initialize a table ('EntryPointTable') that contains the entrypoints (addresses) of all our library routines (init, fct1, fct2, fct3).

We also need a procedure ('GetEntryPointTable') that returns the address (segment and offset) of this table. This procedure will be called by the Modula-2 counterpart of our assembly library through a software interrupt. Therefore, it must end with a return from interrupt instruction (IRET).

Then, we need to choose an interrupt vector ('LibraryVector') which will be used to call this initialization procedure. Any interrupt vector which is not occupied may be used. In our example we used the interrupt vector number 229, which is next to the interrupt number 228, used by the MODULA-2/86 run-time support.

The entry code of the run-time support should then be adapted such that it saves the old value of the library interrupt vector, and sets it to the address of the initialization procedure ('GetEntryPointTable'). This should be done at the same place where the RTS sets up its own interrupt vector, and similar code can be used. In the same way the original value of the library interrupt vector should be restored in the termination code of the run-time support.

5.4.3 Creating a New Executable File NEWM2.EXE

After the modifications have been done, we need to assemble and link our new run-time support, such that it includes our assembly library. For this purpose, we also need the other sources of the MODULA-2/86 run-time support which are provided on the distribution disks: RTS.INC, SERVICES.ASM, TRANSFER.ASM, LOADER.ASM, and DBUG.ASM.

We need to assemble all the .ASM files, including the modified NEWRTS.ASM and the file of our assembly library. This can be done by the DOS commands 'masm newrts;', 'masm services;', and so on.

In order to produce a new executable file we should then link the resulting object files. Assuming that the object code of our assembly library is in a file ASMLIB.OBJ, the DOS command:

```
link newrts+services+transfer+loader+dbug+asmlib;
```

can be used. This produces an output file NEWRTS.EXE,

which we can rename to NEWM2.EXE. Programs that use our assembly library must be started using this new run-time support, by typing newm2 <program name>.

5.4.4 Definition Module for the Sample Assembly Library

The definition module 'AsmLib' for our sample assembly library declares and exports Modula-2 procedures (Fct1, Fct2, Fct3) with appropriate parameter lists, corresponding to the routines in the assembly library (fct1, fct2, fct3). It is no different from a regular Modula-2 definition module. Please refer to the listing at the end of this section.

5.4.5 Implementation Module for the Sample Assembly Library

The implementation module 'AsmLib' for our sample assembly library is given at the end of this section. It declares a variable that corresponds to the table of entrypoints as it is declared in the assembly part. It is essential that the number and order of the procedures in these two tables match. The procedure 'InitLib' calls the procedure 'GetEntryPointTable' of the run-time support through the corresponding software interrupt. Then it copies the table of entrypoints.

The implementations of the exported procedures that correspond to the routines in the assembly library all follow the same scheme: The input parameters are copied into the appropriate registers, the function is called, and the returned values are copied to the output parameters.

Note that around the call to the assembly routine the value of the base pointer register BP is saved and restored. There is no need to save any other registers. (It is assumed that the stack segment SS and the stack pointer SP will be preserved.) The BP register may then be modified by the call, however, it should not be used to pass parameters. The value of the BP register is essential to MODULA-2/86 because it is used to access local variables and procedure parameters.

If there is more than one input parameter (several calls to SETREG), or if there are output parameters (one or more calls to GETREG), then only constants, or variables and value parameters which are declared local to the procedure, should be used with SETREG and GETREG. Also, the second argument of SETREG or GETREG 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 before the first call to SETREG. Only local variables of simple types should be used with GETREG. If necessary, their values should be copied to the real output parameters after the last call to GETREG. If these guidelines are not observed, calls to SETREG or GETREG might destroy parameter values which are in registers.

5.4.6 Sample Assembly Library Listing

```
;************************************************************
;
; this is a sample of a possible assembly library, for
; which we build a Modula-2 interface
;
mycode  group  libcode
mydata  group  libdata

        assume CS:      mycode
        assume DS:      mydata

;************************************************************
     public init; initialization routine of library
     public fct1; entrypoint of first library routine
     public fct2; entrypoint of second library routine
     public fct3; entrypoint of third library routine
;************************************************************

;************************************************************
libdata segment public 'data'
        ; data of the library
libdata ends
;************************************************************
```

```
;**********************************************************
libcode segment public 'code'

init    proc    far
        ; initialization code of the library
        ret
init    endp

fct1    proc    far
        ; entry parameters: description
        ; exit parameters: description
        ; code of the first library routine
        ret
fct1    endp

fct2    proc    far
        ; entry parameters: description
        ; exit parameters: description
        ; code of the second library routine
        ret
fct2    endp

fct3    proc    far
        ; entry parameters: description
        ; exit parameters: description
        ; code of the third library routine
        ret
fct3    endp

libcode ends
;**********************************************************

        end
```

5.4.7 RTS Extensions

```
;***********************************************************
;
; this is a listing of the code that must be inserted into
; the RTS, in order to build a Modula-2 interface for an
; assembly library
;
;***********************************************************

;***********************************************************
code segment public 'code'
; FROM lib IMPORT
        extrn   init:  far
        extrn   fct1:  far
        extrn   fct2:  far
        extrn   fct3:  far

code    ends
;***********************************************************

LibraryVectorNr EQU      229  ;    number of the interrupt
                              ;    vector which is used to
                              ;    call GetEntryPointTable

;***********************************************************
data    segment public   'data'

oldLibraryIV    dd       ?
        ; variable to save the original value of the
        ; library interrupt vector (LibraryVectorNr)

EntryPointTable dd      init, fct1, fct2, fct3
        ; table containing the entrypoints of all
        ; library routines

data ends
;**********************************************************
```

```
;**********************************************************
code    segment

GetEntryPointTable:
        ; return the address of the table that contains
        ; the entrypoints of the library routines
        ;
        ; exit parameters:
        ;   ES:SI hold the address of EntryPointTable
          MOV   AX, data
          MOV   ES, AX
          MOV   SI, offset data: EntryPointTable
          IRET

code      ends
;**********************************************************

; the following code must be inserted at the appropriate
; place into the Modula-2 run-time support (file RTS.ASM),
; in order to save the original value of the interrupt
; vector LibraryVectorNr, and to set it such that it can
; be used to call the function GetEntryPointTable
        ; save interrupt vector and set it to entrypoint
        ; of GetEntryPointTable interrupt service routine
          MOV   BX, LibraryVectorNr*4
        ; save the old value
          MOV   AX, ES: [BX]
          MOV   word ptr oldLibraryIV, AX
          MOV   AX, ES: 2[BX]
          MOV   word ptr oldLibraryIV + 2, AX
        ; set the new-one
          MOV   ES:word ptr [BX],offset GetEntryPointTable
          MOV   ES:word ptr 2[BX], CS

;**********************************************************
```

```
; the following code must be inserted at the appropriate
; place into the Modula-2 run-time support (file RTS.ASM),
; in order to restore the original value of the interrupt
; vector LibraryVectorNr, which is used to call the
; function GetEntryPointTable

        ; restore interrupt vector
          MOV   BX, LibraryVectorNr*4
        ; restore the old value
          MOV   AX,word ptr oldLibraryIV
          MOV   ES:word ptr [BX],AX
          MOV   AX,word ptr oldLibraryIV + 2
          MOV   ES:word ptr [BX]+2,AX

;**********************************************************
```

5.4.8 The Definition Module 'AsmLib'

```
DEFINITION MODULE AsmLib;
  (* this module is the Modula-2 interface to the sample
     library of assembly routines
  *)

  EXPORT QUALIFIED
    Fct1, Fct2, Fct3;
  PROCEDURE Fct1(parameters);
    (* Modula-2 declaration of the procedure interface of
       the first assembly library routine
    *)

  PROCEDURE Fct2(parameters);
    (* Modula-2 declaration of the procedure interface of
       the second assembly library routine
    *)
    PROCEDURE Fct3(parameters);
        (* Modula-2 declaration of the procedure interface of
            the third assembly library routine
        *)

END AsmLib.
```

5.4.9 The Implementation Module 'AsmLib'

```
IMPLEMENTATION MODULE AsmLib;
    FROM SYSTEM IMPORT
        ADDRESS, SWI, CODE, GETREG, SETREG, ES, SI;

    CONST
        LibraryVectorNr = 229;
            (* number of the interrupt vector used to call the
                function GetEntryPointTable in the modified
                RTS; this function returns the address of the
                table with the entrypoints of the assembly
                library routines.
            *)

        PushBP = 55H;
        PopBP = 5DH;
```

```
 TYPE
    EntryPointTable = RECORD
                          init: PROC;
                          (*  entrypoint of a possible
                                function in the assembly
                                library to initialize the
                                library
                          *)
                          fct1: PROC;
                          fct2: PROC;
                          fct3: PROC;
                          (*  entrypoints for all assembly
                                library routines
                          *)
                       END;

VAR
   fcts: EntryPointTable;
      (* local table with the entrypoints of all the
            assembly library routines. It is a copy of an
            identical table in the RTS. It is initialized
            by the procedure 'InitLib' of this module.
            We use a copy for performance reasons.
      *)

 PROCEDURE Fct1(parameters);
 BEGIN
    (* initialize entry parameters, using SETREG *)
    CODE(PushBP); (* save BP *)
    fcts.fct1;
    CODE(PopBP); (* restore BP *)
    (* get values of return parameters, using GETREG *)
 END Fct1;
```

```
PROCEDURE Fct2(parameters);
BEGIN
    (* initialize entry parameters, using SETREG *)
    CODE(PushBP); (* save BP *)
    fcts.fct2;
    CODE(PopBP); (* restore BP *)
    (* get values of return parameters, using GETREG *)
END Fct2;

PROCEDURE Fct3(parameters);
BEGIN
    (* initialize entry parameters, using SETREG *)
    CODE(PushBP); (* save BP *)
    fcts.fctn;
    CODE(PopBP); (* restore BP *)
    (* get values of return parameters, using GETREG *)
END Fct3;

PROCEDURE InitLib;
    VAR
        adr: ADDRESS;
        tableptr: POINTER TO EntryPointTable;
            (* pointer to the original table with the
                entrypoints of the assembly library routines.
            *)
```

```
  BEGIN
    SWI(LibraryVectorNr);
       (*  call the function GetEntryPointTable in the RTS,
             which returns the address of the table with the
             entrypoints of the assembly library routines
       *)
    GETREG(ES, adr.SEGMENT);
    GETREG(SI, adr.OFFSET);
       (* get the address of the entrypoint table
       *)
    tableptr := adr;
    fcts := tableptr^;
       (* copy the table into local table of this module
       *)
  END InitLib;

BEGIN
  InitLib;
  fcts.init;
     (*  call the initialization code of the assembly
           library
     *)

END AsmLib.
```

5.5 LIBRARY SEARCH STRATEGY

No special manipulation is required to build or to use the library of modules. All on-line modules, residing on hard disk or floppy disk, comprise the library. The compiler, linker, and debugger automatically search for referenced modules. The default search strategy can be modified by command options. Note that during the operation of the MODULA-2/86 compiler, linker and debugger, all needed files must be on-line. By default, several drives or directories are searched to find a library module.

For the following discussion we will use the term 'path' or 'path name' as an abbreviation for 'drive and/or directory name'.

5.5.1 Default Names

The MODULA-2/86 compiler, linker, post-mortem debugger, and run-time debugger construct the default filename for a library module from its module name. This is done by truncating the module name if it is longer than a file name may be, and by appending the appropriate extension (SYM, LNK, etc.) to the resulting name. This default filename is then used to find the corresponding file.

5.5.2 The Default Search Strategy

When a library module is needed, several paths will be checked automatically to find the corresponding file. A search strategy, as explained below, is applied by the compiler (for files with extension .SYM), by the linker (for .LNK and .MAP files), and by the debugger (for .REF, .MOD and .DEF files).

The first search is always done using the 'source' or 'master' path. The source or master path is the path you specify when entering the name of the file to compile or link. If the file is not found using this path, the current path -- the current drive and directory as known to your operating system -- is checked for the file. If it is not found there, a third and last automatic search is done using the path from where the compiler (or linker, debugger) was loaded. If the file still cannot be opened, you will be prompted to type in the (path and) file name.

This default search strategy is adequate only if your system has floppy disks. For hard disk systems it is recommended that you organize the disk as described in the installation section of this manual and that you change the default search strategy by setting up the environment (in DOS) for MODULA-2/86. This is done by the following DOS commands:

- SET M2SYM=\M2LIB\SYM<CR>
- SET M2LNK=\M2LIB\LNK<CR>
- SET M2REF=\M2LIB\REF<CR>
- SET M2MAP=\M2LIB\MAP<CR>

It is recommended that these DOS commands be included in your AUTOEXEC.BAT file, which is executed by DOS automatically every time you boot your system. If these environment strings (M2SYM, M2LNK, M2REF, M2MAP) are defined in DOS, they will be used by the MODULA-2/86 system in order to determine which paths are searched automatically. If these environment strings are not defined, then three automatic searches as explained above will be performed. If you use the recommended disk organization, then you should define these environment strings.

Each of these environment strings can denote a number of paths. Different paths must be separated by semicolons. If the environment strings are defined, then the MODULA-2/86 system - the compiler, linker and debugger - will search the library files using all the paths specified by the corresponding environment string. However, the first search is still done using the 'source' or 'master' path. If it fails, the paths specified in the corresponding environment string are checked one after the other according to the order in which they appear. If the file is not found, then you will be prompted to type in the (path and) file name.

Example:

Let us assume, that the M2SYM string has been set by

```
set m2sym=;\mystuff\mylib;\m2lib\sym
```

and that a compilation was started by

```
C> m2 comp<CR>
LOGITECH MODULA-2/86 Compiler, Rel. m.n
Copyright (C) 1983, 1984, 1985 LOGITECH
source file>a:myprog<CR>
```

In this case, the compiler will always perform a first search for symbol files on drive A, using the path you specified with the source file. If the file is not found there, it will try to open the file using the first path from M2SYM. In our case, this first path is empty, because the string starts with a semicolon. An empty path denotes the current directory on the current drive. If a symbol file is not found using the current path, the second path '\mystuff\mylib' is searched. If this search and the last automatic attempt to open a symbol file using the path '\m2lib\sym' both fail, you will be prompted to type in the file name.

This way of searching library files gives you a lot of flexibility to add your own library modules. The simplest way to add your own library modules is to put them into the directories with the standard library. As the example shows, you can also add new libraries of your own and organize it in your own way. Note that, unlike in the case where no environment strings are defined, the current directory is only searched if you did not specify a path when entering the source file name, or when an empty path occurs in the environment string.

5.5.3 The Query Search Strategy

The query search strategy is always applied by the MODULA-2/86 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 (or linking, or debugging).

When you are prompted, several responses are available. They are discussed in the following paragraphs.

- You can enter <ESC>, which means 'no file'. This can be used to indicate that the file is not available or, in the case of the linker, that it should not be included. Depending on the context, entering <ESC> may not allow the successful completion of the program you are currently using.

- You can enter <CR> only, which means that the file name should be constructed from the module name, and that the default search strategy (as explained above) will be applied.

- You can enter a file name only, without specifying any path name. If you do this, that file name will be used, but the file will still be searched for automatically according to the default search strategy. (Remark: Here an empty path name does not denote the current path. If you want to get a file from the current path then you must denote it by 'DK:'. Under DOS you may also use '.\'.)

- You can enter a path name only (terminated by ':' or by '\'). In this case 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.

- You can enter a complete path and file name. In this case also, there will be only one attempt to open the file.

5.6 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.

5.6.1 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 understandable to the user. Arithmetic operations are performed on numbers stored in an encoded, internal format. The procedures 'StrToDec' and 'DecToStr' are used to convert decimal numbers between internal and external format. Procedure 'StrToDec' is used to encode decimal numbers and procedure 'DecToStr is used to decode decimal numbers.

5.6.2 Types

Module 'Decimals' provides the following types:

- Type 'DECIMAL'

  Type 'DECIMAL' is used for the internal representation of a decimal number. Arithmetic operations are performed on variables of type 'DECIMAL'.

- Type 'DecState'

  A variable of type 'DECIMAL' has a state of type 'DecState' associated with it. This state may have the following values:

  - NegOvfl indicates a negative overflow
  - Minus indicates a negative decimal value
  - Zero indicates the value 0
  - Plus indicates a positive decimal value
  - PosOvfl indicates a positive overflow
  - Invalid indicates an invalid number

  The procedure 'DecStatus' returns the state of a decimal variable.

5.6.3 Variables

The following variables are exported from module 'Decimals':

- VAR 'DecValid'

  The variable '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.

- VAR 'Remainder'

  The variable '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 '?'.

5.6.4 Conversion and Status Procedures

The following conversion and status procedures are provided by module 'Decimals':

- Procedure 'StrToDec'

Procedure 'StrToDec' converts numbers from external to internal format. It is explained in greater detail below.

- Procedure 'DecToStr'

  Procedure 'DecToStr' converts numbers from internal to external format. It is explained in greater detail below.

- Procedure 'DecStatus'

  Procedure 'DecStatus' returns the current state of a decimal variable. It can be used to get the sign of a valid decimal number. When an operation fails, the user can call the procedure 'DecStatus' to determine the actual arithmetic error. 'DecStatus' specifies the error status of the decimal variable according to type 'DecState'.

5.6.5 Arithmetic Operations

The following arithmetic operations can be performed with variables of type 'DECIMAL':

- Procedure 'CompareDec'

  Performs the comparison of two decimal values, Dec0 and Dec1. The output is an integer value as follows:

  - -1 if Dec0 is less than Dec1
  - 0 if Dec0 equals Dec1
  - 1 if Dec0 is greater than Dec1

- Procedure 'AddDec'

  Performs the addition of two decimal values, Dec0 and Dec1. The output is the sum of the two values, a decimal value.

- Procedure '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.

- Procedure '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.

- Procedure 'DivDec'

  Performs the division of one decimal value by another. The dividend, Dec0 is divided by the divisor, Dec1. The output is the quotient of the two values, a decimal value.

  The remainder is placed in the global variable 'Remainder' as previously explained.

- Procedure 'NegDec'

  Returns the negative value of a decimal value.

5.6.6 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 ZZZZZZZZZZZ9 the same decimal number would have appeared as:

    92384137138

5.6.7 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. A '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.

5.6.8 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.

5.6.9 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:

- PosOvfl is represented by '+++++++'
- NegOvfl is represented by '-----------'
- Invalid is represented by '???????'

The length of the string is determined by the length of the corresponding picture.

5.6.10 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.

Addition and Subtraction:

A/B	NegOvfl	Minus	Zero	Plus	PosOvfl	Invalid
NegOvfl	NegOvfl	NegOvfl	NegOvfl	NegOvfl	Invalid	Invalid
Minus	NegOvfl				PosOvfl	Invalid
Zero	NegOvfl				PosOvfl	Invalid
Plus	NegOvfl				PosOvfl	Invalid
PosOvfl	Invalid	PosOvfl	PosOvfl	PosOvfl	PosOvfl	Invalid
Invalid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid

Multiplication:

A/B	NegOvfl	Minus	Zero	Plus	PosOvfl	Invalid
NegOvfl	PosOvfl	PosOvfl	Zero	NegOvfl	NegOvfl	Invalid
Minus	PosOvfl	Plus	Zero	Minus	NegOvfl	Invalid
Zero	Zero	Zero	Zero	Zero	Zero	Invalid
Plus	NegOvfl	Minus	Zero	Plus	PosOvfl	Invalid
PosOvfl	NegOvfl	NegOvfl	Zero	PosOvfl	PosOvfl	Invalid
Invalid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid

Division:

A/B	NegOvfl	Minus	Zero	Plus	PosOvfl	Invalid
NegOvfl	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid
Minus	Invalid	Plus	Invalid	Minus	Invalid	Invalid
Zero	Zero	Zero	Invalid	Zero	Zero	Invalid
Plus	Invalid	Minus	Invalid	Plus	Invalid	Invalid
PosOvfl	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid
Invalid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid

5.7 REAL ARITHMETIC

The Modula-2 programming language provides the data type REAL for floating point arithmetic. MODULA-2/86 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 to the minus 308th power, to 1.79 times 10 to the 308th power:

$$2.23E\text{-}308 <= |x| <= 1.79E\text{+}308.$$

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.

MODULA-2/86 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.

5.7.1 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 prefix 'C87' has been added to the names of the object files of those MODULA-2/86 library modules that contain 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' which provides the REAL emulation. The prefix 'E87' has been added to the names of the object files of those MODULA-2/86 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 with the prefix 'E87', your program will always use the emulator for REAL arithmetic.

A MODULA-2/86 program compiled with the Emulator option can also be linked in other ways. It is still possible to link the program such that it requires 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 use of these advanced features of MODULA-2/86 is described in the following sections.

5.7.2 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.

MODULA-2/86 offers the following alternatives for REAL arithmetic:

- At compile time the compiler options Coprocessor and Emulator determine the kind of code generated. The user can 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 module 'Reals' which provides the REAL emulation. The compiler implicitly knows the interface of module 'Reals', therefore no symbol file needs to be provided.

- If you choose 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. The link files with the prefix 'C87' should be used when linking the program. For those library modules that use REAL arithmetic the link files with the prefix 'C87' contain the object code for the 8087.

- If you choose to compile a module for the LOGITECH REAL emulator, how the program will be linked is still flexible. MODULA-2/86 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 the prefix 'E87'. 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 with the prefix 'C87'. A program linked with the files with the prefix 'C87' 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 run-time 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 files with the prefix 'M87' must be used for the modules 'Reals' and 'MathLib0', and the files with the prefix 'E87' must be used for all other library modules that use REAL arithmetic.

Because of the prefix 'E87', 'C87', or 'M87' in the filename, the MODULA-2/86 linker cannot find the corresponding library modules automatically. Therefore, you must decide which Real modules you want to use: emulation, coprocessor or mixed. Then, you must copy these files to files with appropriate names, such that the linker will find them. The following list tells you which files to copy depending upon which method you decide to use:

8087 Inline Code

Compile all your modules which use floating point arithmetic with the /C option.

- Reals
  You will not use the emulator.
- MathLib0
  Copy C87Math0.LNK to MathLib0.LNK.
- RealConversions
  Copy C87RealC.LNK to RealConv.LNK.
- RealInOut
  Copy C87RealI.LNK to RealInOu.LNK.

Pure Emulator

Compile all your modules which use floating point arithmetic with the /E option.

- Reals
  Copy E87Reals.LNK to Reals.LNK.
- MathLib0
  Copy E87Math0.LNK to MathLib0.LNK.
- RealConversions
  Copy E87RealC.LNK to RealConv.LNK.
- RealInOut
  Copy E87RealI.LNK to RealInOu.LNK.

8087 Version of Emulator

Compile all your modules which use floating point arithmetic with the /E option.

