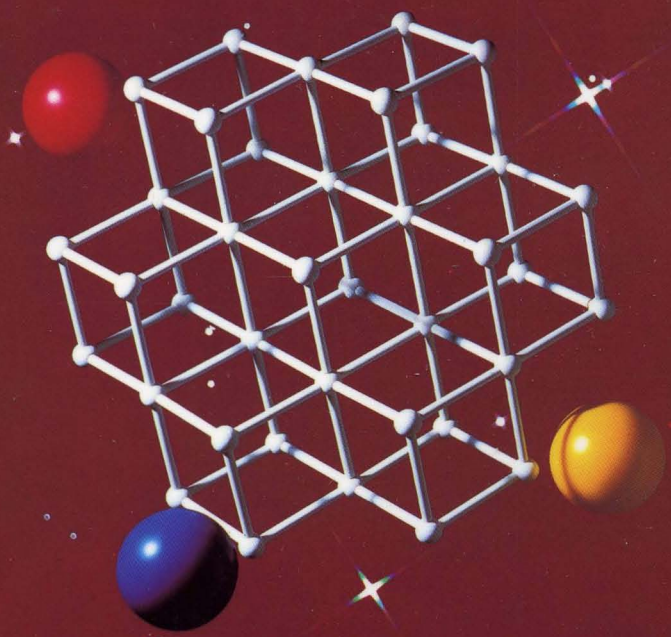
LOGITECH™ MODULA-2 VERSION 3.0



TOOLKIT

LOGITECH™ MODULA-2

Version 3.0

TOOLKIT

This edition applies to ***LOGITECH Modula-2, Version 3.00.***

Document Number	LU-UD-0010-1
First Edition	August 1987
First Printed	August, 1987

Trademarks

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

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

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

Intel is a registered trademark of **Intel Corporation.**

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

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

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

PFIXPLUS is a trademark of **Phoenix Software Associates, LTD.**

Olivetti is a registered trademark of **Olivetti.**

COMPAQ is a registered trademark of **Compaq Computer Corporation.**

LOGITECH SOFTWARE LICENSE AGREEMENT

THIS DOCUMENT IS A LEGAL AGREEMENT BETWEEN YOU, THE LICENSEE, AND LOGITECH, INC ("LOGITECH"). BY USING THIS PROGRAM, YOU ARE AGREEING TO BECOME BOUND BY THE TERMS OF THIS AGREEMENT. IF YOU DO NOT AGREE TO THE TERMS OF THIS AGREEMENT, PROMPTLY RETURN THE DISK PACKAGE AND THE OTHER ITEMS THAT ARE PART OF THIS PRODUCT IN THEIR ORIGINAL PACKAGE, WITH YOUR PAYMENT RECEIPT (THE "RECEIPT"), TO LOGITECH FOR A FULL REFUND.

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

As the only warranty under this Agreement, and in the absence of accident, abuse or misapplication, LOGITECH warrants, to the original Licensee only, that the disk(s) on which the Software is recorded is free from defects in materials and workmanship under normal use and service for a period of ninety (90) days from the date of payment as evidenced by a copy of the Receipt. LOGITECH'S only obligation under this Agreement is, at LOGITECH'S option, to either (a) return payment as evidenced by a copy of the Receipt or (b) replace the disk that does not meet LOGITECH'S limited warranty and which is returned to LOGITECH with a copy of the Receipt. THIS WARRANTY GIVES YOU LIMITED, SPECIFIC LEGAL RIGHTS. YOU MAY HAVE OTHER RIGHTS, WHICH VARY FROM STATE TO STATE.

THE SOFTWARE AND ACCOMPANYING WRITTEN MATERIALS (INCLUDING THE USER'S MANUAL) ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND INCLUDING THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, EVEN IF LOGITECH HAS BEEN ADVISED OF THAT PURPOSE. LOGITECH SPECIFICALLY DOES NOT WARRANT THAT THE OPERATION OF THE SOFTWARE WILL BE LIABLE FOR ANY DIRECT, INDIRECT, CONSEQUENTIAL OR INCIDENTAL DAMAGES ARISING OUT OF THE USE OF OR INABILITY TO USE SUCH PRODUCT EVEN IF LOGITECH HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. SOME STATES DO NOT ALLOW THE EXCLUSION OR LIMITATION OF LIABILITY FOR CONSEQUENTIAL OR INCIDENTAL DAMAGES, SO THE ABOVE LIMITATION MAY NOT APPLY.