- Reals
  Copy C87Reals.LNK to Reals.LNK
- MathLib0
  Copy C87Math0.LNK to MathLib0.LNK
- RealConversions
  Copy E87RealC.LNK to RealConv.LNK.
- RealInOut
  Copy E87RealI.LNK to RealInOu.LNK.

Mixed Emulator

Compile all your modules which use floating point arithmetic with the /E option.

- Reals
  Copy M87Reals.LNK to Reals.LNK.
- MathLib0
  Copy M87Math0.LNK to MathLib0.LNK
- RealConversions
  Copy E87RealC.LNK to RealConv.LNK.
- RealInOut
  Copy E87RealI.LNK to RealInOu.LNK.

COMPARISON OF THE DIFFERENT ALTERNATIVES

5.7.4 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.

5.7.5 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 5.7-1 lists the approximate memory requirements (code and data) for the different implementations of the modules 'Reals' and 'MathLib0'. All numbers are given in bytes.

	Reals	MathLib0	Reals & MathLib0
8087 Inline Code	---	1700	1700
8087 Version of Emulator	500	1700	2200
Pure Emulator	2300	4200	6500
Mixed Emulator	2800	6100	8900

TABLE 5.7-1

5.7.6 Performance

Table 5.7-2 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 5.7-2 allows for a relative comparison of the performance the user can expect when choosing a particular alternative for REAL arithmetic in MODULA-2/86. On the average, 8087 inline code is approximately ten times faster than full emulation of REAL arithmetic on the 8086.

	Addition	Multiplication	Division	Combination
8087 Inline Code	0.11	0.11	0.11	6.6
8087 Version of Emulator	0.22	0.22	0.22	8.2
Mixed Emulator (with 8087)	0.31	0.31	0.31	12.5
Pure Emulator	1.0	1.2	2.1	61.0
Mixed Emulator (without 8087)	1.1	1.3	2.2	63.0

TABLE 5.7-2

5.8 MEMORY ORGANIZATION

5.8.1 Global Memory Organization

The global memory organization when executing a MODULA-2/86 program is shown in Figure 5.8-1. This setup is established by the MODULA-2/86 run-time support.

After loading a program the run-time support creates the main process, which will execute the program. It then transfers control to this main process. The process descriptor of the main process is stored in the data segment of the run-time support.

```
 -------------------------------------------
|          Interrupt vector table           |
 -------------------------------------------  400H
|             Operating system              |
 -------------------------------------------  RTS address
|  Code of the MODULA-2/86 run time system  |
|                 support                   |
 -------------------------------------------  RTS_DS
|      Data of the MODULA-2/86 run-time     |
|                 support                   |
 -------------------------------------------  M2_START_MARK
|      Code of the MODULA-2/86 program,     |
|      loaded by the run-time support       |
 -------------------------------------------
|      Global data of the MODULA-2/86       |
|                 program                   |
 -------------------------------------------  START_MEMORY
|          Heap of main process             |
 -------------------------------------------  Top of main
|                                           |     heap
|               free memory                 |
 -------------------------------------------  Top of main
|                                           |     stack
|          Stack of main process            |
 -------------------------------------------  M2_END_MARK
|              unused memory                |
 -------------------------------------------

                                              TOP_OF_MEMORY
```

FIGURE 5.8-1
GLOBAL MEMORY ORGANIZATION

The memory locations indicated in Figure 5.8-1 have the following definitions:

- RTS address

  Load address of the MODULA-2/86 run-time support. This address depends on the size of the operating system. The RTS is a relocatable program, containing a code and a data segment.

- RTS_DS

  Address of the data segment of the MODULA-2/86 run time support.

- M2_START MARK

  Load address for a Modula-2 program. The MODULA-2/86 code is relocatable and is loaded immediately after the end of the run-time support.

- START_MEM

  Address of the first free paragraph after the code and data of a Modula-2 program loaded by the RTS. At this address the heap of the main process starts.

- Top of main heap

  Current top of heap for the main process. The heap grows toward the stack. When the program uses module Storage' to dynamically allocate and deallocate memory, the heap top may be increased and decreased. If the heap grows too close to the top of the stack, the program is aborted with a heap overflow.

- Top of main stack

  Current top of stack for the main process. The stack grows toward the heap. For each active procedure, the stack contains a procedure activation record. The structure of this record is explained below. If the stack grows too close to the top of the heap, the program is aborted with a stack overflow.

- M2_END_MARK

  End of the memory area actually used by the Modula-2 program and the run-time support.

- TOP OF MEMORY

  End of memory area which in principle could be used by the Modula-2 program and the run-time support. The run-time support gets the value for this address from the operating system.

5.8.2 Subprograms and Overlays

The MODULA-2/86 environment offers a standard overlay schema, with the following characteristics:

- Overlays can be loaded in the form of subprograms. This feature is provided by procedure 'Call' from module 'Program'.
- The size of an overlay is limited only by the physical address space, a maximum of one megabyte.
- The programmer is not concerned about where an overlay is loaded. To load and execute an overlay, one simply calls the procedure 'Call' from module 'Program'. This procedure finds a memory area, on top of stack, large enough to hold the overlay. The relocatable loader then loads the program in that area and control is passed to it. After termination of the overlay, either normal termination or termination caused by an error, execution continues at the statement that immediately follows the call to the procedure 'Call', and the memory occupied by the overlay is freed.
- Only the main process can call subprograms. The procedure 'Call' must not be used from coroutines that have beeen created by means of 'NEWPROCESS'. If the subprogram is called in a 'shared' way (see definition module 'Program'), there is no distinction between the old and the new heap of the main process. Only one heap exists. Any dynamic storage allocated by the subprogram remains allocated upon termination of the subprogram.

If the subprogram is called in an 'unshared' way, it uses its own private heap. If the main process uses module 'Storage' to dynamically allocate memory, the private heap of the subprogram is used. During the execution of the subprogram no memory can be allocated from the old heap of the main program. Upon termination of the subprogram all the memory allocated dynamically by the subprogram is freed and the heap is reset to the old heap.

In either case, the stack is set back to the old stack when the subprogram terminates. This automatically frees the memory area occupied by the code and global data of the subprogram.

After loading of a subprogram, the memory layout is as shown in Figure 5.8-2. (Only the higher part of memory is shown. See also Figure 5.8-1.)

```
|                                      | lower address
 --------------------------------------
|        Code of main program          |
 --------------------------------------
|     Global data of main program      |
 --------------------------------------  START_MEM
|       Old heap of main process       |
 --------------------------------------  Old top of heap
|       New heap of main process       |
|     (heap used by the subprogram)    |
 --------------------------------------  New top of heap
|              Free memory             |
 --------------------------------------  New top of stack
|      New stack of main process       |
|     (stack used by the subprogram)   |
 --------------------------------------  Load address
|        Code of the subprogram,       |
|        loaded by 'Program.Call'      |
 --------------------------------------
|    Global data of the subprogram     |
 --------------------------------------  Old top of stack
|       Old stack of main process      |
 --------------------------------------  M2_END_MARK
|             unused memory            |
 --------------------------------------  TOP_OF_MEMORY
```

FIGURE 5.8-2
MEMORY ORGANIZATION FOR SUBPROGRAMS

5.8.3 Processes

When starting a Modula-2 program, the MODULA-2/86 run-time support automatically creates the main process. This 'default' process gets the largest available region of memory as its workspace. The program which is loaded by the run-time support runs as the main process. Subprograms loaded by the Modula-2 program through module 'Program' run as overlays to the main process. No new process is created to execute subprograms. The Figures 5.8-1 and 5.8-2 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 father 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 organization of the workspace of a process created by a call to 'NEWPROCESS' differs slightly from the organization of the workspace of the main process. It is used for the local and dynamic variables of the process (stack and heap). A part of the workspace also contains the process descriptor, which is needed for the TRANSFER mechanism. This process descriptor is initialized by the call to procedure 'NEWPROCESS'. Figure 5.8-3 shows the organization of the process workspace.

```
 -------------------------------------   Workspace address
|          Process descriptor          |
 -------------------------------------
|           Heap of process            |
 -------------------------------------   Top of process
|         free memory inside of        |      heap
|               workspace              |
 -------------------------------------   Top of process
|           Stack of process           |     stack
 -------------------------------------   Workspace end
```

FIGURE 5.8-3
PROCESS WORKSPACE

The process descriptor of a process created by 'NEWPROCESS' starts at the first paragraph boundary in the process workspace. Approximately 200 bytes are needed for the process descriptor, the initial heap and the initial stack. In addition, at any point in time, there should be approximately 200 bytes of free memory in the process workspace. 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' is 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 MODULA-2/86 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 2 K BYTES. For more complicated processes, a larger workspace is required.

The workspace of a process must be large enough to hold its heap and stack. If the process heap and stack grow too much, using all of the free memory in the process workspace, the program containing this process is aborted with a heap or stack overflow. The maximum size of a process workspace is approximately 64K bytes.

5.8.4 Allocation of Variables

Local variables are variables which are declared inside of 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 allocated at a different place inside of the procedure activation record.

Because the procedure activation record only exists while the procedure is being executed, the lifetime of local variables and of the 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 if it is called by more than one process at the same time, several instances of its procedure activation record will exist.

Global variables are variables which are declared in modules which are not local to a procedure, for instance, in a definition module. They come into existence when the program or subprogram that contains this module is loaded. They are of a more permanent nature than local variables. Their lifetime is limited by the lifetime of the program or subprogram to which they belong.

In general, there exists only one instance of the global variables of a module. Global variables are shared by all the 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 the 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. The use of alignment can improve the performance of the program for an 8086 based system.

The maximum total size of 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 the physical address space which is at most one megabyte on an 8086 or 8088 based system.

5.8.5 The Heap

The heap of the main process starts immediately after the end of the code and data of the Modula-2 program loaded by the run-time support. To avoid unpredictable memory occupation due to the concurrency of multiple processes, the heap of every process is administrated independently. For any process created by 'NEWPROCESS' the heap is the memory area between the bottom of the workspace, after the process descriptor, and the top of stack.

The library module 'Storage' implements this default heap management. It provides procedures 'ALLOCATE' and 'DEALLOCATE', which allocate or deallocate memory from the heap of the calling process. Inside of the corresponding heap, the same strategy of allocation and deallocation is used for the main process and all other processes. The heap grows toward the stack which occupies the high end of the workspace of the process. If upon a call to 'ALLOCATE', the heap grows too close to the stack, the heap manager aborts the Modula-2 program with a heap overflow.

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, some procedure 'ALLOCATE' or 'DEALLOCATE' must be imported or declared in that module. The standard way of doing this 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 used for allocation and deallocation of dynamic memory will then differ from the default strategy that is provided by module 'Storage'.

5.8.6 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 the heap which occupies the lower end of the workspace.

The maximum size a stack can be is 64K bytes. However, in most applications the workspace of a process will be less than 64K bytes. Therefore, the stack size is, in general, limited by the size of the workspace and the occupation of the heap.

When the main process loads a subprogram, the current stack is 'frozen', and the subprogram is loaded on top of it. After this, a new stack is created which starts at the address just below the load address of the subprogram. The value of the stack segment is adjusted, such that the new stack can again grow up to 64K bytes. When the subprogram terminates, the stack segment and the stack pointer are reset to the old 'frozen' stack. This automatically removes the stack, code, and data of the subprogram. The value of the stack segment of any process other than the main process is never modified.

5.8.7 The Procedure Activation Record

Each time a procedure call is executed, a new procedure activation record is created on the stack of the current process. Depending on whether code for the 8086/8088 or 186/286 processors is generated, the format of the activation records differs slightly. The procedure activation record contains the following information (see also Figures 5.8-4 and 5.8-5 below):

- Procedure parameters

  The parameters, if any exist, are pushed onto the stack in the order in which they are declared. Because the stack grows toward lower addresses, the last parameter is found at the lowest address.

- Static link

  The static link is a pointer, within the same stack, to another procedure activation record which constitutes the static environment of the procedure. The static link is used to find variables or parameters in the static environment of the procedure. The static link only exists 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). In the case of code generated for 186/286, 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 consists of 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

  The dynamic link points to the previous procedure activation record within the same stack.

- Display (for 186/286 only)
  The display is a table of pointers, within the same stack, to the other procedure activation records which constitute 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 option for code generation for 186/286 was selected. In the case of 8086/8088 code, access to the static environment is implemented by the static link.

- Local data

  The local data consists of all the variables which are declared inside of the procedure.

```
                                          low addresses
|                                      |
----------------------------------------  stack pointer
|      Local data of procedure         |
----------------------------------------  base pointer
|            Dynamic link              |
----------------------------------------
|         Return offset (IP)           |
----------------------------------------
|      Return code segment (CS)        |
----------------------------------------
|            Static link               |
----------------------------------------
|          Last parameter              |
|               ...                    |
|          First parameter             |
----------------------------------------
|                                      |
                                          high addresses
```

FIGURE 5.8-4
PROCEDURE ACTIVATION RECORD FOR 8086/8088

```
                                               low addresses
|                                      |
 --------------------------------------        stack pointer
|      Local data of procedure         |
 --------------------------------------
|               Display                |
 --------------------------------------        base pointer
|            Dynamic link              |
 --------------------------------------
|         Return offset (IP)           |
 --------------------------------------
|      Return code segment (CS)        |
 --------------------------------------
|           Last parameter             |
|                 ...                  |
|           First parameter            |
 --------------------------------------
|                                      |
                                                high addresses
```

FIGURE 5.8-5
PROCEDURE ACTIVATION RECORD FOR 186/286

5.8.8 Procedure Calling Conventions

A procedure is called with a 'far' intersegment call if at least one of the following conditions is true:

- The procedure is imported from another separately compiled module.
- The procedure is exported from a definition module.
- The procedure is used in an assignment to a procedure variable or as a procedure parameter.
- The procedure 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 the following prologue is executed in the calling procedure:

- The 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 of 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.

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 the base pointer BP is set to the current value of the 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 186/286:

- 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 186/286:

- The instruction LEAVE is called. This instruction automatically removes local variables, display, and dynamic link and resets BP and SP.
- A return instruction is executed which 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.

5.8.9 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 BL.
- Two byte values are passed back in register BX.
- Four byte values are passed back in register ES and in register BX.
- REAL values are always passed back on top of the stack.

Note that in the current release, arrays and record types are not allowed as function types.

5.9 VERSION CHECKING

5.9.1 Module Key and Version Checking

All modules in a program must be compiled with a consistent version of module definitions. When a module definition file is modified all program and implementation modules using that module must be recompiled before a new executable program can be created. Modification of a definition part means a particular compilation into a SYM file. Each time a definition module is compiled, it will produce a new version of that module, which is incompatible with any other version of that module. Even if you do not change the text of the definition part, recompilation will create a new version of the SYM file.

MODULA-2/86 checks for version consistency and will not allow inconsistent versions to be compiled together. The version checking mechanism is simple in concept, but can be complex in application. Each time a definition module is compiled, a new module key is created and included in the resulting SYM file. This key will be different each time the module is compiled.

Once a definition part has been compiled, it becomes possible to compile its implementation part, or another module which uses the definition part - a 'client' module. These other modules will import that 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 included in the compiled output.

At compile, link and load time, MODULA-2/86 verifies that all the keys included for a given definition module are the same. This guarantees that all modules which share an interface were compiled using the same version of that interface. The purpose of this is to ensure the consistency of the program, as if there was only one source file, compiled all at once.

5.9.2 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.

5.9.3 Version Errors During Compilation

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

Example:

```
A.MOD   imports B and C

B.DEF   imports X

C.DEF   imports X
```

Suppose that we compile as follows:

```
X.DEF  =>     X.SYM (version 1)

B.DEF  =>     B.SYM (uses version 1 of X)

X.DEF  =>     X.SYM (version 2)

C.DEF  =>     C.SYM (version 2 of X)

A.MOD=>       A.LNK
```

There will be a version inconsistency error when A.MOD is compiled, because the version of X imported through B is not the same as the version imported through C. 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.

5.9.4 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:

MAIN.MOD imports InOut and Terminal.

INOUT.DEF defines InOut and imports nothing.

INOUT.MOD implements InOut and imports Terminal.

TERMINAL.DEF defines Terminal and imports nothing.

TERMINAL.MOD implements Terminal and imports nothing.

Now suppose these compilations are done:

TERMINAL.DEF	=>	TERMINAL.SYM (version 1)
INOUT.DEF	=>	INOUT.SYM
INOUT.MOD	=>	INOUT.LNK (uses version 1 of Terminal)
TERMINAL.MOD	=>	TERMINAL.LNK (corresponds to version 1)
TERMINAL.DEF	=>	TERMINAL.SYM (version 2)
MAIN.MOD	=>	MAIN.LNK (uses version 2 of Terminal)

Now, linking the program MAIN will generate a version conflict between the version of TERMINAL(.SYM) used by MAIN, and the version used by TERMINAL and INOUT. A solution is to recompile INOUT.MOD and TERMINAL.MOD with the new TERMINAL.SYM and link again.

5.9.5 Version Errors During Loading

When loading a subprogram (overlay), it is possible to have a version error between the program being loaded and the modules which are already in memory. This is always due to two modules, one loading and one already resident, having been compiled with different versions of some interface.

The following is a typical case:

There is a program which contains a module 'Windows' with an interface WINDOWS.DEF. This program calls a subprogram, in which there is a module 'Edit' which imports 'Windows'.

Suppose that WINDOWS.DEF and WINDOWS.MOD are recompiled after the base (main program) has been compiled and linked. Then if EDIT is compiled and linked into its subprogram, that subprogram will be inconsistent with the main program. An error will occur when the main program tries to load the subprogram, because two different versions of module 'Windows' are used. (There is no way to detect it sooner.) The program loader will return to its caller with an error status indicating that there was a module version conflict.

The straightforward correction is to recompile any module in the base that uses 'Windows' and to relink the base. Then the base layer and the subprogram 'Edit' will use the same version of 'Windows'.

6 THE COMPILER

The compiler translates the high level language Modula-2 (filetype .DEF, .MOD) into low level machine code in a linkable object file (filetype .LNK).

6.1 How to Use the Compiler

The compiler is run by typing 'm2 comp'. After displaying the banner with the version number, the compiler will prompt for the filename of the module to be compiled:

```
A> m2 b:comp<CR>

LOGITECH MODULA-2/86 Compiler, Rel. m.n
Copyright (C) 1983, 1984, 1985 LOGITECH
source file>
```

Enter the filename and any option you wish to specify (see the subsection on compiler options below). The default drive is the current disk and the default filetype is 'MOD' for program and implementation modules.

When the compilation of this source is finished the compiler will ask for another source file to compile. You can either enter another filename or terminate the compiler by typing <ESC>.

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

```
A> m2 b:comp filename1 filename2 ... filenameX<CR>
```

In this case the compiler will compile all the files in the specified order. At the end it will return automatically to DOS without any further request for source files.

If you have installed the fully linked version of the compiler (M2C.EXE) you can run it by typing:

```
A> m2c<CR>
```

or

```
A> m2c exampl<CR>
```

To run the fully linked version of the compiler you need at least 512KB of memory. With the fully linked version, the compilation time is reduced by one third.

6.2 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 the compiler base. The necessary files are:

▪ COMP.LOD	compiler base
▪ M2INIT.LOD	initialization
▪ M2PASS1.LOD	syntax analysis
▪ M2PASS2.LOD	declaration analysis
▪ M2PASS3.LOD	block analysis
▪ M2PASS4.LOD	code generation
▪ M2SYMFIL.LOD	symbol file generation
▪ M2LISTER.LOD	lister

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

During compilation temporary work files are created on the current drive. They are deleted before the termination of a compilation.

6.3 Compiler Output Files

Several files are generated by the compiler. They are given the same file name, directory name and device as the source file, with the appropriate filetype attached as follows:

- .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.

- .LNK Object (Link) file
  Compiler output file with the generated 8086 object code in linker format, generated during compilation of an implementation or a program module.

- .LST Listing file
  Normally generated only if errors occur.

6.4 Compilation of a Program Module

Compilation of a program module, in which there are no errors, generates a linkable object file (.LNK) and a debug reference file (.REF). If there are errors, the link and reference files are not produced, but a listing file is produced. The 'L' option (see section on compiler options below) directs the compiler to generate a listing file even if there are no errors.

```
A> m2 b:comp<CR>

LOGITECH MODULA-2/86 Compiler, Rel. m.n
Copyright (C) 1983, 1984, 1985 LOGITECH
 source file> exampl<CR>
Syntax Analysis
 Terminal in file: B:Terminal.SYM
Declaration Analysis
Block Analysis
Code Generation
Termination
 The interactive setting of the options was:S+/R+/T+/A-
 code for 8086/8088 generated
 Codesize: 90 bytes Datasize: 1 bytes
End Compilation
LOGITECH MODULA-2/86 Compiler, Rel. m.n
Copyright (C) 1983, 1984, 1985 LOGITECH
 source file> <ESC>
 ---- no compilation
Termination
End Compilation
A>
```

'Syntax Analysis', '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 found the error. The errors are displayed on the screen and a listing file with error messages is generated.

```
A> m2 b:comp<CR>

LOGITECH MODULA-2/86 Compiler, Rel. m.n
Copyright (C) 1983, 1984, 1985 LOGITECH
source file> exampl<CR>
Syntax Analysis
     Terminal in file: B:Terminal.SYM
 ---- error
Lister
  3     VAR ch CHAR;
******            ^37
* 37: ':' expected
Termination
End Compilation
A>
```

When the error display is more than one page, the compiler will ask if you want to see more errors after each page. If you hit any key it will continue. If you type <ESC> it will stop the error display. In both cases, it will generate the error listing. This is true unless the compiler option '/batch' is used. In this case the compiler will not ask for more errors.

6.5 Compilation of a Definition Module

Compilation of a definition module (filetype 'DEF') is similar to the compilation of a program module. However, as the result of a successful compilation of a definition module, the compiler produces a symbol file (filetype 'SYM'), while as the result of the compilation of a program or implementation module the compiler produces a linkable object file ('LNK'). 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 PRIOR TO ITS IMPLEMENTATION MODULE.

A DEFINITION MODULE MUST BE COMPILED PRIOR TO ANY MODULE THAT IMPORTS IT.

Example:

```
A>_m2 b:comp find.def<CR>

LOGITECH MODULA-2/86 Compiler, Rel. m.n
Copyright (C) 1983, 1984, 1985 LOGITECH
  source file> find.def
Syntax Analysis
Declaration Analysis
Symfile
Termination
End compilation
A>
```

6.6 Compilation of an Implementation Module

Compilation of an implementation module is similar to the compilation of a program module. At compilation of an implementation module the symbol file for this module is needed. This symbol file is produced by the compilation of the corresponding definition module, prior to the compilation of the implementation module.

The compiler output files are the same as those generated when compiling a program module. A linkable object file (.LNK) and a debug reference file (.REF) are generated as the result of a successful compilation. In case of errors only a listing file is produced.

```
A> m2 b:comp find<CR>

LOGITECH MODULA-2/86 Compiler, Rel. m.n
Copyright (C) 1983, 1984, 1985 LOGITECH
 source file> find.mod
Syntax Analysis
 Examp3         in file: A:Examp3.SYM
 Storage        in file: B:Storage.SYM
Declaration Analysis
Block Analysis
Code Generation
Termination
 The interactive setting of the options was:S+/R+/T+/A-
 code for 8086/8088 generated
 Codesize: 234 bytes Datasize: 56 bytes
End Compilation
A>
```

6.7 Symbol Files Needed for Compilation

Symbol files are used by the compiler to provide full inter-module checking. Upon compilation of a definition module, a symbol file containing symbol table information is generated. When the corresponding implementation part is compiled, or when another module - a 'client' - is compiled which imports it, the appropriate symbol file must be read.

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

If a symbol file is not found at all, 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 the user 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 <ESC> is pressed. Pressing <ESC> means that the file is not available. The compiler will stop at the end of the first pass, but first it will list all the required symbol files. This allows you to detect any other missing files.

6.8 Compiler Options

When reading the source file name, the compiler can also accept some options. Options are entered immediately following the filename, with each option preceded by '/'. An option value is a predefined string that defines the state of the corresponding option. The possible values for the following compiler options are listed below, and an explanation of their effects is included thereafter.

6.8.1 Table of Available Options

Option	Value for ON	Value for OFF	Default
query	Query	NOQuery	NOQ
autoquery	Aquery	NOAquery	AQ
interactive	Interactive	Batch	I
listing	Listing	NOListing	NOL
errorlisting	EListing	NOEListing	EL
emulator/ coprocessor	Emulator	Coprocessor	E
8086/80286	2	8	8
version	Version	NOVersion	NOV
statistics	STATistics	NOSTATistics	STAT
stack test	S+	S-	S+
range test	R+	R-	R+
index test	T+	T-	T+
alignment	A+	A-	A-
header in listing	Header	NOHeader	H
footer in listing	Footer	NOFooter	NOF
date in listing	DAte	NODAte	NODA
debug	Debug	NODebug	D

The default value for all options can be set in the compiler parameter module. The defaults shown are those of the distributed compiler. If, after using the compiler for a while, you want to alter the default settings, consult the appendix on system configuration. In the above list of values, the portion in upper case letters must be entered when specifying an option value. The complete name may be given. For the R, S and T options the '+' sign is optional.

6.8.2 Description of the Options

- /Q /NOQ Query option
- /A /NOA Autoquery option

These options define the search mechanism for the symbol files of the imported modules. The following table shows the possible combinations of the setting of these options and the corresponding behavior of the compiler:

Query	Autoquery	Action
Query	Aquery	Ask for filenames
Query	NOAquery	Ask for filenames
NOQuery	Aquery	Tries to find file by default strategy If not found it asks for filename.
NOQuery	NOAquery	If not found compiler ends.

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

- /I /B Interactive/Batch option

This option tells the compiler whether it runs interactive or as a batch job. In the interactive mode, the display of error messages is stopped after a screen page and can be resumed by hitting a key. This facility is turned off in the batch mode. Note: The Autoquery option is not affected by this option.

- /L /NOL Listing option

- /EL /NOEL Error Listing option

These options define whether a listing is generated or not. The following table shows the possible combinations of the setting of these options and the corresponding behavior of the compiler:

Listing	EListing	A listing file is always generated.
Listing	NOEListing	Same as above
NOListing	EListing	If errors are detected, a short error listing file is generated containing only the erroneous lines with error messages
NOListing	NOEListing	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 option

  This option affects the code generation for the floating point arithmetic. If it is 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 option

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

- /STAT /NOSTAT Statistics option

  At the end of a compilation the compiler displays some statistics on the generated code.

- /S+ /S- Stack test option
- /R+ /R- Range and Overflow test option
- /T+ /T- Index and NIL pointer test option
- /A+ /A- Word alignment for variables and record fields

With these options the user can define the initial value of the corresponding compiler directives. (See the following subsection for more information.)

▪ /H /NOH	Header in Listing option
▪ /F /NOF	Footer in Listing option
▪ /DA /NODA	Date in Listing option

These options 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-2/86     filename.ext        Date               Page Number
```

The text for the footer line can only be defined in the compiler parameter module.

- /D /NOD Debug option

This option defines whether the reference file (.REF) is generated or not. This file contains the necessary information for the symbolic debugger.

- /8 /2 8086/80286 option

This option affects the code generation. If it is set to 8086 the compiler generates code for the 8086/8088. If it is set for 80286 the compiler generates code for the 80186/80286. The advanced instructions used are ENTER, LEAVE, PUSH immediate, shift/rotate immediate, integer immediate multiply.

- /A+ /A- Alignment option

This option affects the variable and record field allocation. If it is set to A+, all variables except single bytes are allocated on even boundaries. If it is set to A-, no special effort is made to allocate variables on even boundaries. With this option you can choose either to save memory space (A+) or to increase the speed of program execution (A-).

6.9 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 $<Letter><setting>, without any intervening or preceding spaces.

Letter	Definition
S	Stack overflow test (default S+)
R	Subrange and arithmetic overflow test (default R+)
T	Index test (arrays, case) and NIL pointer test (default T+)
A	Word alignment for variables and record fields (default A-)

Setting	Effect
+	Test code is generated or Alignment on
-	No test code is generated or Alignment off
=	Revert to setting before last

Example:

```
MODULE x;(*$T+*)          test code is generated
...
(*$T-*)                   no test code is generated
CASE i OF
...
END
(*$T=*)                   test code is generated (i.e.
...                       the prior value is restored)
END x
```

6.10 Compiler Messages

There are two types of compilation errors:

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

6.10.1 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.

6.10.2 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 the 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 that this message will never occur. 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 information about the source line helps you to 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. You must extend the memory size. You can invoke the compiler by typing 'm2 /f=0 b:comp' to save some memory space. See the chapter on Program Execution for details.

■ `EOF on Control`

We hope that this message will never occur. It is displayed when an internal consistency check of the lister fails. If you get this message, please 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 message '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 section on installation 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 section on installation of this manual.

■ `<file name> halted`

An overlay of the compiler terminated with an unexpected status. This might happen if you stop the compiler by typing <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. You must extend the memory size. You can invoke the compiler by typing 'm2 /f=0 b:comp' to save some memory space. 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 message '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 message 'EOF on Control'.

- `no compilation`

  No compilation takes place because no source file was specified.

- `no file`

  You typed <ESC> when asked to enter a filename, and did not supply any file.

- `not catalogued: <filetype>`

  The compiler had problems closing the file with the given filetype. Make sure that the disks are in the drives.

■ `not deleted: <filetype>`

The compiler had problems deleting the file with the given filetype. 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-2/86 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 section on installation of this manual.

■ `stack overflow`

There is not enough memory to allocate all data areas the compiler needs or to load a compiler overlay. You must extend the memory size. You can invoke the compiler by typing 'm2 /f=0 b:comp' to save some memory space. See the chapter on Program Execution for details.

■ `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 by typing <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.

6.11 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)`

  The identifier table holds 8,000 characters (Base Language System/512k = 30,000). 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 the use of the same identifiers in different scopes will help to avoid this error message.

- `8: too many identifiers (hash table full)`

  The hash table holds 997 identifiers (BLS/512k = 3571). This limits the number of distinct identifiers in one module, including all the identifiers exported by the imported modules.

  The use of the same identifier names in different scopes might help to avoid this error message.

- `205: implementation restriction: procedure too long`

  The code size per procedure is limited to 3500 bytes (BLS/512k = 5000). Splitting the procedure into smaller entities will help.

- `206: implementation restriction: statement table overflow`

  The number of statements per procedure is limited to 700 (BLS/512k = 1000). Splitting the procedure into smaller entities will help.

■ `209: expresson too complicated: jump table overflow`

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

A breakdown of the expression into several temporary expressions is a possible workaround.

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

The linker table holds the fixup information for the linker. The size of this linker table is 600 entries per procedure (BLS/512k = 850). An 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.

This error can be avoided by splitting the procedure into smaller sections or by reducing the frequency of access to imported variables and calls to imported procedures.

6.12 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   :
77   : 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    :
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)
```

```
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  :
209  : expression too complicated: jump table overflow
210  : too many globals, externals and calls (linker table
           overflow)

211  : implementation restriction: code >= 64K bytes

220  : not further specified error
221  : division by zero
222  : index out of range or conversion error
223  : case label defined twice
```

7 THE LINKER

The linker combines all the separately compiled modules into a single load and executable module. It takes the object files (filetype .LNK) of the modules to be linked as input and produces an executable object file (filetype .LOD) and a map file (filetype .MAP).

7.1 How to Use the Linker

The linker is run by typing 'm2 link'. After displaying the banner with the version number, the linker will prompt for the filename of the main module, the 'master file', of your program:

```
A> m2 b:link<CR>

LOGITECH MODULA-2/76 Linker, DOS 8086, Rel. m.n
Copyright (C) 1983, 1984, 1985 LOGITECH
master file >
```

Enter the filename and any options you wish to specify (see the subsection on linker options below). The default drive is the current disk and the default filetype is 'LNK', for modules compiled and ready to link.

If you use the default values of the options, the linker automatically links all other necessary modules. It also lists all imported modules together with their corresponding file names.

Example:

```
A> m2 b:link<CR>

LOGITECH MODULA-2/86 Linker, DOS 8086, Rel. m.n
Copyright (C) 1983, 1984, 1985 LOGITECH
 master file > exampl<CR>
 name of load file :A:exampl.LOD
 linked with:
     Terminal in file: B:Terminal.LNK
     Termbase in file: B:Termbase.LNK
     System in file: B:System.LNK
     Keyboard in file: B:Keyboard.LNK
     ASCII in file: B:ASCII.LNK
     Display in file: B:Display.LNK
 Solving external references
end linkage
A>
```

You can also run the linker by typing the filename on the command line:

```
A> m2 b:link exampl<CR>
```

By default, the linker applies the search strategy explained in the chapter Library Search Strategy. First, it looks for object files on the disk containing the master file. It uses the module name, truncated if necessary, as a filename, and a filetype of 'LNK'. If an object file is not found, the linker issues a message and asks for a file to use (see the 'A' option below). If the 'Q' option is turned on, the linker will ask for every object file in this way.

When the linker asks for an object file, the request is repeated until an appropriate file is found or <ESC> is pressed. Pressing <ESC> means that the file is not available. The linker will stop before writing the .LOD file, but first it will list all the required object files. This allows you to detect any other missing files.

After successful linking, the program is written to a load file with the same name and on the same disk as the master file (filetype .LOD), unless you have specified another filename (see option 'O' below).

If you have installed the fully linked version of the linker (M2L.EXE) you can run it by typing:

```
        A> m2l<CR>
```

or

```
        A> m2l exampl<CR>
```

7.2 Linker Options

The linker can accept several options, entered immediately following the filename. Each option is preceded by '/', and consists of a letter, specifying the switch and a sign '+' or '-', indicating whether the switch is to be turned on or off. If the '+' or '-' is omitted, '+' is assumed.

The linker options are as follows:

- /B Base Layer option (Default B-)

  This option is used when linking a subprogram (overlay). If this option is enabled, the linker will ask for the name of a map file to use in determining which modules will be resident when the current (sub)program is run. Modules which are referenced by the current program, but are not in the map, are linked into the current program. The linker will only list those imported modules that are actually linked into the subprogram.

  If this option is turned off (default), your program is linked without any base layer.

  When entering the file name for the map file of the base layer, an additional option is available:

- /Q Query Resident Modules (Default Q-)

  If this option is turned on the linker will ask whether each resident module should be linked. Type 'y' if it is to be linked and 'n' if this module will already be in memory upon execution.

  If you want to link a subprogram without having the map file for the base layer - for example, if you link a subprogram before you link the base - simply type <ESC> when you are asked to enter its name. In this case, you should also invoke the Query option for the main module (see below). When prompted for the names of the modules to link, you can press <ESC> to indicate that a module will be loaded with the base layer and does not need to be linked into the subprogram.

- /L Large option (Default L-)

  If this option is turned off, up to 200 modules can be linked. If it is turned on, up to 400 modules can be linked.

- /M Map File option (Default M-)

  If this option is turned on, the linker generates the map file. A map file is needed if this program will be the base for other programs - if it has overlays.

  It is recommended that before executing your program you consult your base map to verify that your modules will be initialized in the correct order. The order of initialization is the order of appearance in the map.

- /O Output File Query option (Default O-)

  If this option is turned on, the system prompts the user to give the name of the .LOD file. No options may be specified with this filename!

- /Q Query option (Default Q-)

  If turned off (default) the linker will search, according to the default search strategy, for each link file using a default name - the module name truncated to the maximum length of a file name with the filetype 'LNK'. If the Automatic Query option (see below) is turned on, the linker enters a temporary query mode whenever a link file is not found. In this case, the linker prompts for the file name.

  If the Query option is turned on you are asked to enter the name of each link file (query search strategy). You can specify the default name by pressing <CR>.

  If a file is not available and will not be in the base at execution time, your program cannot be linked. In this case, press <ESC> to stop the linker. Note that the linker will not stop until it has checked the list of all imported modules.

  If you are linking a subprogram and you specified the Query option together with the Base Layer option, but you did not specify any base file, you can use <ESC> to indicate that a module will be loaded with the base layer and does not need to be linked into the subprogram.

- /A Automatic Query option (Default A+)

  If turned on (default) the linker enters a temporary query mode when a link file is not found. Turning off the Automatic Query option prevents this automatic query mode and is useful in command files.

7.3 Linker Messages

During execution of the linker certain errors may occur. The following error messages may appear:

- `cannot load ...`

  There is not enough memory to allocate all the data area needed by the linker. You must extend the memory size. You can invoke the linker by typing 'm2 /f=0 b:link' to save same memory space. See the chapter on Program Execution for details.

- `checksum error`

  The linker computes a checksum. This message indicates that the computed checksum does not match the value found in the link file. The link file seems to be incorrect.

- `invalid version`

  Make sure that the linker and the compiler are for the same target system.

- `error in format of map file`

  When linking a subprogram (Base Layer option), the linker checks the syntax of the map file of the base layer. Make sure the map file is correct and that it was generated by a compatible linker.

- `error in link-file format`

  The syntax of the link file is not correct. The linker is checking on the correct order of the different link file records. Make sure that you are using the right linker and compiler. Also check if the link file is complete.

- `error in optionlist <default values> <option typed>`

  The option entered is not available. Refer to the description of the possible options above.

- `file not found`

  The link or map file was not found. If the Automatic Query option is turned on, the linker requests the file name, again. You should either type the correct file name or enter <ESC> if the file is not available.

- `not enough memory to link`

  There is not enough memory to allocate all the data area needed by the linker. You must extend the memory size. You can invoke the linker by typing 'm2 /f=0 b:link' to save some memory space. See the chapter on Program Execution for details.

- `linkage aborted: more than 6000 procedures`
- `linkage aborted: more than 200/400 separate modules`
- `linkage aborted: more than 2000/4000 modules under process`

These error messages appear when some internal table of the linker is too small for the program to be linked. The number of separate modules and of imported modules under process depends on the Large option: L- = 200, 2000 L+ = 400, 4000.

- `error in linkage`

The linkage did not terminate successfully. No load file was produced.

- `linkage failed (please report to LOGITECH)`

This message appears when the linker detects some internal inconsistency. If you get this error please contact LOGITECH with a copy of the program which caused the error, and a copy of the dump produced (MEMORY.PMD).

- `link files missing`

The linker cannot link successfully because one or more link files are missing.

- `module not found in this file`

The module name found in the link file does not correspond to the name of the module to be linked.

- `output disk full`

Insufficient disk space to write the load or map file causes the linker to stop execution. To solve the problem, delete superfluous files or copy some files to the disk in the other drive or turn off generation of the map file (see Map option above). The output files are written to the disk where the main module is found.

- `program exceeds memory size (1MB)`

The size of a linked program is limited to one megabyte.

- `version conflict in module: <module name1> imported by <module name2>`

  The module key (time stamp) of the module 'namel' is different from the module key expected by module 'name2'. To solve this problem, recompile the necessary modules in the right order.

8 PROGRAM EXECUTION

To run a MODULA-2/86 program you can either use the program 'M2' (described in section 8.1), or you can use the utility LOD2EXE to generate a standard .EXE file (described in section 8.2).

8.1 Running MODULA-2/86 with M2

To run a MODULA-2/86 program you must invoke the program 'M2' which is supplied with the MODULA-2/86 system. You should copy this program to the diskette or directory which you use to develop your Modula-2 programs or to the directory where you generally keep executable programs. (See the section on 'Installation' for details.) The program 'M2' is the MODULA-2/86 run-time support (RTS) and has the ability to load and execute MODULA-2/86 programs. The default extension for executable MODULA-2/86 programs is 'LOD'.

To run a MODULA-2/86 program enter 'M2 <programname>'. The program name may also be preceded by a drive and/or a directory name, in which case the program is loaded from the specified drive or directory. If a pure file name is entered, the MODULA-2/86 program will be searched in the current directory on the current drive, and if not found, in the directory '\m2lod' on the current drive.

The following is a brief example:

```
C>m2 exampl<CR>
The program worked! (Hit a key)
C>
```

The program 'exampl' has been loaded from the current drive C. It may have been found in the current directory or in the directory '\m2lod'. If the program exists in both directories, the version from the current directory is executed.

8.1.1 Memory Allocation Options

You can specify the amount of memory allocated for the program you execute with options for the run-time support. If no option is specified, (as in the above example), the whole memory except the space used by 'command.com' is given to the application program. There are two options from which you can choose: one to specify the memory used by the application, the other to specify the memory left free for DOS and other applications.

Only one option can be specified at a time. The option, either 'u' or 'f' must come after the name of the run-time support (usually m2), be preceded by a slash, and be followed by an equal sign and the amount of memory in kilobytes (KB), as follows:

```
C> m2/u=xxx exampl
```

where 'u' stands for used memory option. It specifies the memory used by the application, including its code and data.

The other option is indicated as:

```
C> m2/f=yyy exampl
```

where 'f' stands for free memory option. It specifies the amount of memory that must remain available for DOS and other applications.

The following figure shows the memory organization and the meaning of the parameters.

```
 ----------------------------------  000H
|      Interrupt vector table      |
 ----------------------------------  400H
|         Operating system         |
 ----------------------------------
| Code and data of the MODULA-2/86 |
|         run-time support         |
 ----------------------------------+-------+----------
| Code of the MODULA-2/86 program  |       |
|                                  |       |
 ----------------------------------        |
|  Global data of the MODULA-2/86  |       |
|             program              |       |  u option
 ----------------------------------        |
|       Heap of main process       |       |
 ----------------------------------|       |
|                                  |       |
 ----------------------------------        |
|      Stack of main process       |       |
 ----------------------------------  ------+----------
|                                  |       |
 ----------------------------------        |  f option
|           Command.COM            |       |
 ----------------------------------  TOP OF MEMORY----
```

FIGURE 8-1
MEMORY ORGANIZATION AND ALLOCATION PARAMETERS

If the memory space which remains for the application is not enough to load the program, the run-time support tells you that it cannot load your program. If the space is just enough to load the program, but not sufficient to set up a correct workspace (stack and heap), the program terminates with a stack overflow.

8.2 The File Conversion Utility LOD2EXE

The utility program 'LOD2EXE' converts a .LOD file into a standard MS-DOS .EXE file.

The program combines a special version of the MODULA-2/86 run-time support with the .LOD file of the application program into a .EXE file. The adapted RTS version is called 'L2ERTS.L2E' and must be in the current directory. The application program must first be linked with the LOGITECH MODULA-2/86 linker, without the /B+ option for a base file. The 'LOD2EXE' program prompts for an input file to convert. The default extension is .LOD. The name and the pathname of the output file are the same as those of the input file. The extension is .EXE.

The LOD2EXE utility itself is provided only in the .LOD version. However, you can use this .LOD version to produce a .EXE version -- LOD2EXE.EXE.

8.2.1 Memory Allocation Options

You invoke the LOD2EXE utility by typing:

```
A> m2 lod2exe<CR>
```

All available memory, less space for 'command.com' will be allocated when the program is executed.

There are three other ways you can specify how much memory you want to allocate for your application. The option is entered after the filename of the program to convert, separated by a slash. The form of the option is a character ('u', 'f', or 'w') followed by an '=' sign and the amount of memory in kilobytes (KB). If only the character is specified, the LOD2EXE utility will prompt you for the number. Only one option may be specified. If you enter a sequence of options, the last entry will be used. The three options are as follows:

```
A> m2 lod2exe exampl/u=xxx
```

Used memory option: specifies the memory that is used by the application, including code and data.

```
A> m2 lod2exe exampl/f=yyy
```

Free memory option: specifies the amount of memory that must remain available for DOS and other applications.

A> m2 lod2exe exampl/w=kkk

Work space option: defines the size of the workspace (heap and stack) for the application.

The following figure shows the memory organization and the meaning of the parameters:

```
 ----------------------------------  000H
|      Interrupt vector table      |
 ----------------------------------  400H
|         Operating system         |
 ----------------------------------
| Code and data of the MODULA-2/86 |
|         run-time support         |
 ----------------------------------+-------------------+
| Code of the MODULA-2/86 program  |                   |
|                                  |                   |
 ----------------------------------                    |
|  Global data of the MODULA-2/86  |                   |
|             program              |              u    |
 ----------------------------------  -------+  option  |
|       Heap of main process       |        |          |
 ----------------------------------|    w   |          |
|                                  |option  |          |
 ----------------------------------         |          |
|       Stack of main process      |        |          |
 ----------------------------------  -------+----------+
|                                  |    f   |
 ----------------------------------  option |
|           Command.COM            |        |
 ----------------------------------  TOP OF MEMORY----
```

FIGURE 8-2
MEMORY ORGANIZATION AND ALLOCATION PARAMETERS

8.2.2 Example Dialog

```
A> m2 lod2exe exampl<CR>

LOGITECH MODULA-2/86 LOD2EXE, DOS 8086, Rel. m.n.
Copyright (C) 1983, 1984, 1985 LOGITECH.
Convert   from :    exampl.LOD
          to   :    E:exampl.EXE
          using:    L2ERTS.L2E

no option specified

code (size) data (size) ----key----- module
0238 (005A) 0313 (0001) AB3E02CAC724 Exampl
023E (0219) 0314 (0002) A91104323CC8 Terminal
0260 (03BB) 0315 (0069) A91104381612 Termbase
029C (007F) 031C (0000) A911043455B4 Display
02A4 (0314) 031C (0088) A9110433A5D2 Keyboard
02D6 (0011) 0325 (0000) A9110434E3EE ASCII
02D8 (03B0) 0325 (0104) A91104246A54 System
pass2 ......
A>
```

The values of the table written on the screen are in hexadecimal. In case of 'code' and 'data' they constitute the base address in paragraphs. The size is given as a number of bytes. The key is the representation of the three word time stamp generated for each module at compile time (module key).