Table of Contents

Chapter 1
The LOGITECH M2FORMAT Utility 17

Chapter 2
The LOGITECH Linker 57

Chapter 3
The Symbolic Run-Time Debugger 83

TOOLKIT

Introduction

The *LOGITECH Modula-2 Toolkit* is a collection of utilities which streamline and simplify programing and maintaining *Modula-2* programs.

The LOGITECH M2FORMAT Utility

Automatically determines the style features for formatting *Modula-2* source files. It is an intelligent style formatter in the sense that a template file is used to determine the style feature for formatting. You can easily modify this template file to reflect your style preferences.

The LOGITECH Linker

Combines all the separately compiled modules into a single executable file. It takes the object (.OBJ) files of the modules to be linked as input and produces an executable (.EXE or .OVL) file, and a map (.MAP) file as output.

The LOGITECH Symbolic Run-Time Debugger

Lets you monitor the execution of a program. You can run the program in steps. After each step, you can inspect the data and current status of the program. You can modify the values of the variables the program uses.

The LOGITECH M2DECODE Utility

Decodes *LOGITECH Modula-2* .OBJ files into text files with information such as disassembled code or data.

The LOGITECH M2VERS Utility

Helps keep track of different versions of a program.

The LOGITECH M2XREF Utility

Generates cross reference information tables of text files, especially of ***Modula-2*** source files.

The LOGITECH M2MAKE Utility

Builds batch files that compile the minimum number of modules necessary in the correct order.

The LOGITECH M2CHECK Utility

Analyzes a ***Modula-2*** source file and generates a listing file which can be used to avoid common programming errors.

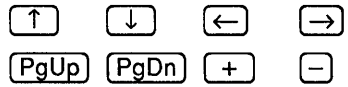
How to Read This Manual

The following conventions are used in this manual:

Keys to be pressed, look like this:

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

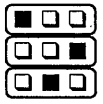
Keys from the **Numeric Keypad** are shown like this:



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

M2L ↵

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

 means press the left mouse button,
 means press the right mouse button, and
 means press the middle mouse button.

( means press both buttons on a two button mouse.)

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

<Application name> ↵

File names look like this:

M2L.EXE

DOS commands and statements look like this:

PATH, COPY

Product names look like this:

MS DOS, LOGITECH Modula 2

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

PROCESS, VAL, MyFunction

These are not emphasized in screen display or program listings.

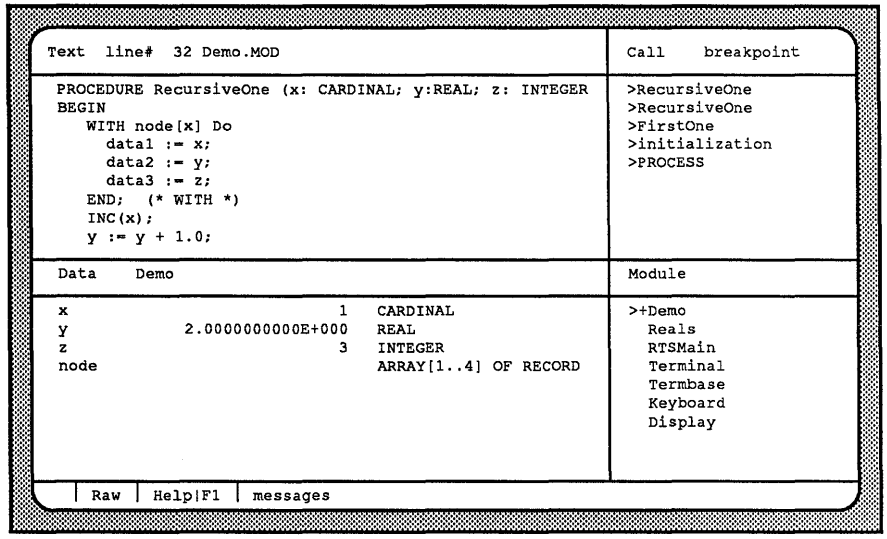
Screen output and some listings look like this:

```
Program Not Found
```