8.3 Aborting MODULA-2/86 Programs

When you type <Ctrl-Break> or <Ctrl-C>, the operating system usually aborts the program currently running. <Ctrl-C> and <Ctrl-Break> have the same effect in MODULA-2/86. 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 the user enters 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 running in an infinite loop. However, under certain circumstances, the whole system might crash if <Ctrl-Break> was accepted. MODULA-2/86 tries to prevent this from happening. Therefore, typing <Ctrl-Break> will sometimes have no effect at all.

In MODULA-2/86, the library module 'Break' allows the user to define how a program will behave when <Ctrl-Break> or <Ctrl-C> is typed. If module 'Break' is linked into a program, a memory dump (file MEMORY.PMD) will be generated when the user types <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, module 'Break' must be imported explicitly into one of the modules that constitute the program. Normally, one imports it in the main module of the program. If module 'Break' is not linked with a program, no memory dump will be generated when the program is terminated by typing <Ctrl-Break> or <Ctrl-C>.

With module 'Break' it is also possible to disable aborting the program - to ignore <Ctrl-Break> and <Ctrl-C>. In addition, it is possible to install a break-procedure which will be called when the user types <Ctrl-Break> or <Ctrl-C>. With a break-procedure a dump will not be generated automatically. When module 'Break' is used, typing <Ctrl-Break> once, in almost all cases, will stop the program or call the installed break-procedure.

When debugging a program with the LOGITECH MODULA-2/86 run-time debugger, one should link module 'Break' with the program.

8.4 Command Line Arguments

When a MODULA-2/86 program is executed using the command 'm2 <programname>', any text which follows the name of the program is taken as keyboard input. This means, for example, that it works to say 'm2 comp my prog/batch/noaquery'. Note that this works for any MODULA-2/86 program that does keyboard input using the modules Terminal or InOut.

This allows MODULA-2/86 programs to be used more easily with the DOS Batch facility, which requires that all input to a program be on the command line. Because the compiler, linker and debugger accept a space (as well as <CR>) to terminate an argument, multiple arguments may be given on the command line. For example:

```
m2 link overlay1/b mainline
```

9 THE SYMBOLIC POST-MORTEM DEBUGGER

9.1 Introduction

The post-mortem debugger is an instrument used to inspect a crashed program to determine what went wrong with the program and where the problem occurred. When a program terminates abnormally, the memory image is saved on disk in a dump file. The post-mortem debugger allows the user to inspect this dump file symbolically. The debugger displays the state of the program using the corresponding module and procedure names. It shows the names and values of variables according to their type structure.

The MODULA-2/86 run-time support creates a memory dump file, named MEMORY.PMD, when:

- a run-time error occurs.
- a program calls the standard procedure 'HALT'.
- a program calls procedure 'Terminate', exported by module 'System', with a status 'asserted'.
- the user types <Ctrl-Break> or <Ctrl-C> while running a program that imports module 'Break'.

In addition to the dump file (PMD filetype), the post-mortem debugger uses reference files (REF filetype) and sometimes, source files (MOD and DEF filetype) of the component modules of the stopped program. The compiler generates the reference file when it compiles the implementation of a module. The reference file contains symbolic information and the addresses of all the variables, procedures, and statements in a module.

The symbolic post-mortem debugger can be used even if some, or all, reference files are missing. However, without reference files it is impossible to symbolically examine the corresponding modules. Without source files the debugger cannot show the text of the corresponding modules in the program.

The sample screens used for illustration refer to the sample program included at the end of this chapter.

9.1.1 Installation

There are two ways to install the MODULA-2 post-mortem debugger depending on whether you have two floppy disk drives or one floppy and a hard disk. In both cases we assume you have successfully completed the installation process as described in the Installation chapter.

Two Floppy Disk Drives

If your system uses two floppy disk drives, prepare a copy of the MODULA-2 post-mortem debugger disk. Use this disk when running the debugger on .PMD files. Your debugger disk contains:

- the MODULA-2 post-mortem debugger files (.LOD).
- the reference files (.REF) of the library modules.

One Floppy and a Hard Disk

If your system is equipped with a hard disk and one floppy, copy all .LOD files in the directory where you keep MODULA-2/86 executable files (usually in '\m2lod'). Copy all the .REF files in the directory where you keep .REF files (usually in '\m2lib\ref').

Before you run the post-mortem debugger make sure that the PATH and 'M2xxx' environment variables are correctly set.

9.1.2 Windows

The post-mortem debugger displays five different types of information about your program. This information is divided into windows which are invoked using global commands. Only one window can be displayed at a time. This section gives an overview of the different windows.

- Call Window

  The Call window displays the chain of procedure calls of a process. All Modula-2 programs contain at least one process - the main program itself. When you initiate the post-mortem debugger, the debugger displays the Call window which details the call chain of the process that was running when the program stopped. The first procedure of the call chain is the procedure in which the program stopped. You can select a procedure to examine its text or data, (i.e. local data).

If the program which was stopped contains more than one process, the Call window can show the call chain of each process in the program. (For further information see the Data Window or Raw Window section).

- Module Window

  The Module window displays the list of modules that constitute the program being debugged. You can select a module to display its text or data (i.e. global data).

- Data Window

  The Data window displays the data of the procedure or module that you selected in the Call or Module window. You can browse into the data by selecting a data item to display substructures or by selecting a PROCESS variable to display another call chain.

- Text Window

  The Text window displays the text of the procedure or module selected in the Call or Module window. If you choose to inspect the procedure in which the program stopped, a greater-than sign indicates the line that was being executed when the program stopped. If you inspect any other procedure, the greater-than sign indicates the line containing the call to the next procedure in the call chain.

  If you select a module, the Text window shows the end of that module for the lines displayed previously. This window can also display the corresponding definition module, if one exists.

- Raw Window

  The Raw window displays the memory image, recorded when the program stopped. The debugger shows the memory contents in hexadecimal format (as bytes, words or addresses), in decimal format (as integers, cardinals or reals), or in text format (as characters). You select the information at a particular address for display. You also can select a process descriptor at a specific address to build its call chain.

9.2 How to Run the Post-Mortem Debugger

To initiate the post-mortem debugger, enter:

```
A> m2 b:pmd<CR>
```

The debugger responds with a sign-on message:

```
LOGITECH MODULA-2/86 Post-Mortem Debugger, DOS 8086
```

followed by the release number and date and a copyright notice. Then, the debugger asks for the name of your dump file. The default filename is MEMORY.PMD (type <CR> to indicate the default filename).

After you enter the name of the dump file, the debugger executes some internal initialization and then displays the Call window. When the debugger displays symbolic information, it needs to have access to the corresponding reference files. The debugger automatically searches for reference and source files according to the default search strategy. If a file cannot be found, the debugger will ask you to type in the correct file name. Enter <ESC> to indicate that the file is not available.

Some modules are defined as a part of the System Library. If the corresponding files are not found with the default search strategy, the post-mortem debugger does not prompt for their filenames. The system library includes 'ASCII', 'DiskDirectory', 'DiskFiles', 'Display', 'FileSystem', 'Keyboard', 'Program', 'Reals', 'Storage', 'System', 'TermBase', and 'Terminal'.

When entering the name of the dump file, you can also specify the Query option, by following the filename with '/q'. When the Query option is turned on, the user is prompted to enter the filename of the reference and source files. If the user types <CR>, the system uses the default path and the default name. If the user types the name only, the system uses the default path (see the section on query search strategy). If the user types <ESC>, the system considers the file absent.

Another option you can specify is the Version option by following the filename with '/v'. The Version option displays the date and version of the run-time debugger.

The run-time support (M2.EXE) reserves enough K bytes of memory for the DOS command interpreter (COMMAND.COM) on top of memory. This area of memory can be used to load the application program. The only disadvantage of this is that DOS will need to load from disk its command interpreter each time the debugger terminates.

To run the debugger using all the available memory, type:

```
        A> m2\f=0 pmd<CR>
```

Refer to the Program Execution chapter for more details.

9.3 Window Format

Only one of the five windows is displayed at a time. The first two lines of each window indicate the commands available for the particular window. The first, upper case letter of each command is the command character. You may type a new command without waiting for the completion of the previous one. The debugger then interrupts the display in progress and starts the new command.

9.3.1 Selecting an Item for Display

In each window, one of the displayed items is considered selected. The post-mortem debugger displays the position of the selected item as a line number in the lower part of each window. In the Raw window the debugger displays the address of the selected memory location.

You can select a different item using the cursor keys. You can also type in a new line number to select the item at that position.

9.3.2 Execution Markers

In the Call, the Module and the Text windows certain lines are marked with a greater-than sign (>). The greater-than sign is used as an Execution Marker to indicate active code.

In the Call window all the procedures are marked because they were all active when the program stopped. In the Module window, only the modules to which these active procedures belong are marked. In the Text window, when the debugger displays the text belonging to an active procedure, it indicates with a greater-than sign (>) the line that was being executed when the program stopped.

9.3.3 Numbers and Addresses

In general, the post-mortem debugger displays hexadecimal values followed by an "h", and octal values followed by a "c". The only exceptions are the values displayed in the Raw window, where the header line indicates the format of the window contents.

The post-mortem debugger shows addresses in the format <segment>:<offset> with both parts in hexadecimal representation.

When the address is related to code, the segment corresponds to the beginning of the module code. When the address is related to module data, the segment indicates the beginning of the module data. In both cases, the offset is relative to the beginning of the segment.

9.4 The Post-Mortem Debugger Commands

The post-mortem debugger has two main types of commands - global commands and local commands. The user has access to the global commands in all the windows and can use them whenever the post-mortem debugger is ready to accept commands. Local commands are only applicable to the particular window in which they appear and are explained in the appropriate sections. Many of the global and local commands operate on the selected item in the window shown when the command is typed.

9.4.1 Global Commands

The global commands appear on the second line of each window, below the window name and local commands, as follows:

```
=[Call Mod Data Txt Raw Ini] Hex Quit #[P L N H]
```

You invoke a global command by entering the first letter of the command name, shown in upper case on the command line. When invoking a command in the first set of brackets, you must preceed each command with an equal sign (=). When invoking a command in the second set of brackets, you must preceed each command with a pound sign (#).

For example, to invoke the Module window, enter:

=M

To change the page mode, enter:

#P

The following global commands are available:

- =Call

  Invokes the Call window.

- =Module

  Invokes the Module window.

- =Data

  Invokes the Data window.

- =Text

  Invokes the Text window.

- =Raw

  Invokes the Raw window.

- =Init

  Switches to the Call window and redisplays the process that was running when the program stopped.

- Hexadecimal

  Converts a number from decimal to hexadecimal. Enter a decimal number when prompted and the debugger will display the hexadecimal value. The decimal number cannot be greater than 65535 (decimal).

- Quit

  Quits the debugger. You are prompted for a veto. Enter Yes to terminate the debugger session.

- #Page

  Turns the page mode on and off. If the page mode is on the debugger displays information one screen at a time. If the page mode is off, the debugger scrolls information off the top of the screen and adds new lines at the bottom. The default is page mode off.

- #Help

  Turns the help mode on and off. If the help mode is on, the two help lines are displayed on the top of the screen. The default is help mode on. This can be changed with the option /H- when you start the PMD.

- #Line

  Specifies the number of lines the debugger displays at one time. The user can invoke this command in scroll or page mode. The default value is 16.

- #Noise

  Turns the bell that signals erroneous input on and off. The default is on.

- <Ctrl-X>

  Erases all characters typed, without leaving the command, when a command requires additional input - for example, the Hexadecimal command.

- <Esc>

  Cancels the command, when a command consists of several keystrokes - for example the commands to select another window - or when a command requires additional input - for example, the Hexadecimal command.

9.5 Post-Mortem Debugger Windows

The following sections explain each of the five windows, their components and primary functions.

9.5.1 Call Window

The Call window displays the chain of procedure calls of a process. When you initiate the post-mortem debugger, the debugger displays the Call window which details the call chain of the process that was running when the program was stopped.

The line number displayed with the first procedure indicates the line in the source text where the program was stopped. The line number displayed with the second procedure indicates the line in the source text where the first procedure was called. After the line number, a statement number indicates which statement is executed in the line.

When analyzing a process that was not actively running when the program stopped, the first procedure displayed in the Call window shows where this process was suspended. The line number indicates where the last 'TRANSFER' or the interrupt occurred.

The displayed call chain is limited to 32 procedures. You can select a procedure to examine its data or text.

The following example shows the Call window for the sample program 'Demo', as the debugger will initially display it when debugging the corresponding memory dump. A listing of the sample program 'Demo' is included at the end of this chapter. The 'Status: halted' indicates that a 'HALT' statement was executed in procedure 'LastOne'. This occurred in the second statement on line 48 in module 'Demo'.

```
CALL |
=[Call Mod Data Txt Raw Ini] Hex Quit #[P L N H]
------------------------------------------------------------
Status: halted

 1 > LastOne          in Demo stops at line 48, statement 2
 2 > RecursiveOne     in Demo       at line 37, statement 2
 3 > RecursiveOne     in Demo       at line 38, statement 1
 4 > FirstOne         in Demo       at line 24, statement 1
 5 > initialization   of Demo       at line 57, statement 1
 6 > PROCESS

  Position > 1
```

SAMPLE SCREEN 9-1

If a reference file of a module is missing, the procedure names are replaced by procedure numbers within the module. The body of the main module is number 0. The procedures declared in the definition module are numbered starting from 1 according to their declaration sequence. The procedures that appear only in the implementation module get the numbers in the order of their declaration (header). Local modules are handled as procedures.

The last item in the call chain, shown as 'PROCESS', refers to the process currently being examined. You may select this item and invoke the Data window to display the contents of the process descriptor. The process descriptor is a set of data associated with a variable of type 'PROCESS'. It contains information describing the current status of the process; for example, the 8086/88 registers.

There are no local commands in the Call window.

The typical use of the Call window is to select a procedure and invoke the Data or Text window.

9.5.2 Module Window

The Module window displays the list of modules that constitute the program being debugged. You can select a module in order to display its data or text. The modules that contain code that was active when the program stopped are marked by a greater-than sign (>).

The Module window also displays the addresses of the code and data of each module. The number of modules displayed cannot exceed 255 and the PMD cannot debug a program with more modules.

The following example shows the Module window as the debugger displays it, for instance when the Module command is used in the situation shown in Sample Screen 1. The Module window displays the names and addresses of the modules in the program being debugged. The module named 'Demo' is marked with a greater-than sign, indicating that some of its code was active when the program stopped.

```
MODULE | FIND
=[Call Mod Data Txt Raw Ini] Hex Quit #[P L N H]
---------------------------------------------------------
     Module Name              CodeAddress        DataAddress

 1   System                   0C8C:0000h          0CE6:0000h
 2   ASCII                    0C8A:0000h          0CE6:0000h
 3   Keyboard                 0C59:0000h          0CDD:0000h
 4   Display                  0C51:0000h          0CDD:0000h
 5   Termbase                 0C0B:0000h          0CD6:0000h
 6   Terminal                 0BE8:0000h          0CD5:0000h
 7 > Demo                     0BC5:0000h          0CD1:0000h

 Position > 7
```

SAMPLE SCREEN 9-2

Local Commands in the Module Window

There is one local command in the Module window, the Find command.

- Find

  This command prompts the user for a module name (case sensitive) and, if it is found, the PMD sets the selected position to this module.

The typical use of the Module window is to select a module and invoke the Data or Text window.

9.5.3 Data Window

The Data window displays the variables and/or parameters of the selected procedure or module. On the first line below the global commands it shows the name of the procedure or module being examined. It indicates the current values of each variable and the type of the variable. The Data window also allows you to view the contents of structured variables such as arrays or records. You can select a data item to display substructures.

Local Commands in the Data Window

There are seven local commands in the Data window. The user invokes the local commands by entering the first character, shown in upper case, of each command name. (The Exchange command is an exception. It is denoted by 'X'.) The local commands appear on the first line of the window, to the right of the window name as follows:

```
DATA | Son Father Left(dec index) Right(inc) Var X Addr
                                              Examine(process)
```

- Son

  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'.

If the selected item is of hidden type, the debugger will show the content of the variable as it is defined in the corresponding module. Note that with modules compiled with a previous compiler (before 2.0), the debugger will prompt the user for the module name.

- Father

  Displays the data structure above the current level. The Father command is the reverse of the Son command.

- Variables

  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.

- 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.

  Example:

```
TYPE Point = RECORD
                 x,y: INTEGER;
                 END;
VAR Figure = ARRAY [1..100] OF Point;
```

  If the selected item is *Figure [17].x* then command Right will select *Figure [18].x.*

- eXchange

  The Exchange command switches the debugger screens between displays of module and procedure data (i.e. global/local data). If you are currently examining the data of the selected procedure, the Exchange command will show the data of the corresponding module and vice-versa.

- Address

  Displays the address of the selected data item.

- Examine

  This command can be used when you select a variable of type 'PROCESS'. Otherwise, the debugger prompts you to introduce the address of the process descriptor - the content of a variable of type PROCESS. The Examine command switches to the Call window and displays the call chain of the process to be examined. You may enter the global command Init to show the Call window of the process that was running when the program stopped.

The following example shows the data for module 'Demo' as displayed by the debugger when the Data command is used in the situation shown in Sample Screen 2, except that, in addition, the position has been changed to '4'. The first column lists the four variables in module 'Demo', and the second and third columns indicate the current value and type of each variable. For example, the first variable is 'x'. It has a current value of '1' and is of type 'INTEGER'.

```
DATA | Son Father Left(dec index) Right(inc) Var X
                     Addr Examine(process)
=[Call Mod Data Txt Raw Ini] Hex Quit #[P L N H]
-----------------------------------------------------------
Demo.

 1  x                               1     INTEGER
 2  y                 2.0000000000E+000     REAL
 3  z                               3     INTEGER
 4  node                            ARRAY[1..4] OF RECORD

 Position > 4
```

SAMPLE SCREEN 9-3

If you choose to examine the contents of structured data, the Data window indicates the complete 'path'. This path is shown on the first line below the global commands. It includes the name of the procedure or module, the name of the variable, the names of the record fields, and the indices in the case of arrays. There is no limitation on the length of this path. If the path contains a list, the { } contain the number of the list element. For example,

```
MODULE Sample
 TYPE T=RECORD
              i:CARDINAL;
              next: POINTER TO T;
             END;
 VAR
           p:T;
 END Sample;
```

If you examine the 3rd list element of 'Sample' the debugger writes:

```
Sample.p.next {3}
```

which is equivalent to the Modula-2 expression, Sample.p.next^.next^.next.

The following example shows the content of the variable 'node' of module 'Demo'. This screen illustrates the effect of the Son command when used in the situation shown in Sample Screen 3. The path is indicated on the first line below the global commands as 'Demo.node'. Because 'node' is an array of records, the numbers in brackets show the indices into the array. The next columns usually show the content and type of each array element. In the case of structured data, as indicated by 'RECORD DATA' in the example, the Son command must be used to display the values.

```
DATA | Son Father Left(dec index) Right(inc) Var X
                     Addr Examine(process)
=[Call Mod Data Txt Raw Ini] Hex Quit #[P L N H]
------------------------------------------------------------
Demo.node

 1  [1]                                RECORD              DATA
 2  [2]                                RECORD              DATA
 3  [3]                                RECORD              DATA
 4  [4]                                RECORD              DATA

 Position > 1
```

SAMPLE SCREEN 9-4

If the Son command is used in the situation shown in Sample Screen 4, the result will be as illustrated by the following example. It shows the three fields of the record 'node[1]', the value of each field and its type. For example, the field 'data1' has a value of '1' and is of type 'CARDINAL'.

```
DATA | Son Father Left(dec index) Right(inc) Var X
                     Addr Examine(process)
=[Call Mod Data Txt Raw Ini] Hex Quit #[P L N H]
------------------------------------------------------------
Demo.node[1]

 1  data1                            1    CARDINAL
 2  data2          2.0000000000E+000    REAL
 3  data3                               INTEGER

 Position > 1
```

SAMPLE SCREEN 9-5

To examine another element of the array, you invoke the local command Right. When applied in the situation shown in Sample Screen 5, this moves the position to the second element of the array, and the Data window displays the fields of 'Demo.node[2]'. This situation is shown in the following example.

```
DATA | Son Father Left(dec index) Right(inc) Var X
                         Addr Examine(process)
=[Call Mod Data Txt Raw Ini] Hex Quit #[P L N H]
-------------------------------------------------------------
Demo.node[2]

 1  data1                             2   CARDINAL
 2  data2              3.0000000000E+000   REAL
 3  data3                             4   INTEGER

 Position > 1
```

SAMPLE SCREEN 9-6

9.5.4 Text Window

The Text window shows the text of the selected module or procedure. The greater-than sign (>) indicates the line in which the program was stopped, the call of the next procedure, or the last process transfer or interrupt. The Text window shows the module name on the line below the global commands. It lists the source text with a line number within this module.

Local Commands in the Text Window

There are three local commands in the Text window. You invoke the commands by entering the first character, shown in upper case, of each command name. The local commands appear on the first line of the window, to the right of the window name as follows:

```
TEXT | Address Def Imp Find
```

- Address

  Shows the code address of the selected source line.

- Definition

  Displays the corresponding definition module (DEF filetype), if an implementation module is displayed.

- Implementation

  Redisplays the corresponding implementation module (MOD filetype) if the user is currently inspecting the definition module.

- Find

  Searches for a procedure. The user is prompted to enter the porcedure name. If more than one procedure has the same name (nested procedures), the Find command searches first for the one which is less nested.

The following example shows the source text of module 'Demo', lines 43 through 58. In this case the greater-than sign indicates that the program stopped on line 48.

```
TEXT | Address Def Imp Find =[Call Mod Data Txt Raw Ini] Hex
Quit #[P L N H]
-----------------------------------------------------------
Demo
 43         BEGIN
 44              node[4].data1 := x;
 45              node[4].data2 := y;
 46              node[4].data3 := z;
 47              INC (x);
 48 >            y := y + 1.0; HALT; INC (z)
 49         END LastOne;
 50
 51       BEGIN        (** MAIN **)
 52           WriteString ('Here we go ...');
 53           WriteLn;
 54           x := 1;
 55           y := 2.0;
 56           z := 3;
 57 >         FirstOne (x, y, z)
 58       END Demo.

   Position > 48
```

SAMPLE SCREEN 9-7

9.5.5 Raw Window

The Raw window displays the memory contents around a given address. The Raw window can show the memory contents in several different formats. The local commands for this window list the user's choice of formats.

The initial address of the selected memory location, depends on the window from which you invoke the Raw window. The values are set as follows:

- When initiated from the Call window, the Raw window sets the selected address to the beginning of the local variables of the selected procedure. Note: This corresponds to the value of the 'base pointer' register.
- When initiated from the Module window, the Raw window sets the selected address to the beginning of the global variables of the selected module. Note: This corresponds to the beginning of the module's 'data segment'.
- When initiated from the Data and Text windows, the selected address corresponds to the address of the selected variable or line (statement). If a definition module is displayed in the Text window the selected address corresponds to the last selected line (statement) in the corresponding implementation module.

You can introduce a new address which will then be selected. You enter addresses in the form <segment>:<offset>. Both parts are 4 digit, hexadecimal numbers. In the Raw window it is also possible to enter the offset only. In this case, the 'segment' of the selected address will not be changed.

Local Commands in the Raw Window

There are ten local commands in the Raw window. Eight of the local commands indicate the different data formats available. You invoke the first two commands by entering the first character, shown in upper case, of each command name. The commands that switch between different display formats must be preceeded by a pound sign (#).

The local commands appear on the first line of the window, to the right of the window name as follows:

```
RAW | Examine Son #[Byte Word Addr Char Text Int Unsigned
                                                     Real]
```

- Examine

Assumes the memory content at the selected address is of type 'PROCESS' - a pointer to a process descriptor. The Examine command displays the Call window of this process. The global command Init can be used to show the Call window of the process that was running when the program stopped.

- Son

  Takes the contents of the selected memory location as the new selected address. You typically enter this command to follow a linked list.

The representation of data is chosen by the following commands:

- #Byte

  BYTE (hexadecimal) format.

- #Word

  WORD (hexadecimal) format. This is the default format.

- #Address

  ADDRESS (hexadecimal) format.

- #Char

  CHAR (octal) format. In CHAR format, non-printable characters are displayed as octal numbers.

- #Text

  Text format. In Text format, non printable characters are displayed as dots.

- #Integer

  INTEGER format.

- #Unsigned

  Unsigned CARDINAL format.

- #Real

  REAL format.

The selected address in Sample Screen 8 is 0D00:0024. Using the Son command, you can select the memory contents at this address to become the new address to display. An address is expressed as a 4 digit segment followed by a 4 digit offset separated by a colon. The segment is 0CF7 and the offset is 005C; hence, the new address is 0CF7:005C. Sample Screen 9 shows the Raw window with 0CF7:005C as the new address. Both screens are represented in ADDRESS format, hexadecimal.

In Sample Screen 8, the window displays the statement, 'out of dump'. This indicates this part of the memory is not contained in the memory dump because the program did not use it.

```
RAW | Examine Son #[Byte Word Addr Char Text Int Unsign
                                                  Real]
=[Call Mod Data Txt Raw Ini] Hex Quit #[P L N H]
------------------------------------------------------------
address (hexadecimal) ( <segment>:<offset> )
                    0            4            8            C

0D00:0000   0000:0079    0000:0000    0000:0000    0000:0000
0D00:0010   A911:0000    55B4:0434    0C51:0002    0001:0CDD
0D00:0020   0CF7:00B8    0CF7:005C    6279:654B    6472:616F
0D00:0030   0000:0000    0000:0000    0000:0000    0000:0000
0D00:0040   0433:A911    0002:A5D2    0CDD:0C59    00E6:0001
0D00:0050   008A:0CF7    5341:0CF7    0049:4943    0000:0000
0D00:0060   0000:0000    0000:0000    0000:0000    A911:0000
0D00:0070   E3EE:0434    0C8A:0002    0001:0CE6    0CF7:0114
0D00:0080   0CF7:00B8    7473:7953    0000:6D65    0000:0000
0D00:0090   0000:0000    0000:0000    0000:0000    0424:A911
0D00:00A0   0002:6A54    0CE6:0C8C    000F:0001    00E6:FFFF
0D00:00B0   4E8A:0CF7    61F9:82FA    F982:087C    8003:7F7A
0D00:00C0   4E88:5FE1    FA46:8AFA    2076:203C    3CFA:468A
0D00:00D0   8A19:737F    8E2E:FA46    8B00:001E    7100:5A0E
0D00:00E0  -- out of dump
0D00:00F0  -- out of dump

 Position > 0D00:0024
```

SAMPLE SCREEN 9-8

```
RAW | Examine Son #[Byte Word Addr Char Text Int Unsigned
                                                  Real]
=[Call Mod Data Txt Raw Ini] Hex Quit #[P L N H]
------------------------------------------------------------
address (hexadecimal) ( <segment>:<offset> )
                        0            4            8             C
0CF7:0000   6F6D:6544    0000:0000    0000:0000    0000:0000
0CF7:0010   0000:0000    0000:0000    0385:A951    0031:7F58
0CF7:0020   0CD1:0BC5    002E:0001    FFFF:0CF7    6554:FFFF
0CF7:0030   6E69:6D72    0000:6C61    0000:0000    0000:0000
0CF7:0040   0000:0000    A911:0000    3CC8:0432    0BE8:0002
0CF7:0050   0001:0CD5    0CF7:005C    0CF7:0000    6D72:6554
0CF7:0060   6573:6162    0000:0000    0000:0000    0000:0000
0CF7:0070   0000:0000    0438:A911    0002:1612    0CD6:0C0B
0CF7:0080   008A:0001    002E:0CF7    6944:0CF7    616C:7073
0CF7:0090   0000:0079    0000:0000    0000:0000    0000:0000
0CF7:00A0   A911:0000    55B4:0434    0C51:0002    0001:0CDD
0CF7:00B0   0CF7:00B8    0CF7:005C    6279:654B    6472:616F
0CF7:00C0   0000:0000    0000:0000    0000:0000    0000:0000
0CF7:00D0   0433:A911    0002:A5D2    0CDD:0C59    00E6:0001
0CF7:00E0   008A:0CF7    5341:0CF7    0049:4943    0000:0000
0CF7:00F0   0000:0000    0000:0000    0000:0000    A911:0000

  Position > 0CF7:005C
```

SAMPLE SCREEN 9-9

9.6 Post-Mortem Debugger Messages

The following is an alphabetical list of the Post-Mortem Debugger error messages. When error messages are caused by certain commands only, these commands are listed in brackets.

- `Already at the beginning of the array`
  [Data window, Left command]

  The first element of the array is already being displayed.

- `Already at the end of the array`
  [Data window, Right command]

  The last element of the array is already being displayed.

- `Already at top level (current process)`
  [Data window, Father and Variables commands]

  The descriptor of the selected process is already being displayed.

- `Already at top level`
  [Data window, Father and Variables commands]

  The variables of the selected procedure or module are already being displayed.

- `Assert: 'error message'`

  The debugger aborts with this message when an internal error is detected. We hope that this message will never appear. If it does, please report it to LOGITECH.

- `Can not display data window with a wrong reference file`
  [Data command]

  When a wrong reference file is used, symbolic debugging is not possible for that module and its procedures. In this case the debugger cannot display the Data window. (For further explanation, see the note at the end of this section.)

- `Can not display definition module without the text file (.DEF)`
  [Text window, Definition command]

  The text of the definition module cannot be displayed because the corresponding source file is missing.

- `Can not display text window without the reference file`
  [Text command]

  The text of the selected module cannot be displayed because the corresponding reference file is missing. To display the text of a module, both the source and the reference files are needed.

- `Can not display text window without the text file (.MOD)`
  [Text command]

  The text of the selected module cannot be displayed because the corresponding source file is missing.

- `Can not find reference file (disk removed)`

  The reference file cannot be opened where it was found before. Most likely, the disk has been removed. Reinsert the disk and try again.

- `Can not find text file (.MOD) (disk removed)`

  The source file cannot be opened where it was found before. Most likely, the disk has been removed. Reinsert the disk and try again.

- `Invalid set`
  [Data window, Son command]

  If a SET variable has its unused bits uncleared, the value is considered INVALID. The unused bits are the bits which are not used to represent the value of the set.

- `No code address in the definition module`
  [Text window, Address command]

  Because a definition module does not contain any code the Address command is not applicable.

- `No data in the selected module`
  [Data command, Exchange command in Data window]

  The selected module does not contain any variable declarations.

- `No data in the selected procedure, global data displayed`
  [Data command, Exchange command in Data window]

  The selected procedure contains neither a parameter list nor any variable declarations. If global variables exist in the module, they are shown and this message is displayed.

- `No data in the selected procedure and module`
  [Data command, Exchange command in Data window]

  The selected procedure and module contain neither a parameter list nor any variable declarations.

- `No selected procedure in this module`
  [Data window, Exchange command]

  When the Data window shows module data, the Exchange command can only be used if the selected procedure belongs to the same module.

- `No text associated with the current process`
  [Text command]

  The last element (PROCESS) displayed in the Call window represents the process descriptor kept internally by the Modula-2 system. It does not correspond to any particular declaration or statement in any particular source file.

■ `Not modified (page length too small)`
[Lines command]

The number typed in for the new page length is invalid.

■ `No valid call list for the examined process`
[Data window and Raw window, Examine command]

The values found in the memory dump indicate that the call list of the process is invalid. Most likely, the call list has been destroyed by your program or you are trying to examine an invalid process.

■ `No valid module list for the examined process`
[Data window and Raw window, Examine command]

The module list found in the memory dump is invalid. Most likely, the module list has been destroyed by your program or you are trying to examine an invalid process.

■ `No valid process descriptor (out of dump)`
[Data window and Raw window, Examine command]

The process descriptor found in the memory dump is invalid. Most likely, the process descriptor has been destroyed by your program or you are trying to examine an invalid process.

■ `Out of dump`

The debugger tried to display or use the contents of the memory dump at some address that is not a part of the dump. This can happen if you entered an invalid address. It also can happen if the memory dump contains invalid addresses or pointers. The memory dump contains all memory locations that might legally have been used by your program. Most likely, you are trying to use an uninitialized or invalid pointer or address. This might be happening if your program has destroyed (overwritten) part of the memory.

■ `Out of dump (NIL value)`

The debugger tried to use an address or pointer whose value was NIL.

- `Out of dump (>1 M Byte)`

  The debugger tried to use an address which is outside of the address space of one megabyte.

- `This data is not an array element`
  [Data window, Left and Right command]

  The selected data item is not part of an array structure. The Left and Right commands are applicable inside of an array structure only.

- `This data is not structured`
  [Data window, Son command]

  The type of the selected data item is not a structured type (record, array, pointer). The Son command is applicable to structured data only.

- `This local module does not contain variables`
  [Data window, Son command]

  The selected local module does not contain any variable declarations.

- `This data is not a PROCESS`
  [Data window, Examine command]

  The type of the selected data item is not PROCESS. The Examine command is applicable to processes only.

- `This line does not contain a statement`
  [Text window, Address command]

  The selected line in the source text does not contain any statement. The Address command of the Text window is applicable to statements only.

- `This module name is not in the module list`
  [Module window, Find command]

  The selected module is not found.

- `This procedure/module name is not in the module`
  [Text window, Find command]

  The selected procedure or local module is not found.

- `Too many modules`
  [Initialization of the PMD]

  The number of modules of the process that was stopped exceeds the limit set in the debugger. In the initialization of the PMD, this error is fatal (the debugger is left).

- `Too many procedure calls (remaining calls can not be viewed)`

  The number of procedures in the call chain (of the stopped or selected process) exceeds the limit set in the debugger. Symbolic debugging is only possible for the procedures within the limit.

- `WARNING: markers and breakpoint locations could be incorrect (wrong files)`
  [Text command]

  Some inconsistency has been detected between the reference file and the source file. This might happen if the source is modified after the last compilation (and linkage). (For further explanation, see the note at the end of this section.)

- `Wrong reference file (too large)`

  When a reference file is larger than 64K bytes, symbolic debugging is not possible for that module and its procedures. The corresponding modules must be split into smaller modules, to allow symbolic debugging.

- `Wrong version of reference file`

  When a wrong reference file is used, symbolic debugging is not possible for that module and its procedures. (For further explanation, see the note at the end of this section.)

- `Wrong version of reference file (bad structure)`

  Some inconsistency inside of the reference file has been detected. Symbolic debugging is not possible for that module and its procedures. (For further explanation, see the note at the end of this section.)

- `Wrong version of reference file (compiler version)`

  The debugger cannot recognize the format of the reference file. This may happen when you try to use a compiler and a debugger which are not from the same release. In this case, symbolic debugging is not possible for that module and its procedures. Please use the appropriate versions of the compiler and the debugger. (For further explanation, see the note at the end of this section.)

- `Wrong version of reference file (module name conflict)`

  The name of the module in the reference file does not match with the name of the module loaded. When a wrong reference file is used, symbolic debugging is not possible for that module and its procedures. (For further explanation, see the note at the end of this section.)

NOTE

The debugger displays an error message if it detects some inconsistency between the memory dump and the reference file or between the source file and the reference file. This might happen if the source or the reference file (or both) used for debugging does not correspond to the version of the module used in the program being debugged. For instance, one of these files might contain an old version. Or, more likely, these files correspond to a newer version of the module because the module has been changed and/or recompiled since the program was linked. The best way to solve this problem is to recompile this module, to relink the program and to debug this new version (using a new memory dump). If this is not possible, you should quit the debugger and restart it using the Query option. When the debugger prompts you to enter the file names of the files in question, enter <ESC> to indicate that these files are not present. In this case, no symbolic debugging will be possible for that module, but the debugger will not display incorrect information.

9.7 Sample Program 'Demo'

```
003F   MODULE  Demo;
003F
003F       FROM Terminal IMPORT WriteLn, WriteString;
003F
003F       TYPE DATA = RECORD
003F                        data1 : CARDINAL;
003F                        data2 : REAL;
003F                        data3 : INTEGER
003F                     END;
003F
003F       VAR x : INTEGER;
003F        y : REAL;
003F        z : INTEGER;
003F        node : ARRAY [1..4] OF DATA;
003F
003F   PROCEDURE FirstOne (x:CARDINAL,y:REAL;z:INTEGER);
0049   BEGIN
0051        node[1].data1 := x;
005D        node[1].data2 := y;
0068        node[1].data3 := z;
0076        INC (x);
007A        y := y + 1.0;
0094        INC (z);
00A8        RecursiveOne (x, y, z)
00B1   END FirstOne;
00B1
00B1   PROCEDURE RecursiveOne(x:CARDINAL,y:REAL;
                                                  z:INTEGER);
00BD    BEGIN
00DE        WITH node [x] DO
00E8                data1 := x;
00F1                data2 := y;
0101                data3 := z
010B        END; (** WITH **)
0115        INC (x);
0119        y := y + 1.0;
0133        INC (z);
```

```
014F        IF x > 3 THEN LastOne (x, y, z)
0169                      ELSE RecursiveOne (x, y, z)
016C        END (** IF **)
0172   END RecursiveOne;
0172
0172   PROCEDURE LastOne (x:CARDINAL; y:REAL;  z:INTEGER);
0175     BEGIN
017D        node[4].data1 := x;
0189        node[4].data2 := y;
0194        node[4].data3 := z;
01A2        INC (x);
01C4        y := y + 1.0; HALT; INC (z)
01CA    END LastOne;
01CA
01D4    BEGIN     (** MAIN **)
01E2            WriteString ('Here we go ... ');
01E7            WriteLn;
01EC             x := 1;
01F7            y := 2.0;
0204            z := 3;
022B            FirstOne (x, y, z)
0000   END Demo.
```

10 THE SYMBOLIC RUN-TIME DEBUGGER

10.1 Introduction

The symbolic run-time debugger allows you to monitor the execution of a program. You can execute the program in steps, simulating slow motion. After each step, you may inspect the data and the current status of the program. You can modify the values of the variables the program uses. There are several ways you can step through the program. Depending on the situation, you may decide to execute in larger or smaller steps.

The structure and user interface of the run-time debugger are very similar to that of the post-mortem debugger. The run-time debugger uses the same windows and screen layout as the post-mortem debugger. The run-time debugger commands are a superset of the post-mortem debugger commands:

- All global commands of the post-mortem debugger are also valid in the run-time debugger.
- In any particular window, all local commands of the post-mortem debugger are also valid in the run-time debugger.

This chapter describes those features and functions which are specific to the run-time debugger. The post-mortem debugger is documented in another chapter. Please refer to that chapter for a description of the commands that are common to the post-mortem debugger and the run-time debugger.

The sample screens used for illustration in this chapter refer to the sample program 'Demo' included at the end of the chapter on the symbolic post-mortem debugger.

10.1.1 Installation

There are two ways to install the MODULA-2 run-time debugger depending on whether you have two floppy disk drives or one floppy and a hard disk. In both cases we assume you have successfully completed the installation process as described in the Installation chapter.

Two Floppy Disk Drives

If your system uses two floppy disk drives, prepare a copy of the MODULA-2 run-time debugger disk. Use this disk when running the debugger on .LOD files. Your debugger disk contains:

- the MODULA-2 run-time debugger files (.LOD)
- the reference files (.REF) of the library modules

One Floppy and a Hard Disk

If your system is equipped with a hard disk and one floppy, copy all .LOD files in the directory where you keep MODULA-2/86 executable files (usually in '\m2lod'). Copy all the .REF files in the directory where you keep .REF files (usually in '\m2lib\ref').

Before you run the run-time debugger, make sure that PATH and 'M2xxx' environment variables are correctly set.

The reference file for module MathLib0, running for the mixed emulator version (M87MATH0.REF), is distributed in your Base Language System - Linker diskette 3/3.

10.2 How to Run the Run-Time Debugger

To initialize the run-time debugger, enter:

```
A> m2 rtd<CR>
```

The debugger responds with a sign-on message:

```
MODULA-2/86 Run-Time Debugger
```

followed by the version number and a copyright notice. Then, the debugger asks for the name of the program you wish to debug. Enter the name of the load file (filetype .LOD) followed by <CR>. The debugger will then load your program into memory, and display the Module window. At this point, the program has not started to execute.

You may set breakpoints before executing the program. You instruct the debugger to start the execution of the program by entering one of the Go commands.

When the program has terminated, the debugger prompts you to enter the name of the next program to debug. Enter <ESC> to exit the run-time debugger.

10.2.1 Run-Time Debugger Options

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

```
    A> m2 rtd/q/s<CR>
```

when starting the run-time debugger.

The following options are available:

- **/Query**

  The Query option indicates that reference and source files should be searched for according to the query search strategy (see the corresponding section of the MODULA-2/86 manual for a description). You will be prompted to enter the reference and source file names. If the Query option is not specified, the debugger automatically searches for these files according to the default search strategy.

- **/Display**

  The Display option indicates that the debugger has to save/restore the screen of the application when a breakpoint is encountered or when the execution is resumed. This option allows the debugging of a program which does input/output to the screen. Note that the content of the application's screen cannot be seen when the programmer communicates with the debugger.

- **/Large**

  The Large option enlarges the internal workspace of the run-time debugger. This workspace is used for storing information on the program being debugged. In particular, it contains information for each module of the program. When debugging large programs consisting of many modules, the default workspace of the run-time debugger may be too small. This would lead to a stack or heap overflow in the debugger itself. If you use the Display option, the debugger saves the content of the screen in its workspace. It is important to have enough space allocated in the workspace for the copy of the screen.

  The size of the default workspace is 16 K bytes. When the Large option is used, this size is increased to 32 K bytes.

- **/Swap**

  The Swap option enlarges the memory available to the program being debugged. This enlargement is made by swapping a part of the run-time debugger code with the program being debugged. A more complete description of the Swap option is given in the next section.

- **/Version**

  The Version option displays the date and version of the run-time debugger.

- **/Help**

  The Help option turns the help mode on and off. If the help mode is on, the two help lines are displayed on the top of the screen. The default is help mode on. This can be changed with the option /H- when you start the RTD.

10.2.2 Memory Requirements and Swapping

The run-time debugger requires approximately 225 K bytes of memory to run. The remaining memory can be used by the program being debugged. For example, on a system with 256 K bytes of memory, you can debug a program that uses approximately 35 K bytes.

The requirement of 225 K bytes includes the run-time debugger and approximately 34 K bytes for the operating system (DOS 2.0). If your operating system is larger than 34 K bytes, you should compute the requirements accordingly.

The run-time support (M2.EXE) reserves enough K bytes of memory for the DOS command interpreter (COMMAND.COM) on top of memory. This area of memory can be used to load the application program. The only disadvantage of this is that DOS will need to load from disk its command interpreter each time the debugger terminates.

To run the debugger using all the available memory, type:

```
        A> m2/f=0 rtd<CR>
```

Refer to the Program Execution chapter for more details.

With the Swap option, it is possible to enlarge the memory space available to the program being debugged by approximately 40 K bytes. On a system with 256 K bytes of memory, this allows you to debug programs that use up to 75 K bytes.

When you specify the Swap option, part of the run-time debugger is only loaded into memory when needed. This part includes the handling of the Call window, the Module window, the Data window and the Raw window. When the program has been stopped and you invoke one of these windows, the program is swapped out to disk. It will be swapped into memory as soon as you resume execution.

Note that the handling of the Text window belongs to the resident part of the run-time debugger. As long as you activate this window only, no swapping will occur. This allows you to step through a program avoiding the delay caused by swapping.

When you choose the Swap option, the debugger creates the two swap files RTDSWAP.RTD and RTDPROG.RTD in the current directory of the current drive. Both files have a fixed size of approximately 45 K bytes. Therefore, when using the Swap option you should make sure 90 K bytes of disk space are available.

10.2.3 Programs Taking Command Line Arguments

With the run-time debugger, you can debug programs that take arguments on the command line. When the debugger asks for the program to be debugged, you should enter the arguments in the usual way. For example:

Assume the program 'mycopy' is normally started under DOS by entering:

```
        A> m2 mycopy file1 file2<CR>
```

With the run-time debugger, the following will start the program in the same way:

```
        name of the program (.LOD)> mycopy file1 file2<CR>
```

10.3 Control of Program Execution

There are two ways you can control the program being debugged. One is to set breakpoints on some specific statements of the program. The other is to step through the program, stopping at each statement or procedure call.

When the debugger stops the execution of the program, either at a breakpoint or after a step has been executed, you can inspect and modify the content of variables in any part of the program. You may examine any process, and you may view or change the data of any module or any active procedure.

10.3.1 Breakpoints

One way for you to monitor program execution is to indicate to the run-time debugger certain points at which the execution of the program should stop. These points are called breakpoints. When the program executes a statement on which a breakpoint is set, the program stops and you may examine the data structures and the status of the program.

You may set a breakpoint on any statement of the program. The debugger sets no limit to the number of breakpoints. You may set or remove breakpoints before you start the execution of the program or any time the program is stopped.

Each breakpoint has an occurrence counter associated with it. Each time you set a breakpoint, the debugger prompts you to specify a limit for the occurrence counter. This counter tells the debugger how many times to execute the statement before stopping the program. Once an occurrence counter has reached its limit, the debugger stops the program each time it encounters this breakpoint.

For example, you set the limit of the counter for a particular breakpoint to five. The run-time debugger will execute the program until the fifth time it reaches the statement on which this breakpoint is set. If you continue the execution of the program, the debugger will stop the program each time this breakpoint is encountered.