Program source code looks like this:

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

Sample Screens look like this:



Other LOGITECH Products

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

LOGITECH Modula-2

The ***LOGITECH Modula-2*** Development System provides today's programmers with the most powerful development environment available for the PC. In addition to the tools and utilities described in the ***LOGITECH Modula-2 User's Manual*** and in the ***LOGITECH Modula-2 Toolkit***, LOGITECH offers the following:

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

Site licences are available for all of the above.

The **LOGITECH C7 Mouse**

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

The **LOGITECH Bus Mouse**

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

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

Notes:

Installation

This chapter tells you how to install the *LOGITECH Modula-2 Toolkit* under *DOS*. Remember: before you attempt to install the *Toolkit*, you should have already installed the system described in the *LOGITECH Modula-2 User's Manual.*

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

Instructions are given for installation to a set of floppy diskettes, as well as to a hard disk.

NOTE

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

What do you need?

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

- A floppy disk **drive A**, and either —
  - A floppy disk **drive B**, or
  - A hard disk **drive C**.
- **320 K** of RAM, **640 K** recommended.

What have you purchased?

Manuals

The instructions you are reading are in the *LOGITECH Modula-2 Toolkit Manual.*

Diskettes

These four diskettes come with the *LOGITECH Modula-2 Toolkit*:

Disk 1:	**Standard Library Sources I** **Utilities I**
Disk 2:	**Standard Library Sources II**
Disk 3:	**Linker** **Utilities II**
Disk 4:	**Run-Time Debugger**

The following files are on these diskettes:

Disk 1: Standard Library Sources I
Utilities I:

ASM.ARC	.ASM Files of the Library
RTS.ARC	Sources for the M2RTS.LIB
UTILS1.ARC	First part of the utilities
Installation batch files	

Disk 2: Standard Library Sources II:

DEF.ARC	.DEF Files of the Library
MOD1.ARC	.MOD Files of the Library
MOD2.ARC	.MOD Files of the Library

Disk 3: Linker
Utilities II:

M2L.ARC	***LOGITECH Linker***
UTILS2.ARC	Second part of the utilities

Disk 4: Run Time Debugger :

RTD.ARC	***LOGITECH Symbolic Run Time Debugger (RTD)***

Most of the files in the ***LOGITECH Modula-2 Toolkit*** have been archived (packed) into smaller files (those with the extension .ARC). The ***ARC*** utility allows you to unarchive (unpack) them into your working disk.

The LOGITECH Modula-2 Toolkit on Floppy Diskettes

NOTE

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

1) Put Write-Protect tabs on all your ***LOGITECH Modula-2*** diskettes, and

2) Use the **DISKCOPY** and **DISKCOMP** commands from your ***DOS*** files to back up your diskettes. Then put your original diskettes in an archival area and use the copies for all installation.

3) Prepare formatted diskettes with readable labeling, before you copy the the files in the Installation procedure which follows.

Installing on Floppy Diskettes

To use the ***LOGITECH Modula-2 Toolkit*** on a dual floppy system, examine the contents of the diskettes you received and become familiar with the files on these diskettes.