10.3.2 Step Mode

You can also instruct the debugger to execute the program statement by statement or procedure call by procedure call. The debugger 'steps' through the program stopping its execution at the beginning of the next statement or procedure call. Another possible step is to execute the program up to the return from the current procedure. If a breakpoint is encountered during the execution of a step, the program will stop at the breakpoint. Anytime the program is stopped, you may examine its current status and data.

10.3.3 Overview of the Run-Time Debugger Commands

There are five global commands which most clearly distinguish the run-time debugger from the post-mortem debugger. These commands allow you to control the execution of the program by stopping at specific points in the program. Whenever the program is stopped, you can examine its current status, and display and modify its data. In this way you can determine more specifically the location and cause of problems in your program.

The five global commands are described in detail in the corresponding section. The following list briefly defines each command. You invoke these global commands by entering the letters of the command name, shown in upper case on the command line. For example, you activate the Go Breakpoint command by typing GB.

- Go Breakpoint

  Stop at the next breakpoint or overlay loading.

- Go Statement

  Stop at the next statement, breakpoint, or overlay loading.

- Go Procedure

  Stop on the next procedure call, breakpoint, or overlay loading.

- Go Return

  Stop on the return from the current procedure, or the next breakpoint, or overlay loading.

- Go End

  Execute the program until the end, ignoring breakpoints, but not the overlay loading.

10.3.4 Run-Time Errors

When a run-time error occurs in the program being debugged or when the program calls the standard procedure HALT, the run-time debugger gains control and displays the Call window. No memory dump (file MEMORY.PMD) is generated. The run-time debugger also indicates in the Call window the cause of the run-time error. You can now inspect the program, but you cannot resume the execution. When you activate a Go command, the debugger displays the following message:

```
Note: Program stopped due to error or HALT
```

Then, the debugger asks for a new program to debug, as when the program terminates normally.

10.3.5 Stopping Programs During Execution

A program being debugged, with or without the run-time debugger, should import module 'Break', so that its object file will include this module.

The program being debugged can be stopped by typing <Ctrl-C> when it is waiting for input, or <Ctrl-Break> at any other time it is executing, for instance when it runs in an infinite loop. If a program that contains module 'Break' is stopped in this way, the run-time debugger handles this situation in the same way as when a run-time error occurs. It displays the Call window, and you can inspect the status and the data of the program as they were when <Ctrl-C> or <Ctrl-Break> was typed. It is not possible to resume the execution of the program. Upon the next Go command, the run-time debugger will display a message. It then prompts you to enter the name of the next program to debug, as when the program terminates normally.

If a program that does not contain module 'Break' is stopped by <Ctrl-C> or <Ctrl-Break>, the run-time debugger will not display the Call window. Instead, it will just terminate the program.

10.3.6 Debugging Programs that Use Overlays

Each time an overlay is called, the run-time debugger stops the execution when the overlay has been loaded, but before it has started execution. This is similar to what happens when you start debugging a program. The debugger displays the Module window when the overlay has been loaded. You may then set breakpoints or start the execution of the overlay in step mode.

10.4 Window Format

The run-time debugger has the same windows as the post- mortem debugger: the Call window, the Module window, the Data window, the Text window and the Raw window. As in the post-mortem debugger, the first two lines of each window indicate the commands available.

10.4.1 Markers

As in the post-mortem debugger, the greater-than (>) sign is used in the run-time debugger as an execution marker to indicate active code. It appears in the Call, the Module and the Text windows and its meaning is the same in the run-time debugger as in the post-mortem debugger.

In the Call, the Module and the Text windows certain lines are marked with an asterisk (*) to indicate where you have set breakpoints throughout the program. A breakpoint can be set at any statement in any procedure or module.

The breakpoint at which a program stops is marked with a pound sign (#) which replaces the asterisk.

10.4.2 Selecting an Item for Display

Like the post-mortem debugger the run-time debugger displays the position of the selected item in the lower part of each window. You may select a different item using the cursor keys or by entering a new position.

10.5 The Run-Time Debugger Commands

Like the post-mortem debugger, the run-time debugger has two types of commands - global and local. The same definitions apply to these commands in the run-time debugger as in the post-mortem debugger. Local commands are only applicable to the particular window in which they appear and are explained in the appropriate sections.

10.5.1 Global Commands

In addition to the global commands available in the post-mortem debugger, six new global commands are available in the run-time debugger. The global commands appear on the second line of each window, below the window name and local commands:

```
=[Call Mod Data Txt Raw Ini] Hex Quit #[P L N H D]
Go[Bpt Pro Ret Sta End]
```

The following describes the global commands available in the run-time debugger only:

- **Go Breakpoint**

  Instructs the debugger to execute the program until the next breakpoint or the next overlay loader.

- **Go End**

  Instructs the debugger to execute the program until the end, ignoring all breakpoints. It will, however, stop on the overlay loadings.

For the following commands, the debugger stops the program either at the next breakpoint it encounters or the next overlay loading, or after the specified step has been completed, whichever comes first:

- **Go Procedure**

  Instructs the debugger to execute the program until the next procedure call, If no breakpoint or overlay loading is encountered, the execution stops at the beginning of the procedure, right after it has been called.

- **Go Return**

  Instructs the debugger to execute the program until the return from the current procedure. If no breakpoint or overlay loading is encountered, the execution stops at the statement following the procedure call in the calling procedure.

- **Go Statement**

  Instructs the debugger to execute the program until the next statement, or to the next breakpoint, or to the next overlay loading.

10.5.2 Activating the Step Mode

When you invoke the Go Statement or the Go Procedure command, the step mode is active only in certain modules. The debugger executes the program and stops at each statement or procedure in those modules in which you have enabled the step mode. Unless a breakpoint is encountered, the program will not stop in a module where the step mode is not enabled. When a program is loaded by the debugger, by default the step mode is disabled in all modules that belong to the system library. For all other modules, the step mode is enabled.

In the Module window, the run-time debugger marks modules where the step mode is enabled with a plus sign (+) preceeding the module name. It does not mark modules with step mode disabled. You may change the default and enable or disable the step mode in any module when the Module window is displayed.

10.5.3 Display of Information

When the debugger stops executing the program at a breakpoint or after a step has been performed, it displays the same window which was shown when you initiated the execution of the program. For example, if you invoke the Go Procedure command from the Text window, the debugger will again display the Text window when it stops executing the program.

10.5.4 Use of the Step Mode in a Multi-Process Program

If the program to be debugged contains more than one process, the step mode is only applicable to one process at a time. The commands Go Statement, Go Procedure and Go Return always refer to the current process only. When you invoke one of these commands, the debugger will stop the program in the current process - the same process in which it was stopped the last time.

If you wish to stop the program in another process, you must set a breakpoint on a statement in a procedure that will be executed by this other process. When the debugger encounters this breakpoint, you select the appropriate step mode to examine this new process. The step mode is then only applicable to the new process. Whenever the debugger stops the execution of the program you can set appropriate breakpoints to stop the program in the original process, or in any other process.

10.6 Run-Time Debugger Windows

The following sections describe those aspects of the run-time debugger windows which differ from the post-mortem debugger windows. They explain the local commands which are available in the run-time debugger only.

10.6.1 Call Window

The Call window in the run-time debugger has the same major components and functions as in the post-mortem debugger. It displays the chain of procedure calls of a process. In the run-time debugger the Call window cannot be invoked before you have started the program with the Go command. Because no procedure of the program is active at that time, the Call window would be empty. When this error occurs, the debugger displays the following message:

```
    Error: Cannot display Call window during loading
```

to indicate that the program has been loaded into memory but has not been started yet.

There are no local commands available in the run-time debugger Call window.

The sample screens used for illustration in this chapter refer to the sample program 'Demo' included at the end of the chapter on the symbolic post-mortem debugger.

The following example shows the Call window. The message 'Status: procedure step' indicates that the program stopped after it completed a Go Procedure command. The two procedures marked with an asterisk have breakpoints in them.

```
CALL |
=[Call Mod Data Txt Raw Ini] Hex Quit #[P L N H D] Go[Bpt
                                              Pro Ret Sta End]
-----------------------------------------------------------
Status: procedure step
 procedure step for this process

 1 *> RecursiveOne   in Demo stops at line 36, statement 1
 2 *> RecursiveOne   in Demo       at line 38, statement 1
 3 > FirstOne        in Demo       at line 24, statement 1
 4 > initialization  of Demo       at line 57, statement 1
 5 > PROCESS

Position > 1
```

SAMPLE SCREEN 10-1

10.6.2 Module Window

The Module window displays the list of modules that constitute the program being debugged. The modules in which the step mode is enabled are marked with a plus sign (+).

Local Commands in the Module Window

There are three new local commands in the Module window of the run-time debugger. You invoke them by entering the first character (shown in upper case) of each command name. The local commands appear on the first line of the window, to the right of the window name as follows:

```
MODULE | Find Enablestep Disablestep Alldisablestep
```

- **Enablestep**

  Enables the step mode in the selected module. When you invoke the Go Procedure and Go Statement commands to step through the program, the program will only stop in the modules where the step mode is enabled.

- **Disablestep**

  Disables the step mode in the selected module. For all modules of the system library, the step mode is disabled by default.

- **Alldisablestep**

  Disables the step mode in all the modules.

10.6.3 Data Window

The Data window displays the variables and/or parameters of the selected procedure or module.

Local Commands in the Data Window

There is one additional local command in the Data window of the run-time debugger. The Modify command appears at the end of the first line of the window after the window name and the other local commands.

- **Modify**

  Modifies the contents of the selected variable or parameter. The debugger prompts you to enter the new value according to the type of the data item:

  - CARDINAL, INTEGER, REAL
    You enter the new value which must be of the same type.
  - BYTE, WORD
    You enter the new value as a hexadecimal number.

- ADDRESS, POINTER, PROCESS, PROCEDURE VARIABLE
  You enter the new value in the form <segment>:<offset>. Both parts are four digit, hexadecimal numbers. If you want to modify a process variable, you must enter the address of the new process descriptor. To modify a procedure variable, enter the address of the entry point (BEGIN).

WARNING:
The modification of a PROCESS variable and a PROCEDURE variable could be very hazardous. Use these modifications very carefully!

- BOOLEAN
  You change items of type BOOLEAN by entering a T for TRUE or an F for FALSE.
- CHAR
  You modify items of type CHAR by entering a character in quotes, such as 'a' or "a", or by entering an octal value.
- BITSET
  You modify items of type BITSET by entering a binary number. The binary number consists of up to 16 digits of 'one' or 'zero', indicating that the corresponding bit should or should not be set. If you do not wish to modify a certain bit, you can enter an x at this position and the debugger will retain the original value for this bit.
- SET
  You modify items of type SET by invoking the Son command to list the contents of the set. The run-time debugger then lists the possible elements in the set and indicates whether each element is in the set or not. To change the elements included in the set, you must select a particular element and activate the Modify command. By responding with T for TRUE or an F for FALSE to the prompt 'In set?' you can then include or exclude that element into or from the set.

  The modifications of a set element clears the unused bits of the set. This can be used to correct an invalid set.
- ENUMERATION
  You modify the value by entering the name of the element to which you want to set the value. The element name must be given as defined by the declaration of the enumeration type.

The following sample screens show the path you follow to modify the content of an array element with a record structure. First, you invoke the Son command to view the elements of the variable 'node' of the module 'Demo' (Sample Screens 10-2 & 10-3). Next, you again invoke the Son command to display the fields of the record 'node[1]', and the value and type of each field (Sample Screen 10-4). Finally, you modify the value of the first field which is of type CARDINAL. You invoke the Modify command and enter a 6 to change the value from 1 to 6. Sample Screen 10-5 shows the modified data.

```
DATA | Son Father Left(dec index) Right(inc) Var X Addr
                                   Examine(process) Modif
 =[Call Mod Data Txt Raw Ini] Hex Quit #[P L N H D] Go[Bpt
                                        Pro Ret Sta End]
----------------------------------------------------------
Demo.

 1  x                                1  INTEGER
 2  y                 2.0000000000E+000  REAL
 3  z                                3  INTEGER
 4  node                                ARRAY[1..4] OF RECORD

 Position > 4
```

SAMPLE SCREEN 10-2

```
DATA | Son Father Left(dec index) Right(inc) Var X Addr
                                    Examine(process) Modif
 =[Call Mod Data Txt Raw Ini] Hex Quit #[P L N H D] Go[Bpt
                                           Pro Ret Sta End]
-----------------------------------------------------------
Demo.node
 1 [1]                                    RECORD    DATA
 2 [2]                                    RECORD    DATA
 3 [3]                                    RECORD    DATA
 4 [4]                                    RECORD    DATA

   Position > 1
```

SAMPLE SCREEN 10-3

```
DATA | Son Father Left(dec index) Right(inc) Var X Addr
                                    Examine(process) Modif
 =[Call Mod Data Txt Raw Ini] Hex Quit #[P L N H D] Go[Bpt
                                          Pro Ret Sta End]
-----------------------------------------------------------
Demo.node[1]
 1  data1                                1  CARDINAL
 2  data2                2.0000000000E+000  REAL
 3  data3                                3  INTEGER

  Position > 1 new value (cardinal) > 6
```

SAMPLE SCREEN 10-4

```
DATA | Son Father Left(dec index) Right(inc) Var X Addr
                                     Examine(process) Modif
 =[Call Mod Data Txt Raw Ini] Hex Quit #[P L N H D] Go[Bpt
                                            Pro Ret Sta End]
------------------------------------------------------------
Demo.node[1]

 1  data1                               6  CARDINAL
 2  data2                2.0000000000E+000  REAL
 3  data3                               3  INTEGER

 Position > 1
```

SAMPLE SCREEN 10-5

10.6.4 Text Window

The Text window displays the text of the module or procedure in which the debugger stops the program. The greater-than sign (>) indicates the line in which the debugger stopped the program, the call of the next procedure, or where the last process transfer or interrupt occurred.

Local Commands in the Text Window

The three local commands specific to the Text window of the run-time debugger allow you to set and delete breakpoints.

- **Set Breakpoint**

  Sets a breakpoint in the selected line. If more than one statement is on the line, the run-time debugger prompts you to indicate on which statement you wish to set the breakpoint. The run-time debugger also prompts you to set a limit for the occurrence counter associated with the breakpoint. You may type <CR> for the default value for this limit which is one.

If a breakpoint is already set on the selected statement the debugger replaces the old value of the occurrence counter with the new one.

- **Clear Breakpoint**

  Removes a breakpoint on the selected line. If more than one breakpoint is on the line, the debugger prompts you to indicate from which statement you wish the breakpoint to be removed.

- **Kill Breakpoints**

  Removes all breakpoints from the program.

10.6.5 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 post-mortem debugger.

Local Commands in the Raw Window

There are three additional local commands in the Raw window of the run-time debugger.

- **Input / Output**

  Used to read in and write out data through an I/O port. The debugger prompts whether to read in or write out a byte or a word. It then asks you to enter the address of the serial port which will be used.

- **Modify**

  Allows you to modify the memory contents at the selected address. The debugger asks you for the new value and specifies in which format it should be entered. The format to be used depends on the format in which the Raw window currently displays the memory contents.

10.7 Run-Time Debugger Messages

The following is an alphabetical list of the run-time debugger error messages. When error messages are caused by certain commands only, these commands are listed in brackets. For error messages not listed in this chapter, please refer to the list of error messages of the post-mortem debugger.

- `Can not display call window during loading`
  [Call command]

  The Call window shows the chain of active procedures. If the program has not started execution, no procedures are active; thus, the Call window would be empty.

- `Local data can not be modified until past BEGIN`
  [Data window, Modify command]

  The local data of a procedure does not exist before the procedure entry code, to which the 'BEGIN' corresponds, has been executed.

- `No breakpoint in definition module`
  [Text window, Set breakpoint command]

  A breakpoint can only be set on a statement. Because a definition module does not contain statements, no breakpoints can be set there.

- `No breakpoint to clear`
  [Text window, Clear breakpoint command]

  No breakpoint is set at the selected statement, therefore it cannot be removed.

- `No statement in this line`
  [Text window, Set breakpoint command]

  The selected line does not contain any statements. A breakpoint can only be set on a statement. A line that contains only a symbol like 'END', 'IF', 'CASE', 'LOOP' or similar is considered to contain no statement.

- `Not enough workspace to execute this command`
  [Any window]

  The workspace is too small. Use the Large option to enlarge it.

- `Not modified (new value out of range)`
  [Data and Raw windows, Modify command]

  The new value the user entered is not within the valid range. No modification has been made.

- `Process descriptor can not be modified`
  [Data window, Modify command]

  The run-time debugger does not allow you to modify process descriptors. Modification of the process descriptors may cause unpredictable behavior of the program and the run-time debugger.

- `Structured data can not be modified (use Son command)`
  [Data window, Modify command]

  The Modify command is applicable only when the selected data item is of a simple type. You should invoke the Son command and select those elements or fields you want to modify.

- `This type of data can not 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.

11 UTILITIES

11.1 Introduction

The following utility programs are described in this chapter:

- VERSION
  Helps to administrate different versions of one program.

- M2DECOD
  Decodes MODULA-2/86 link and load files into text files containing information such as disassembled code or data.
- XREF
  Generates cross reference information tables of text files, especially of Modula-2 source texts.

11.1.1 Installation

All utilities come in both .EXE and .LOD format. The .EXE format can be invoked by simply typing the utility name, for example, 'XREF<CR>'. The .EXE format is generally faster to load in memory. The .LOD version is provided so that you can run the utility from the editor using the command 'run' in the File Menu (Alt-F9) and typing the utility name followed by <CR>.

There are two ways to install the MODULA-2 Utilities depending on whether you have two floppy disk drives or one floppy and a hard disk. In both cases, we assume you have successfully completed the installation process as described in the Installation chapter.

Two Floppy Disk Drives

If your system uses two floppy disks, copy the MODULA-2 utility disk (XREF, M2DECOD, VERSION). Use this disk when running the utilities.

One Floppy and a Hard Disk

If your system is equipped with a hard disk and one floppy, copy all the .LOD and L2ERTS.L2E files in the directory where you keep MODULA-2/86 executable .LOD files (usually in '\m2lod').

Copy all .EXE files in the directory where you keep all .EXE public executable files. This directory should be contained in your PATH variable (see MS DOS environment and PATH variables).

11.2 The Link-Load File Decoder Utility M2DECOD

The utility program 'M2DECOD' decodes MODULA-2/86 link and load files. The different records of the file are written on the output file in textual form. The text file contains information such as disassembled code or data areas.

The codeblocks are disassembled in a more or less standard assembler language. In the most simple use of the utility, there are no symbols used for variables, procedures and labels. For each instruction the value of the IP, the code in HEX and a mnemonic form of the instruction are written. Each procedure is identified by a number. The correspondence of this number with the procedure in the source program can be found with the following rule:

The body of the main module is number 0. The procedures declared in the definition module are numbered starting from 1 according to their declaration sequence. The procedures that appear only in the implementation module get the numbers in the order of their declaration (header). Local modules are handled as procedures.

Because the generated output may be large in case of large modules, the decoder generates a multiple of decoded files, each of them smaller than 60K bytes. This makes it possible to look at them with an editor that has the famous 64K limitation. The filenames of the output files are automatically generated, starting with the extensions .DC1, .DC2, and so on.

To decode a link file type:

```
A> m2decod exampl<CR>
```

To decode a load file you must type the extension:

```
A> m2decod exampl.LOD<CR>
```

M2DECOD can add symbolic information to the decoded file in two ways:

- Using the REF file, the decoded file is enriched with procedure names, line numbers, variable names, addressing modes, and RTS calls. The link/load file decoder detects the RTS calls when the following instructions appear in a code area:
  MOV AL,<function code>
  INT <RTS vector>
  The following option indicates whether the decoder is using the REF file:
  /R+ (default)
  /R-
- Using the REF file and the MOD file, the decoded file contains the Modula-2 statements instead of the line numbers. The following option indicates whether the decoder uses the MOD file:
  /S+ (default)
  /S-

Note that the MOD file can be used only if the REF file is present.

The decoder uses the SYM files of the imported modules to find the name of the imported procedures.

- The following option defines the search mechanism for the SYM, REF, and MOD files:
  /Q+
  /Q- (default)

If the option is set, the decoder will prompt for a file name; otherwise, the program first will try to automatically find the file with the default search strategy.

If there is no need to decode the whole module, you can set an option to decode either a list of procedures or one module.

- The following option allows you to select a list of procedures to be decoded:
  /P

Separate the procedure names by a comma ',' and indicate the end of the list by typing <CR>. Type <ESC> to exit the program. When a procedure is not visible from the module level, you must prefix the procedure name with the nesting procedure or module name followed by a period '.'.

- The following option allows you to select a module to be decoded:
  /M

You can select only one module. This option is valid for a LOD file only. Type <ESC> to exit the program.

11.2.1 Example Dialog:

A> m2decod exampl<CR>

```
LOGITECH MODULA-2/86 LNK/LOD file-Decoder, Rel. m.n
Copyright (C) 1985 LOGITECH
Input file: exampl.lnk
exampl: DK:exampl.REF
exampl: DK:exampl.MOD
Terminal: \M2LIB\SYM\Terminal.SYM
your decode file is: C:exampl.DCl
```

11.3 The Cross-Reference Utility XREF

The utility program 'XREF' generates cross reference information tables of text files, especially of Modula-2 source texts.

The program reads a text file and generates a table with line number references to all identifiers occurring in the text. All work symbols of Modula-2 are omitted from the table. The program also skips strings (enclosed by single or double quotes) and comments (from '*' to '*'). The program prompts for an input file. The default extension is .MOD. The generated table is listed on a cross-reference file in alphabetical order. Upper case letters are defined to be greater than lower case letters.

If the lines on the input file start with a number, for example, if it is a listing file generated by the compiler, XREF takes these numbers as referencing line numbers. Otherwise, XREF generates a listing file with line numbers. The name and the pathname of the output files are the same as those of the input file. The extensions are .XRF and .LST.

The program accepts some options after the filename. The options and their meaning are:

- /S
  Displays statistics on the terminal, for example, number of lines, identifiers and reference numbers.

- /L
  Generates a listing file with new line numbers.

- /N
  Generates no listing file. The line numbers in the reference table will refer to the line numbers on the input file. All lines on the input file without leading line numbers are skipped.

11.3.1 Example Dialog

A> xref<CR>

```
LOGITECH MODULA-2/86 Cross-Reference Utility, Rel. m.n
Copyright (C) 1983, 1984, 1985 LOGITECH
input file (ESC to quit) > grep.MOD/S<CR>
     lines         :    143
     identifiers   :     33
     characters    :    261
     refnumbers    :    144
input file (ESC to quit) > <ESC>
A>
```

11.4 The Source Versions Manager Utility VERSION

The purpose of this utility is to help administrate different versions of one program. All modifications to such a program are made in only one source text and the various versions can be derived automatically from that 'master copy' using Version. Note that the 'master copy' is itself the source of one of the target versions and can be compiled without being processed by Version. Once another version must be produced, Version is applied to the 'master copy' and generates the converted source text.