To see the contents of an archived file (extension .ARC), use the executable file ARC.EXE on **Disk 1**. For example, to list the contents of the .ARC file UTILS2.ARC, insert the corresponding Disk (i.e., **Disk 3**) into **drive B**; then insert **Disk 1** (with the **ARC** utility) into **drive A** and type:

A:ARC -L B:UTILS2.ARC ↵

Use the following procedures to prepare your working diskettes for a floppy disk environment:

Step 1: Prepare Working Diskettes

Label and format nine (**9**) diskettes with labels that reflect their contents. We suggest:

YourDisk **1**	M2MAKE	**Refer to Chapter 7**
	M2CHECK	**Refer to Chapter 8**
YourDisk **2**	M2FORMAT	**Refer to Chapter 1**
	M2DECODE	**Refer to Chapter 4**
	M2VERS	**Refer to Chapter 5**
	M2XREF	**Refer to Chapter 6**
YourDisk **3**	M2L	**Refer to Chapter 2**
YourDisk **4**	RTD	**Refer to Chapter 3**
YourDisk **5**	DEF	Standard Library Definition Files
YourDisk **6**	MOD1	Standard Implementation Files I
YourDisk **7**	MOD2	Standard Implementation Files II
YourDisk **8**	ASM	Standard Library Assembly Files
YourDisk **9**	RTS	Run Time Support Source Files

Step 2: Copy ARC.EXE to each Working Diskette.

Insert the copy of **LOGITECH** Disk **1** in drive **A** and one of your blank, labeled diskettes in drive **B**. Type,

COPY A:ARC.EXE B: ↵

Insert another blank labeled diskette in drive **B** and repeat the above instruction until all diskettes have a copy of ARC.EXE.

Steps **3** through **11** detail a procedure for "unARCing" the files from a source diskette to a standard **360 K** working diskette.

If you have high density diskettes, you can optimize your ***Modula-2*** environment by combining the contents of several diskettes to a single high density diskette.

Step 3: Copy compressed files to YourDisk 1

Insert ***LOGITECH Toolkit Disk 1*** in drive **A**. Insert YourDisk **1** in drive **B**. Type,

B:ARC -E A:UTILS1.ARC B:*.* ↵

Step 4: Copy compressed files to YourDisk 2

Insert ***LOGITECH Toolkit Disk 3*** in drive **A**. Insert YourDisk **2** in drive **B**. Type,

B:ARC -E A:UTILS2.ARC B:*.* ↵

Step 5: Copy compressed files to YourDisk 3

Insert ***LOGITECH Toolkit Disk 3*** in drive **A**. Insert YourDisk **3** in drive **B**. Type,

B:ARC -E A:M2L.ARC B:*.* ↵

Step 6: Copy compressed files to YourDisk 4

Insert ***LOGITECH Toolkit Disk 4*** in drive **A**. Insert YourDisk **4** in drive **B**. Type,

B:ARC -E A:RTD.ARC B:*.* ↵

Please also copy all the .CFG files from the ***PMD Disk*** to YourDisk **4**.

Step 7: Copy compressed files to YourDisk 5

Insert ***LOGITECH Toolkit Disk 2*** in drive **A**. Insert YourDisk **5** in drive **B**. Type,

B:ARC -E A:DEF.ARC B:*.* ↵

Step 8: Copy compressed files to YourDisk 6

Insert ***LOGITECH Toolkit Disk 2*** in drive **A**. Insert YourDisk **6** in drive **B**. Type,

B:ARC -E A:MOD1.ARC B:*.* ↵

Step 9: Copy compressed files to YourDisk 7

Insert ***LOGITECH Toolkit Disk 2*** in drive **A**. Insert YourDisk **7** in drive **B**. Type,

B:ARC -E A:MOD2.ARC B:*.* ↵

Step 10: Copy compressed files to YourDisk 8

Insert ***LOGITECH Toolkit Disk 1*** in drive **A**. Insert YourDisk **8** in drive **B**. Type,

B:ARC -E A:ASM.ARC B:*.* ↵

Step 11: Copy compressed files to YourDisk 9

Insert ***LOGITECH Toolkit Disk 1*** in drive **A**. Insert YourDisk **9** in drive **B**. Type,

B:ARC -E A:RTS.ARC B:*.* ↵

Step 12 Review

The disks we have called YourDisk **1-9** are now ready to use.

You may now delete ARC.EXE from each diskette if you wish, since it is not used further by ***LOGITECH Modula-2***.

Running the Toolkit on a Dual Floppy System

When you work with ***LOGITECH Modula-2*** on floppy diskettes:

Drive A holds your ***Modula-2 Working diskette***.

It should contain:

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

Drive B holds a ***LOGITECH Modula-2*** utility diskettes:

To Compile
A working copy of the ***LOGITECH Compiler*** is in drive **B**.

To Link
the ***LOGITECH Linker*** is in drive **B** (e.g. YourDisk **3**).
A working copy of the **Library disk** is swapped in drive **B** after you start the ***Linker***. You start with the ***Linker*** diskette in drive **B**, when the ***Linker*** prompts for **masterfile**, swap the ***Linker*** diskette in drive **B** with the Library diskette and proceed.

To Run RTD
The ***LOGITECH Debugger*** disk is in drive **B** (e.g. YourDisk **4**).

To Run Utilities
One of the two ***LOGITECH Programming Utilities*** disks is in drive **B**, depending on which utility you want to use (e.g. YourDisk **1** or **2**).

NOTE

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

If you are using high density diskettes, study the following section on hard disk systems, on the environment variables used by ***LOGITECH Modula-2***, and also study the **Section 9.1, Library Search Strategy** in the ***LOGITECH Modula-2 User's Manual.***

Hard Disk Systems

If you used the automatic installation program when you installed the base software described in the ***LOGITECH Modula-2 User's Manual***, you will be able to use it again here to install the ***LOGITECH Modula-2 Toolkit*** software. Make sure you are on the hard disk and in the directory where you previously defined your ***Modula-2*** system. Then, with your working copy of **Toolkit Disk 1** in drive A, type:

A:INSTALL \YOUR_DIR ↵

This creates additional directories as needed and copies the necessary files from **Disk 1** to the ***LOGITECH Modula-2*** area on your hard disk. You are then prompted for successive disks as needed.

If your system has special constraints, such as directory names that conflict with those used by ***LOGITECH Modula-2***, then you must install step-by-step.

Step 1: Add Additional Directories.

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

CD \YOUR_DIR ↵

MD \M2LIB\ASM ↵
MD \M2LIB\RTS ↵

Step 2: Install Disks.

- Insert **Disk 1** in **Drive A**, and type:

```
COPY      A:ARC.EXE                    ↵

ARC -E    A:ASM.ARC    M2LIB\ASM\*.*   ↵
ARC -E    A:RTS.ARC    M2LIB\RTS\*.*   ↵
ARC -E    A:UTILS1.ARC M2EXE\*.*       ↵
```

- Insert **Disk 2** in **Drive A**, and type:

```
ARC -E    A:DEF.ARC    M2LIB\DEF\*.*   ↵
ARC -E    A:MOD1.ARC   M2LIB\MOD\*.*   ↵
ARC -E    A:MOD2.ARC   M2LIB\MOD\*.*   ↵
```

- Insert **Disk 3** in **Drive A**, and type:

```
ARC -E    A:M2L.ARC    M2EXE\*.*       ↵
ARC -E    A:UTILS2.ARC M2EXE\*.*       ↵
```

- Insert **Disk 4** in **Drive A**, and type:

```
ARC -E    A:RTD.ARC    M2EXE\*.*       ↵
```

Chapter 1
The LOGITECH M2FORMAT Utility

1.1 Introduction

LOGITECH M2FORMAT is an intelligent style formatter for any compilable *Modula-2* source files. It is intelligent in the sense that a template file automatically determines the style features for formatting. You can easily modify this template file to reflect your style preferences. *M2FORMAT* includes the template compiler and source file formatter.

M2FORMAT is easy to learn and easy to use.

You do not have to specify format parameters numerically. Instead *M2FORMAT* adapts the style you have used for the syntactical constructs of *Modula-2* in the template file when formatting these constructs in your code. The template file gives an example of the style. The style features consists of the number of lines and indentations between syntactic elements. The template compiler reads the template file and extracts the formatting data. Such data will then be used by *M2FORMAT*.

CAUTION

Compile or syntax check source files before attempting to reformat the file using ***M2FORMAT***. However, ***M2FORMAT*** will also identify syntax errors in the source file before exiting.