We suggest you always modify the same version of the program and generate the other versions from this original copy. This method will avoid confusion as to what changes were made in which versions. Furthermore, to avoid any confusion and to save disk space, the source texts of the derived versions can be deleted after their use.

11.4.1 Marking the Version Dependent Parts

In the master copy, the portions of text that belong to a specific version are enclosed by a 'start marker' and an 'end marker'. Both start and end markers are Modula-2 comments and they contain the list of versions to which they belong. There are two kinds of markers: one to activate, another to deactivate a portion of text.

Before using the markers, each version must be declared with a definition comment. Only 64 versions can be present in a file. They are numbered from 0 to 63. The version names can be written in upper or lower case. The Version Utility does not distinguish between upper and lower cases.

Syntax

```
definitionComment       :=="(*V" number "=" symbol "*)".
activeStart Marker      :=="(*<" version {"," version} "*)".
activeEndMarker         :=="(*" version {"," version} ">*)".
inactiveStartMarker     :=="(*<" version {"," version}.
inactiveEndMarker       :== version {"," version} ">*)".
version                 :== symbol | number.
```

Example 1

```
(*V1=DOS Version for IBM PC-DOS*)
(*V2=CP/M Version for CP/M 86*)
PROCEDURE ReadCh (VAR ch: CHAR);
BEGIN
(*<DOS*) DOSCALL(1, ch); (*DOS>*)
(*<CP/M CPMCALL(1, ch); CP/M>*)
If ch = CR THEN ch := EOL END;
END ReadCh;
```

A portion of text may be marked as belonging to several versions. In this case, the version names in the markers must be separated by a comma.

Example 2

```
(*V1=A*)
(*V2=B*)
(*V3=C*)
(*V4=D*)
VAR ch: CHAR;
BEGIN
(*<C,D*)
          INBYTE (OEOH, ch);
          (*<C*) IF ch = OC THEN RETURN END; (*C>*)
          buffer [i] := ch;
          INC(i);
(*C,D>*)
          Write (ch)
END
```

The versions can be specified either by a version numbers or by the symbol associated to the version. We recommend using the version name rather than the version number. The version number is automatically replaced by the version name, if it is defined, once the file is processed by Version.

11.4.2 Invoking Utility Version

To invoke the Version utility type:

```
A> version<CR>
```

The program prompts the user for the file name of the master copy and for the name of the output file (by default, it is the same file as the input file). Then Version shows the various versions defined in the master copy.

The program then prompts for the name of the versions which have been activated. To select more than one version, you enter a list of version numbers or version names separated by a comma. The program asks if the unselected parts of the master copy must be deleted. Type <ESC> to exit the program.

At the end of the processing, the Version program indicates which defined versions were encountered in the file.

11.4.3 Example Dialog

A> version<CR>

```
LOGITECH MODULA-2/86 Version, DOS 8086, Rel m.n
Copyright (C) 1985 LOGITECH
 Input file: example1.mod<CR>
 Output file: example1.mod <CR>
 Versions defined:
   1: DOS
   2: CP/M
Enter version names: cp/m<CR>
Suppress inactive text and brackets (y/-)? yes
Are you sure (y/-)? no
done
   The following versions are present:
referenced          version        defined
in the file         number         as:
-------------------------------------------
     yes               1           DOS
     yes               2           CP/M
```

A> type example1.mod

```
   (*V1=DOS Version for IBM PC-DOS*)
   (*V2=CP/M Version for CP/M 86*)
   PROCEDURE ReadCh (VAR ch: CHAR);
   BEGIN
   (*<DOS DOSCALL(1, ch); DOS>*)
   (*<CP/M*) CPMCALL(1, ch); (*CP/M>*)
   IF ch = CR THEN ch := EOL END;
   END ReadCh;
```

A> type example1.bak

```
(*V1=DOS Version for IBM PC-DOS*)
(*V2=CP/M Version for CP/M 86*)
PROCEDURE ReadCh (VAR ch: CHAR);
BEGIN
(*<DOS*) DOSCALL(1, ch); (*DOS>*)
(*<CP/M CPMCALL(1, ch); CP/M>*)
IF ch = CR THEN ch := EOL END;
END ReadCh;
```

A set of options can be specified after the input file name:

- /Q+ (Default)
  /Q-

  If the output file already exists, the programs asks whether or not the old one can be deleted. This prompt can be suppressed by entering /Q- after input filename. The old file will be deleted without any warnings. If the output filename is the same as the input filename, a backup of the input file will be generated with a 'BAK' extension.

- /B+
  /B- (Default)

  This option indicates that the Version program is used in a batch file. This option disables keyboard polling while the error listing writes to the screen.

11.4.4 Error Handling

The three types of error messages are warning messages, error messages and fatal error messages.

Warning and error messages are listed with five lines which surround the location where the error was detected. To interrupt the listing, type any key and then, type any key to continue.

Warning messages

The warning messages are only indicators that something is wrong. The execution is not stopped.

- `warning: can't activate version because of deactivated environment`

  This message occurs when a portion of text which should be activated is nested in a deactivated one.

Error messages

The execution is not stopped but the output file will be deleted.

- `error: identifier not declared`

  An undeclared version name is used in a marker.

- `error: syntax error in version list`
- `error: open bracket(s) at the end of file`
- `error: structure error in nesting of brackets`

  Occurs when a nesting bracket is closed before the nested one.

- `error: two consecutive open brackets with same symbol(s)`
- `error: bracket terminator expected`
- `error: bad identifier`

- `error: bad version number`
- `error: "=" required`
- `error: multiple definition of identifier`

Fatal error messages

These messages occur when the processing cannot be continued due to an internal problem of Version. The execution is stopped and the output file is deleted.

- `fatal error: too long symbol`

  This error occurs when an identifier that is too long is encountered. The maximum length of an identifier is 40 characters.

- `fatal error: buffer full`

  The input buffer is full. This error occurs if there is a bracket that is too long. The maximum length for a bracket is 256 characters.

- `fatal error: can't create table`

  There is not enough memory to create the name table.

- `fatal error: name table full`

  Too many versions are defined.

12 SYSTEM AND LIBRARY MODULES

Unlike many other programming languages, the Modula-2 language does not provide standard functions that handle input and output. In Modula-2 systems, these functions are typically provided by library modules. Library modules that implement the typical, basic operating system functions are also called 'system modules' or 'low level modules'.

Besides these system modules the MODULA-2/86 module library contains a number of additional, more general library modules. Examples are modules that provide for formatted input/output, or modules that provide functions that do not depend on the operating system, for example, string handling.

This section provides an overview about the system and general library modules available with MODULA-2/86. The definition parts of all library modules are given in the library section of this manual.

12.1 The System Modules

There are a number of system modules that constitute the interface between the MODULA-2/86 system and the application programs. They provide access to the underlying operating system and to the MODULA-2/86 system itself. These modules may be used, imported, directly by application programs.

System modules rely on the support of the operating system. For some of the system modules, even the definition part is operating system dependent. These system modules should be considered internal modules of the MODULA-2/86 system. They should not be used directly by application programs, because this would severely affect program portability.

The following is a list of highly system dependent modules of the MODULA-2/86 system that should not be used directly. Their filenames are given in parentheses:

■ DiskFiles	(DISKFILE.DEF)
■ Display	(DISPLAY.DEF)
■ Keyboard	(KEYBOARD.DEF)
■ Reals	(no definition module)

Partially system dependent modules of the MODULA-2/86 system which should be used with care:

▪ System	(SYSTEM.DEF)
▪ Termbase	(TERMBASE.DEF)

System modules that are part of the interface between the MODULA-2/86 system and application programs:

▪ Break	(BREAK.DEF) (for IBM-PC only)
▪ Debug	(DEBUG.DEF)
▪ Devices	(DEVICES.DEF)
▪ DiskDirectory	(DISKDIRE.DEF)
▪ DOS3	(DOS3.DEF)
▪ DOS31	(DOS31.DEF)
▪ ErrorCode	(ERRORCOD.DEF)
▪ FileSystem	(FILESYST.DEF)
▪ LogiMouse	(LOGIMOUS.DEF)
▪ Mouse	(MOUSE.DEF)
▪ Program	(PROGRAM.DEF)
▪ Storage	(STORAGE.DEF)
▪ Terminal	(TERMINAL.DEF)
▪ TimeDate	(TIMEDATE.DEF)

12.2 The General Library Modules

General library modules provide features that are not typically part of operating systems. In some cases, they provide access to operating system features through higher level concepts.

The MODULA-2/86 library includes the following general library modules. Their filenames are given in parentheses:

▪ ASCII	(ASCII.DEF)
▪ CardinalIO	(CARDINAL.DEF)
▪ Conversions	(CONVERSI.DEF)
▪ Decimals	(DECIMALS.DEF)
▪ Directories	(DIRECTOR.DEF)
▪ FileMessage	(FILEMESS.DEF)

▪ FileNames	(FILENAME.DEF)
▪ InOut	(INOUT.DEF)
▪ MathLib0	(MATHLIB0.DEF)
▪ NumberConversion	(NUMBERCO.DEF)
▪ Options	(OPTIONS.DEF)
▪ Processes	(PROCESSE.DEF)
▪ ProgMessage	(PROGMESS.DEF)
▪ RealConversions	(REALCONV.DEF)
▪ RealInOut	(REALINOU.DEF)
▪ RS232Code	(RS232COD.DEF)
▪ RS232Int	(RS232INT.DEF)
▪ RS232Polling	(RS232POL.DEF)
▪ Strings	(STRINGS.DEF)

12.2.1 Brief Descriptions of Library Modules

For quick reference, an alphabetical listing of the library modules with brief functional descriptions of each is provided in this section. More detailed information is available in the library section of this manual.

- MODULE ASCII
  Symbolic constants for non-printing ASCII characters

- MODULE Break
  Handling of <Ctrl-Break> and <Ctrl-C>. The implementation runs on the IBM-PC only. Module 'Break' is not available for other systems.

- MODULE CardinalIO
  Terminal input/output of CARDINALs in decimal and hexadecimal

- MODULE Conversions
  Convert from INTEGER and CARDINAL to string

- MODULE Decimals
  Provides functions for arithmetic and formatting with decimal numbers of 18 or less digits

- MODULE Debug
  Provides debug information upon abnormal termination of a program

- MODULE Devices
  Handling of interrupt driven devices
- MODULE Directories
  Directory listing with wild cards
- MODULE DiskDirectory
  Interface to directory functions of the underlying OS
- MODULE DiskFiles
  Interface to disk file functions of the underlying OS
- MODULE Display
  Low-level Console Output
- MODULE DOS3
  Additional DOS 3.0 function calls
- MODULE DOS31
  Additional DOS 3.1 function calls
- MODULE ErrorCode
  Handle return code to the operating system
- MODULE FileMessage
  Write file status/response to the terminal
- MODULE FileNames
  Read a file specification from the terminal
- MODULE FileSystem
  File manipulation routines
- MODULE InOut
  Standard high-level formatted input/output
- MODULE Keyboard
  Default driver for terminal input

- MODULE LogiMouse
  LOGITECH mouse driver interface extensions
- MODULE MathLib0
  Real Math Functions
- MODULE Mouse
  Mouse driver interface (Microsoft compatible)
- MODULE NumberConversion
  Conversion between numbers and strings
- MODULE Options
  Read a file specification, with options, from the terminal
- MODULE Processes
  (pseudo-) concurrent programming with SEND/WAIT
- MODULE ProgMessage
  Write program status message to the terminal
- MODULE Program
  Subprogram loading and execution
- MODULE RealConversions
  Conversion between REAL numbers and strings
- MODULE RealInOut
  Terminal input/output of REAL values by means of module 'InOut'
- MODULE Reals
  LOGITECH REAL arithmetic emulator
- MODULE RS232Code
  High-speed interrupt-driven input/output via the serial port
- MODULE RS232Int
  Interrupt-driven input/output via the serial port

- MODULE RS232Polling
  Polled input/output via the serial port
- MODULE Storage
  Standard dynamic storage management
- MODULE Strings
  Variable-length character string handler
- MODULE System
  Additional system dependent facilities
- MODULE Termbase
  Terminal input/output with redirection hooks
- MODULE Terminal
  Terminal Input/Output
- MODULE TimeDate
  Get/set the current time

APPENDIX A - GLOSSARY

The following terms have specific meanings as used in the MODULA-2/86 manual:

- Base layer

  A program which calls a subprogram is the base layer of that subprogram. For example, the passes of the MODULA-2/86 compiler are called sequentially by the compiler base which is their base layer.

- Compilation unit

  A part of a program which is contained in a separate file and can be compiled separately. The compilation units in Modula-2 are definition, implementation, and program modules. These modules can be compiled separately as long as the imported definition modules have already been compiled. Only definition modules can export objects. If objects should be exported from a compilation unit, it must be split into a definition module and an implementation module.

- Definition module

  The definition part of a Modula-2 module. The section on basic concepts in this manual contains a brief description of definition modules. For more information on the use of definition modules please refer to the corresponding sections in the book Programming in Modula-2 by Niklaus Wirth.

- Development system

  The entire system, both hardware and software, used to develop a program. On the software side, this includes the operating system and any kind of utility programs and libraries. When used to develop MODULA-2/86 programs, it also includes the MODULA-2/86 run-time support, as well as the MODULA-2/86 compiler, linker, debuggers, editor, utilities, and library.

- Language support

  An assembly program, which can be 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-2/86 is part of the MODULA-2/86 run-time support.

- Library

  In general, a library is a set of functions or procedures which can be used by any program. In MODULA-2/86 the library is equal to the set of all available MODULA-2/86 library modules.

- Library module

  A MODULA-2/86 module, consisting of a definition and an implementation part, which is available for use by any MODULA-2/86 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 information on the use of implementation modules please refer to the corresponding sections in the book 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. Usually, the main module is a program module. It may also be an implementation 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 module code of that program or implementation module, which is considered to be the main module of a program. 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 the 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' in this way.

  Using overlays, a program that otherwise would require a large amount of memory can run on a computer with much less memory. MODULA-2/86 provides a simple overlay concept in the form of subprograms.

- Program

  A Modula-2 program consists of all the modules which are imported directly or indirectly by its main module. When a program is linked, all these modules are included in the resulting load file. A MODULA-2/86 program may also call another MODULA-2/86 program, 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' is also used to refer to the entity consisting of 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 inside of any other module. The module code of a program module constitutes a main program. The section on basic concepts in this manual contains a brief description of program modules. For more information on program modules please refer to the corresponding sections in the book Programming in Modula-2 by Niklaus Wirth.

- Run-time support (RTS)

  An assembly program which includes the language support and further configuration-dependent functions. These functions 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 implementaion or a program module, and has been compiled separately.

- Subprogram

  A MODULA-2/86 program which is called by another MODULA-2/86 program. A subprogram consists of all the modules which are imported directly or indirectly by its main module, and 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-2/86 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 please refer to the appendix on memory organization.

- System module

  A library module that provides access to operating system functions such as memory management, overlay loading, terminal I/O and file handling. The term 'system module' also indicates that the implementation of such a module is system dependent. If even the definition module of a system module is system dependent, it is called a low level module. Low level modules should not be used directly by application programs.

- 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-2/86 or MODULA-2/VX86 does not require a terminal or disks. The software configuration may be reduced to the MODULA-2/86 run-time support and your program.

- Workspace

  The region of memory allocated to a process for stack, program variables, heap, and subprograms. When a MODULA-2/86 program is started, it begins execution as the 'main process', and it claims the largest available region of memory as its workspace. A subprogram shares the workspace of its base layer. However, when a new process is created (see the section on system dependent facilities and the appropriate section of the book Programming in Modula-2) it must be assigned a workspace. The workspace may be in a module's data area, on the heap, or even in a procedure's local data area on the stack. Just make sure that the process doesn't have a longer lifetime than its workspace!