Each template file must be "compiled" just once by the template compiler before it is usable for formatting source files. The compiled template data may then be used repeatedly whenever the particular style in the template is desired. There are features such as spacing and newline, and a number of selectable style features that are specified by "comment commands", which let you select left and right margins, as well as options on how to handle comments and line overruns. Template file compilation and source file formatting are separate operations, so recompilation of template files is required only if changes in style are made in the template file.

Features of ***M2FORMAT*** include:

1. Intelligent determination of style features from a template file.
2. Style consistency check through template "compilation".
3. Detailed list of style inconsistencies.
4. Production of a template **<filename>**.TMD for use by format program.
5. Separate template compilation and formatting for increased speed and style enforcement.
6. Selection of multiple template files for formatting.
7. User options provided through **comment commands** in template or source file.
8. Output can be specified for hardcopy or as a compilable source file.
9. Intelligent handling of all ***Modula-2*** constructs, including comments.
10. Format any size file, limited only by disk space.

11. User selected source file may include **PATH**.
12. Line overruns identifiable by comments in the formatted output file so compilability is unaffected.
13. Select files by command line, prompt, or selection from directory file list.
14. Turn formatting or syntax checking off or on for selected portions of source file.
15. Default text file provides the name of a default template file, and a set of masks that filter the file names displayed by the file directory.
16. Hardcopy output lets you define attributes (printer escape codes) before and after keywords, identifiers, comments and procedure headings.
17. Distance is set from page bottom, so major constructs start on the next page.

Although ***LOGITECH M2FORMAT*** should be used as a final step in the software development life cycle to ensure consistency of style in the final software documentation, it can be used at any stage of the development process to enhance readability.

1.2 System Configuration and Getting Started

LOGITECH M2FORMAT runs under *PC-DOS* or *MS-DOS 2.x* or higher on any *IBM PC* or *IBM PC*-compatible computer with at least 256K of RAM.

Make backup copies of all the diskettes before use. The template file contains representations for all constructs in the *Modula-2* language. When modifying the template file for defining new style features, be careful that you don't delete any constructs. Before running *M2FORMAT*, set the environment variable **M2F** to the directory where *M2FORMAT* system files are kept (for example, **SET M2F=C:\M2EXE\FORMAT**).

NOTE

M2FORMAT requires the ANSI.SYS file for correct operation.

M2FORMAT includes:

TEMPL — The template file. Contains style features to be used by the *M2FORMAT* software. Every *Modula-2* construct is included, as well as formatting options that are selectable through **comment commands** (either in the template file or in your *Modula-2* source file). **Comment commands** in your source file take precedence over the default values in the template file.

M2FC.EXE — The template compiler. It verifies consistency of style features in the template file and generates information used by M2FORMAT. M2FC.EXE produces a template data file, **<filename>**.TMD.

M2FORMAT.EXE — The stand alone version of *M2FORMAT*. It formats a *Modula-2* source file, based on style features contained in a user-specified template file. The template you specify must have been compiled using M2FC.EXE, to produce the resulting *.TMD file. M2FORMAT.EXE produces a separate output file that contains the formatted source file. Formatted *Modula-2* source files are given the name **<filename>**.FMT for implementation and program modules and **<filename>**.FMD for definition modules.

PTM2FORM.EXE The ***POINT*** version of M2FORMAT. This version of ***M2FORMAT*** can be used as an extension of the "***LOGITECH POINT Editor***" version 1.5. (see the "***POINT Editor***" documentation for information on ***POINT*** extensions).

DEFAULT.M2F This file contains information used by M2FORMAT for determining the default template file and how to display directories for file selection. This file first contains the default template data filename on line one. Next are mask definitions, one on each line, for use by the directory option when running program M2FORMAT.EXE. The default template filename is TEMPL.TMD and the default filename masks for the directory option are *.MOD and *.DEF. These default filenames provide a listing of only those files that have .MOD or .DEF extensions when the directory is displayed by M2FORMAT. To add additional masks, write each on a separate line (with a limit of 10 masks). Put DEFAULT.M2F in the subdirectory that is set by the environment variable **M2F**.