APPENDIX B BIBLIOGRAPHY OF MODULA-2 BOOKS

MODULA-2 FOR PASCAL PROGRAMMERS

Author:	Richard Gleaves
Publisher:	Springer-Verlag

SOFTWARE ENGINEERING WITH MODULA-2 AND ADA

Authors:	Richard Wiener, Richard Sincovec
Publisher:	John Wiley and Sons

MODULA-2 SOFTWARE DEVELOPMENT APPROACH

Authors:	Gary A. Ford, Richard Wiener
Publisher:	John Wiley and Sons

INTRODUCTION TO MODULA-2

Author:	Paul M. Chirlian
Publisher:	Matrix Publisher

MODULA-2 SEAFARER'S GUIDE AND SHIPYARD MANUAL

Author:	Edward Joyce
Publisher:	Addison Wesley

INTRODUCTION TO COMPUTER SCIENCE WITH MODULA-2

Author:	Lewis Pinson
Publisher:	John Wiley and Sons

MODULA-2 WITH DATA STRUCTURES

Author:	B.K. Walker
Publisher:	Wadsworth

MODULA-2

Authors:	John Beidler, Paul Jackowitz
Publisher:	PWS

A GUIDE TO MODULA-2

Author:	Kaare Christian
Publisher:	Springer-Verlag

DATA STRUCTURES USING MODULA-2

Author:	Richard Wiener
Publisher:	John Wiley and Sons

LIBRARY MODULES

```
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, dc1, 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;  dc1 = 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.
```

```
DEFINITION MODULE Break;
(*
   Handling of the Ctrl-Break interrupt

This module provides an interrupt handler for the
Ctrl-Break interrupt 1BH 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
                  Crtl-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
occurence 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 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.
```

```
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;


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 representa-
tion 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 '+'.
*)

END Conversions.
```

```
DEFINITION MODULE Debug;
(*
   Helps analyze unsuccessful program execution

If the module 'Debug' is linked to the application program,
no dump file is generated. Instead, the system generates a
list of the call sequence within the current process that
lead to the termination displayed on the terminal.
If the necessary reference files are available in the
current directory, each line displays the name of the
procedure, its module and the source line number, where
the next procedure is called or (in case of the first line)
where the program was stopped.

If there are missing reference files, the procedure names
are replaced by procedure numbers within the module, and
the line numbers are replaced by the value of the
instruction pointer. The module body (module
initialization code) is referred to as procedure zero.
It is followed, in the order of their declaration, by the
procedures declared in the definition module, if any, and
then by the other procedures of the module.
Sample Terminal Output:


===>  halt called

MODULE : Demo   PROCEDURE : LastOne         LINE   :     48
MODULE : Demo   PROCEDURE : RecursiveOne    LINE   :     37
MODULE : Demo   PROCEDURE : RecursiveOne    LINE   :     38
MODULE : Demo   PROCEDURE : FirstOne        LINE   :     24
MODULE : Demo   PROCEDURE : Demo            LINE   :     57


===>  halt called


To be able also to debug all initialziation codes of
imported modules, module 'Debug' should be forced to be
initialized as the first module. This can be accomplished
by importing it as the first module in the main module of
the application program.
*)

END Debug.
```

```
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 (Dec0,Dec1: DECIMAL): INTEGER;
(*
Compares two DECIMAL numbers and returns an integer value
indicating the comparison result:

 -1 if Dec0 is less than Dec1
  0 if Dec0 equals Dec1
  1 if Dec0 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 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 occurence 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.
```

```
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;
                   VAR result       : 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.
```

```
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
                0C 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 (0C) under CP/M-86.
*)


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.
```

```
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 errorClass: 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);
                        (* AX,CF *)


(* 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: ADDRESS;
                             (* DS:DX *)
                             VAR nameNumberIndFlag: BYTE;
                             (* CH *)
                             VAR nameNumber: BYTE;
                             (* CL *)
                             VAR errorCode: WORD);
                             (* AX,CF *)

    (* DOS 3.1 function 5E02H *)
    PROCEDURE SetPrinterSetup(redirectionListIndex: WORD;
                              (* BX *)
                              setupStringLength: WORD;
                              (* CX *)
                              setupBuffer: ADDRESS;
                              (* DS:SI *)
                              VAR errorCode: WORD);
                              (* AX,CF *)
(* DOS 3.1 function 5E03H *)
PROCEDURE GetPrinterSetup(redirectionListIndex: WORD;
                          (* BX *)
                          setupBuffer: ADDRESS;
                          (* ES:DI *)
                          VAR setupStringLength: WORD;
                          (* CX *)
                          VAR errorCode: WORD);
                          (* AX,CF *)

(* 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.
```

```
DEFINITION 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 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,
out:    resultFN        the file specification read,
        ReadInName      which parts are in specification.

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.
```

```
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 pysical 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>.

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 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 occured, 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..0FFFEH] OF CHAR;
(*- file buffer pointer type *)

File     = RECORD
             bufa: BuffAdd;
             (*- buffer address *)
             buflength: CARDINAL;
             (*- size of buffer in bytes. In the
                 current release it is always a
                 multiple of 128. *)
             validlength: 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)

Character Sequence          Character Returned
   in File:
Symbolic     Octal          Symbolic     Octal
--------------------------------------------------
<cr, lf>     15C, 12C         <EOL>        36C
<cr>         15C              <EOL>        36C
<lf>         12C              <EOL>        36C
<Ctrl-z>     32C              <nul>         0C

<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 apporopriately.


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)

Character to write            Character Sequence
                                   in File:
Symbolic     Octal            Symbolic     Octal
----------------------------------------------
<EOL>         36C             <cr, lf>     15C, 12C
<cr>          15C             <cr>         15C
<lf>          12C             <lf>         12C
<Ctrl-z>      32C             <Ctrl-z>     32C

<Ctrl-z>, i.e. 32C, indicates end-of-file.

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       charater to check

out:            0C for illegal characters and c otherwise;
                lowercase letters are transformed into
                uppercase letters.

Which characters are leagl 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 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 Zurich, 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.
```

```
DEFINITION MODULE LogiMouse;
(*

LOGITECH Mouse Driver Interface Extensions

Short description:

The functions implemented in this module are a Modula-2
interface to the additional functions of LOGITECH's Mouse
Driver.  For a detailed description of these functions,
please refer to LOGITECH MOUSE DRIVER PROGRAMMER'S
REFERENCE MANUAL.
Those functions which are common to the LOGITECH Mouse
Driver and to the Microsoft Mouse Driver are described in
the Mouse module. This module is to be considered an
extension of the Mouse module.


Microsoft is a registered trademark of
Microsoft Corporation

*)

  FROM SYSTEM IMPORT
    BYTE;

  EXPORT QUALIFIED
    IsLogiMouse,

    DoubleChar,
    GetVersion,

    AccelType, AccelVector,
    GetAcceleration, SetAcceleration,

    SetGrid,
    MouseMode,
    GetMouseMode, SetMouseMode,

    Accel, AccelPointer,
    MoveStr, MoveStrPointer,
    ButtonStr, ButtonStrPointer,
    KybdParam, KybdConfigType,
    GetKeyboardConfig, SetKeyboardConfig,

    RS232Param,
    GetRS232Config, SetRS232Config;

  VAR
    IsLogiMouse: BOOLEAN;
      (* Flag that indicates, whether a LOGITECH mouse
         driver is loaded or not. If its value is FALSE,
         none of the following functions will work
         properly
      *)


  TYPE
    DoubleChar = ARRAY [0..1] OF CHAR;

  PROCEDURE GetVersion(VAR version, release: DoubleChar);
    (* Logitech Mouse Driver System Call 28
           Input : AX = 28 System Call 28
                   BX <-- version
                   CX <-- release
           Output: BX --> version code
                   CX --> release number
       If the driver doesn't support this function, the
       initial values of 'version' and 'release' are
       returned. So an application can initialize them
       first to spaces (or another code) in order to
       detect if the driver supports this function or
       not.
    *)
```

```
TYPE
  AccelType = (defaultAcc,
               userAcc,
               noAcc
              );

  AccelVector = ARRAY [0..127] OF BYTE;


PROCEDURE GetAcceleration(VAR accelType: AccelType;
                          VAR accel: AccelVector);
  (* Logitech Mouse Driver System Call 29
        Input : AX = 29 System Call 29

        Output: BX    --> acceleration type
                ES:DX --> acceleration vector address
  *)

PROCEDURE SetAcceleration(accelType: AccelType;
                          accel: AccelVector);
  (* Logitech Mouse Driver System Call 30
        Input : AX = 30 System Call 30
                BX    <-- acceleration type
                ES:DX <-- acceleration vector address
  *)

PROCEDURE SetGrid(horizontalGrid,
                  verticalGrid: CARDINAL);
  (* Logitech Mouse Driver System Call 31
        Input : AX = 31 System Call 31
                BX <-- horizontal grid value
                CX <-- vertical grid value
  *)
```

```
TYPE
  MouseMode = (mouseBased,
               keyBased,
               driverInactive
              );

PROCEDURE GetMouseMode(VAR mousemode: MouseMode);
  (* Logitech Mouse Driver System Call 32
        Input : AX = 32 System Call 32

        Output: BX --> mouse mode
  *)

PROCEDURE SetMouseMode(mousemode: MouseMode);
  (* Logitech Mouse Driver System Call 33
        Input : AX = 33 System Call 33
                BX <-- mouse mode
  *)

TYPE
  KybdConfigType = (userConf,
                    defaultConf,
                    reservedConf,
                    scanCodeConf
                   );

  Accel  = ARRAY [0..127] OF CHAR;
  AccelPointer = POINTER TO Accel;

  ButtonStr = ARRAY [0..30] OF CHAR;
  ButtonStrPointer = POINTER TO ButtonStr;
```

```
    MoveStr = ARRAY [0..3] OF CHAR;
    MoveStrPointer = POINTER TO MoveStr;

    KybdParam = RECORD
                    LPButStrLength:     CARDINAL;
                    LPButStrPtr:        ButtonStrPointer;
                    LRButStrLength:     CARDINAL;
                    LRButStrPtr:        ButtonStrPointer;
                    MPButStrLength:     CARDINAL;
                    MPButStrPtr:        ButtonStrPointer;
                    MRButStrLength:     CARDINAL;
                    MRButStrPtr:        ButtonStrPointer;
                    RPButStrLength:     CARDINAL;
                    RPButStrPtr:        ButtonStrPointer;
                    RRButStrLength:     CARDINAL;
                    RRButStrPtr:        ButtonStrPointer;
                    UpMovStrLength:     CARDINAL;
                    UpMovStrPtr   :     MoveStrPointer;
                    DownMovStrLength:   CARDINAL;
                    DownMovStrPtr   :   MoveStrPointer;
                    LeftMovStrLength:   CARDINAL;
                    LeftMovStrPtr   :   MoveStrPointer;
                    RightMovStrLength:  CARDINAL;
                    RightMovStrPtr   :  MoveStrPointer;
                    xscale           :  CARDINAL;
                    yscale           :  CARDINAL;
                  END;

  PROCEDURE GetKeyboardConfig(VAR param: KybdParam);
    (* Logitech Mouse Driver System Call 34
          Input : AX = 34 System Call 34
                  ES:DX <-- address of parameter
    *)
```

```
PROCEDURE SetKeyboardConfig(config: KybdConfigType;
                            param: KybdParam);
  (* Logitech Mouse Driver System Call 35
       Input : AX = 35 System Call 35
               BX    <-- configuration type
               ES:DX <-- address of parameter
  *)

(* the following part is only valid for serial mice
   using a RS232 interface
*)

TYPE
  RS232Param = RECORD
                 BaudRate          : CARDINAL;
                 Protocol          : CARDINAL;
                 ReportRate        : CARDINAL;
                 RevisionNumber    : CARDINAL;
                 AutoBaud          : BOOLEAN;
                 dummy             : BOOLEAN;
                 (* for alignment reasons *)
                 CommunicationPort: CARDINAL;
               END;

PROCEDURE GetRS232Config(VAR param: RS232Param);
  (* Logitech Mouse Driver System Call 36
       Input : AX = 36 System Call 36
               ES:DX <-- address of parameter
  *)
  PROCEDURE SetRS232Config(param: RS232Param);
    (* Logitech Mouse Driver System Call 37
          Input : AX = 37 System Call 37
                  ES:DX <-- address of parameter
    *)

END LogiMouse.
```

```
DEFINITION MODULE MathLib0;
(*
   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.
```

```
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,

    SetMickeysPerPixel,

    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: Button;
                     VAR buttonStatus: ButtonSet;
                     VAR buttonPressCount: CARDINAL;
                     VAR 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: Button;
                    VAR buttonStatus: ButtonSet;
                    VAR buttonReleaseCount: CARDINAL;
                    VAR 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 <-- treshold 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 ('+') preceeding the number is always
     accepted, a minus sign ('-') is only accepted when
     converting to INTEGER.
   - 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 charatcers 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, StringToNum,
   CardToString, IntToString, 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
       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   the base of the number represented in
              the string

out:   num    converted number
       done   TRUE if successful conversion,
              FALSE  or 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 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 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.
```

```
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;
                             VAR name:       ARRAY OF CHAR;
                             VAR term:       Termination;
                             acceptOption:   BOOLEAN;
                             VAR readInName: 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 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 explizitly 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 initilization of s, Awaited(s) is FALSE.
*)


END Processes.
```

```
DEFINITION MODULE ProgMessage;
(*
   Write program status message to the terminal
*)


FROM System IMPORT Status;

EXPORT QUALIFIED WriteStatus;


PROCEDURE WriteStatus (st: Status);
(*
- Write a short description of a program status on the
  terminal.

in:     st      a Status, as returned by Program.Call

The message may be up to 32 characters long.
*)


END ProgMessage.
```

```
DEFINITION MODULE Program;
(*
   Subprogram loading and execution

Under MODULA-2/86, programs can be divided into
subprograms (we call them 'programs') which are
loaded upon request.

These programs are executed like procedures:

   - They have only one entry-point (module code of
     the program's main module).
   - After termination, their data do not exist any
     longer. The of the program also disappears, and
     will be reloaded from disk upon the next activation.
   - Programs may themselves activate other programs.

Derived from the Lilith Modula-2 system developed by the
group of Prof. N. Wirth at ETH Zurich, Switzerland.
*)


FROM System IMPORT Status;

EXPORT QUALIFIED
   Call, GetErrorInfo;


PROCEDURE GetErrorInfo (VAR msg: ARRAY OF CHAR);
(*
- Obtain more information about a load error.

out:  msg           a string related to the last error.
```

```
After Call (below) has returned a Status value of
'modulenotfound' and 'incompatiblemodules', GetErrorInfo
will return the name of the offending module. (length is
up to 24 characters). It returns an empty string in all
other cases.
*)


PROCEDURE Call (programName: ARRAY OF CHAR;
                shared: BOOLEAN;
                VAR st: Status);
(*
- Load and execute a (sub) program.

in:   programName   file specification for the program,
      shared        whether to share resources,

out:  st            terminating status of the subprogram.

The file whose name is given in 'programName' is opened
loaded, and started.  There is no default device or file
type; these must be supplied by the caller.  The file must
contain a linked, relocatable Modula-2 (sub)program.

The load address is defined by the default allocation
schema, in which programs are loaded on top of stack and
a new stack is created for execution of the new program.

If 'shared' = TRUE then all sharable resources allocated
by the called program are owned by the calling program
(or possibly the caller of the caller...). Shared
resources are not released upon termination of the new
program.
```

```
Upon termination of the program, its memory is freed and
the old stack is established.  All the resources used by
a terminating program are released, if they are not shared
and if they have not been released explicitly by the
program (files, heap, etc).

Any value of 'st' other than 'normal' indicates an
abnormal termination of the subprogram. In some cases
GetErrorInfo (above) will provide additional details.

Cautions:

In case of abnormal termination, Call does NOT print any
kind of error message.

Do not assign a procedure in the current program to a
procedure variable which could still exist after the
current program terminates (for example, a variable in a
shared resource or in the calling program). When the
current program terminates, all procedures in it must be
considered to cease to exist.

The loader in this module is not reentrant. This means
that interrupt processes must not load overlays!
*)


END Program.
```

```
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, 100E0, 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.
```

```
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). Charcters
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
unmasks 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 = 0C, 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;
(*
   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
unmasks 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 = 0C, 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.
```

```
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 = 0C, 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.
```

```
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'.

Derived from the Lilith Modula-2 system developed by the
group of Prof. N. Wirth at ETH Zurich, Switzerland.
*)


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 (0FFF0H)
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.
```

```
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 SYSTEM;
(*
 The interface of the standard module SYSTEM cannot be
 described completely with a regular Modula-2 definition
 module. Module SYSTEM offers some features which expand
 the language itself. Therefore, the following description
 does not comply everywhere with the syntactic and semantic
 rules of Modula-2, and this pseudo definition module
 cannot be compiled. This description just intendeds to
 give a brief overview of the capabilities provided by
 the standard module SYSTEM in MODULA-2/86.

 For additional information please refer to the
 corresponding section of the MODULA-2/86 USER'S MANUAL and
 the corresponding sections of the Modula-2 language report
 in 'Programming in Modula-2'.
*)

  EXPORT QUALIFIED
    AX, BX, CX, DX, SI, DI, ES, DS, CS, SS, BP, SP,
    RTSVECTOR,
    BYTE, WORD, ADDRESS, PROCESS,
    ADR, SIZE, TSIZE,
    NEWPROCESS, TRANSFER, IOTRANSFER, LISTEN,
    GETREG, SETREG,
    CODE, SWI, ENABLE, DISABLE,
    INBYTE, OUTBYTE, INWORD, OUTWORD,
    DOSCALL (* DOS only *), CPMCALL (* CP/M-86 only *);
```

```
CONST

  AX, CX, DX, BX, SP, BP, SI, DI, ES, CS, SS, DS;
  RTSVECTOR;
  (*
   Programs should refer to these registers and to the
   RTS interrupt vector through these constants only.
   The values listed below should never be used directly
   in any program. The values used by the compiler are
   listed here for documentation purposes only:

   AX =  0;  CX =  1;  DX =  2;  BX =  3;
   SP =  4;  BP =  5;  SI =  6;  DI =  7;
   ES =  8;  CS =  9;  SS = 10;  DS = 11;

   RTSVECTOR = 228;
  *)


TYPE

  BYTE;
  WORD;
  (*
   Assignements and passing as parameters are the only
   operations possible with objects of type BYTE or WORD.
  *)

  ADDRESS = POINTER TO WORD;
  ADDRESS = RECORD
              OFFSET, SEGMENT: CARDINAL;
            END;
  (*
   Type ADDRESS can be used as if these two different
   type declarations were existing at the same time.
  *)
```

```
PROCESS;
(*
 Type PROCESS is used for process handling.
*)


PROCEDURE ADR (AnyVariable): ADDRESS;
(*
 Return the ADDRESS of the variable.
*)

PROCEDURE SIZE (AnyVariable): CARDINAL;
(*
 Return the size of the variable.
*)

PROCEDURE TSIZE (AnyType): CARDINAL;
PROCEDURE TSIZE (AnyVariantRecordType,
                 Tag1Const, Tag2Const, ... ): CARDINAL;
(*
 Return the size of a variable of this type.
*)

PROCEDURE NEWPROCESS (ProcessBody: PROC;
                      WorkspaceAddr: ADDRESS;
                      WorkspaceSize: CARDINAL;
                      VAR Process: PROCESS);
(*
 Create a new PROCESS.
*)

PROCEDURE TRANSFER (VAR FromProcess: PROCESS;
                    VAR ToProcess: PROCESS);
(*
 Transfer control from one PROCESS to another PROCESS.
*)
```

```
PROCEDURE IOTRANSFER (VAR InteruptHandler: PROCESS;
                      VAR InterruptedProcess: PROCESS;
                      InterruptVectorNumber: CARDINAL);
(*
 Set up an interrupt PROCESS and transfer control.
*)

PROCEDURE LISTEN;
(*
 Lower the priority temporarily.
*)

PROCEDURE GETREG (Register: CARDINAL;
                  VAR Value: BYTEorWORD);
(*
 Get a value from a register.
*)

PROCEDURE SETREG (Register: CARDINAL;
                  Value: BYTEorWORD);
(*
 Put a value into a register.
*)

PROCEDURE CODE (Code1Const, Code2Const, ... : BYTE);
(*
 Insert (binary) machine instructions into the code.
*)

PROCEDURE SWI (InterruptVectorNumber: CARDINAL);
(*
 Issue a software interrupt.
*)
```

```
    PROCEDURE ENABLE;
    PROCEDURE DISABLE;
    (*
     Enable or disable interrupts.
    *)

    PROCEDURE INBYTE  (Port: CARDINAL;
                       VAR Value: BYTEorWORD);
    PROCEDURE OUTBYTE (Port: CADINAL;
                       Value: BYTEorWORD);
    (*
     Get or put a byte value from or to an I/O port.
    *)

    PROCEDURE INWORD  (Port: CARDINAL; VAR Value: WORD);
    PROCEDURE OUTWORD (Port: CARDINAL; Value: WORD);
    (*
     Get or put a word value from or to an I/O port.
    *)

    PROCEDURE DOSCALL (FunctionNumber: CARDINAL; ... );
    (*
     DOS only: Activate a DOS function.
    *)

    PROCEDURE CPMCALL (FunctionNumber: CARDINAL; ... );
    (*
     CP/M-86 only: Activate a CP/M-86 function.
    *)

END SYSTEM.
```

```
DEFINITION MODULE System;
(*
   Additional system dependent facilities

This module may be seen as an extension of the standard
module SYSTEM.

Derived from the Lilith Modula-2 system developed by the
group of Prof. N. Wirth at ETH Zurich, Switzerland.
*)


FROM SYSTEM IMPORT ADDRESS, PROCESS;

EXPORT QUALIFIED
   Status, Terminate,
   ProcessDescriptor, ProcessPtr, curProcess,
   TermProcedure, CallTermProc,
   InitProcedure, CallInitProc,

   (* The following constants define the processor's
      registers. These constants are exported here for
      reasons of compatibility with older versions and
      they should not be used any more. Instead, the
      corresponding constants (AX, BX, etc) from the
      standard module SYSTEM should be used.
   *)
   RegAX, RegBX, RegCX, RegDX, RegSI, RegDI,
   RegES, RegDS, RegCS, RegSS, RegBP, RegSP;
```

```
TYPE
   Status =
      (normal, warned,
       (*
        - considered non-fatal; no dump is produced for
          the first two cases
       *)
       stopped, asserted, halted,
       caseerr, stackovf, heapovf,
       functionerr, addressoverflow,
       realoverflow, realunderflow, badoperand,
       cardinaloverflow, integeroverflow,
       rangeerr, dividebyzero,
       coroutineend,
       loaderr, callerr, programnotfound,
       modulenotfound, incompatiblemodule,
       filestructureerr, illegalinstr,
       RTSfunctionerr, interrupterr);
   (*
   - This type defines the possible values for a program's
     status. The meaning of these values can be printed to
     the terminal by means of ProgMessage.WriteStatus .
   *)


PROCEDURE Terminate (st: Status);
(*
- Terminate the current (sub) program.

in:     st      terminating status.

If the value of 'st' is different from 'normal' or
'warned', memory is dumped on the disk file MEMORY.PMD,
which can be used for subsequent debugging. The value of
'st' will be returned to the caller of the terminating
program by means of the parameter 'st' of the procedure
'Program.Call'.
```

```
This procedure never returns to the caller.
*)

(*$A+*)
(* Make sure, that the following declaration is compiled
   with alignment, in order to allocate a dummy byte
   after the field 'status'.
*)
TYPE
   ProcessDescriptor =
      RECORD
        AX, CX, DX, BX   : CARDINAL;
        SP, BP, SI, DI   : CARDINAL;
        DS, SS, ES, CS   : CARDINAL;
        IP               : CARDINAL;
        flags            : BITSET;
        (*
        - all the above fields denote the values of the
          processor's registers for the process
        *)
        status           : Status;
        (*- status of the process *)
        programId        : CARDINAL;
        (*
        - identifier of the current program, incremented
          for every subprogram layer
        *)
        auxId            : CARDINAL;
        (*- currently not used *)
        sharedId         : CARDINAL;
        (*
- program identifier of the last subprogram layer
  which was called with parameter 'shared' = FALSE
  (see module 'Program')
*)
fatherProcess    : PROCESS;
(*
- process which created this process (by means
  of NEWPROCESS)
*)
unused           : CARDINAL;
(*- currently not used *)
interruptMask    : BITSET;
(*
- priority mask effective while this process is
  running. The mask register of the interrupt
  controller is set to the logical OR of this
  priority mask and of the device mask kept by
  the MODULA-2/86 run-time support.
*)
debugStatus      : CARDINAL;
(*- auxiliary status field used for the debugger *)
progEndStack     : ADDRESS;
(*
- value of (SS,SP), used by the system for
  aborting a program.
*)
intVector        : CARDINAL;
(*
- interrupt vector used by this process if it is
  an interrupt handler.
*)
oldISR           : ADDRESS;
(*
- old value of interrupt vector if this process
  is an interrupt handler.
*)
interruptedProcess : ADDRESS;
(*- used by IOTRANSFER *)
```

```
        heapBase          : ADDRESS;
        (*
        - address of heap base, with the header of the
          free-list.
        *)
        heapTop           : ADDRESS;
        (*
        - address of first free byte after the last
          allocated area on the heap.
        *)
        modTable          : ADDRESS;
        (*
        - points to the last element in a list with
          module descriptors.
        *)
      END;
(*$A=*)
(* reset alignment option *)


TYPE
   ProcessPtr = POINTER TO POINTER TO ProcessDescriptor;


VAR
   curProcess: ProcessPtr;
   (*
   - Points to a pointer, which in turn points at any
     moment to the workspace of the current process's.
     This variable is 'read-only' and should not be used
     in application programs.

   WARNING: Improper use of this variable may cause
   unpredictable behaviour of the system.
   *)
```

```
PROCEDURE TermProcedure (p: PROC);
(*
- Declare a termination routine.

in:     p       termination procedure.

The procedure 'p' will be called upon termination of the
current program or subprogram. Typical use is for drivers,
which have to release resources used by the terminating
program. Up to 20 termination routines can be installed.
*)


PROCEDURE CallTermProc;
(*
- Call all termination procedures for the current
  program.

Calls all procedures declared with 'TermProcedure' in the
current program. 'CallTermProc' is automatically called
at the termination of a program or subprogram.
*)


PROCEDURE InitProcedure (p: PROC);
(*
- Declare an initialization routine.

in:     p       initialization procedure.

Analoguous to 'TermProcedure', but for routines that have
to be called before execution of a program.
Up to 20 initialization routines can be installed.
*)
```

```
PROCEDURE CallInitProc;
(*
- Call all initialization procedures for the current
  program.

Analoguous to 'CallTermProc'.
*)


CONST
   RegAX = 0; RegCX = 1; RegDX =  2; RegBX =  3;
   RegSP = 4; RegBP = 5; RegSI =  6; RegDI =  7;
   RegES = 8; RegCS = 9; RegSS = 10; RegDS = 11;
   (*
   - These constants define the processor's registers.
     They are declared here for reasons of compatibility
     with older versions and they should not be used any
     more. Instead, the corresponding constants (AX, BX,
     etc) from the pseudo-module SYSTEM should be used.
     They may be used as parameters for the standard
     procedures SETREG and GETREG (except that SP, BP,
     CS, SS may not be used with SETREG).
   *)


END System.
```

```
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
     hes been entered.
   *)


TYPE
   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.
   *)
```

```
TYPE
   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 execution 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   (15C) = 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.
```

```
DEFINITION MODULE TimeDate;
(*
   Access to the system's date and time
*)


EXPORT QUALIFIED
   SetTime, GetTime, Time;


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.
*)


END TimeDate.
```

INDEX

INDEX OF PROCEDURES OF LIBRARY MODULES

in alphabetical order by procedure name

AddDec	(Decimals)
ADR	(SYSTEM)
Again	(FileSystem)
ALLOCATE	(Storage)
arctan	(MathLib0)
Assign	Strings)
AssignRead	(Termbase)
AssignWrite	(Termbase)
Available	(Storage)
Awaited	(Processes)
BusyRead	(RS232Code)
BusyRead	(RS232Int)
BusyRead	(RS232Polling)
Call	(Program)
CallInitProc	(System)
CallTermProc	(System)
CancelRedirection	(DOS31)
CardToString	(NumberConversion)
ChangeDirectory	(DiskDirectory)
Close	(FileSystem)
CloseInput	(InOut)
CloseOutput	(InOut)
CODE	(SYSTEM)
CompareDec	(Decimals)
CompareStr	(Strings)
Concat	(Strings)
ConditionalOff	(Mouse)
ConvertCardinal	(Conversions)
ConvertHex	(Conversions)
ConvertInteger	(Conversions)

LOGITECH MODULA-2
Software Development System

Read what Jerry Pournelle says of LOGITECH MODULA-2

"Modula-2 has been getting theoretical applause for years; what it has needed was a good low-cost compiler for a popular machine. LOGITECH has remedied that defect and is to be congratulated."

Byte
March, 1986

"...they'll do for Modula-2 what Borland's Turbo did for Pascal."

InfoWorld
January 13, 1986

LOGITECH

Logitech Inc.
805 Veterans Blvd., Redwood City, CA 94063
Telephone: 415/365-9852

Logitech SA
CH-1111 Romanel/Morges
Switzerland
41-21-87-96-56

Algol Logitech Spa
Via Durazzo 2
20134 Milano, MI
Italy
39-2-215-5622

Logitech Far East Ltd.
15 R&D 2
Science Based Industrial Park
Hsinchu, Taiwan, ROC
886-3-577-8241