PRINTER.M2F Needed only for the hardcopy option. The first line contains the physical page length for hardcopy output. The second line contains the number of lines from bottom of page that causes major constructs such as procedure headings, compound statements (IF, CASE, WHILE, etc) to start on the next page of hardcopy. Then comes a set of printer escape code sequences which you can define. These are sent to the output file when comments, procedure headings, identifiers, and keywords are encountered. (See **Section 1.8.2, Configuring PRINTER.M2F** for a detailed discussion of the file.) Put PRINTER.M2F in the subdirectory that is set by the environment variable **M2F**.

The diagram in **Figure 1.1** shows how these files interact with your source files to produce formatted output files.

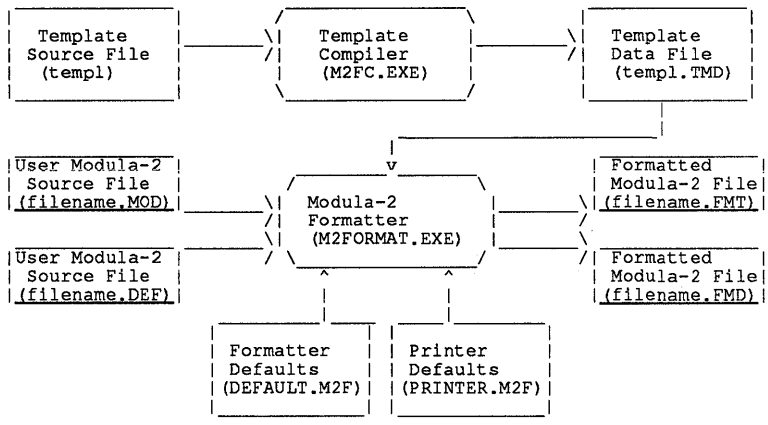


Figure 1-1 Model for *M2FORMAT* Software

1.3 Recommended Disk Organization

If you have a harddisk, create a separate directory that contains the template compiler (M2FC.EXE), the template source and data files (*.TMD), and the configuration files "PRINTER.M2F" and "DEFAULT.M2F". Set the environment variable "**M2F**" to this directory name. Place the ***LOGITECH M2FORMAT*** M2FORMAT.EXE file in a sub-directory that is contained in the **PATH.**

This organization keeps ***M2FORMAT*** out of any working directories and keeps the seldom used template compiler and its template files away from other common directories.

For floppydisk users, the compiler and template source files need not be on-line, once template data files are created. The ***M2FORMAT*** M2FORMAT.EXE program file, .TMD template data files, and the configuration files PRINTER.M2F and DEFAULT.M2F must be online when formatting.

1.4 Things you should know about M2FORMAT

LOGITECH M2FORMAT software is intended for use with *Modula-2* program, implementation, and definition modules that are complete and syntactically correct. Any attempt to use it with incorrect or incomplete program segments will cause *M2FORMAT* to report a syntax error and halt without formatting.

In addition to recognizing and formatting standard *Modula-2* statements, some basic syntax extensions have been added. These extensions are discussed in **Section 1.9, Syntax Extensions**. Other non-standard *Modula-2* extensions can be left unformatted by using the "syntax check off flag" as discussed in **Section 1.7, Formatting Options.**

TEMPL is a template file that can be modified to reflect new style features. Be careful not to delete any constructs in this file. Instead, change only the spacing, newline, or comment selectable features. Multiple template data files may be generated and used as input to M2FORMAT.EXE.

Comment handling is unique in *M2FORMAT*. In normal operation, comment text is packed with one space between words. The starting position of most comments is defined in the template file. Comments that exceed end of line are wrapped to the next line on a word boundary and are placed directly under the first word of the comment on the preceding line. To preserve comments such as tables with special spacing requirements, precede the comment with "format-off" `(*.f-*)`, and use "format-on" `(*.f+*)` after the comment.

Line wrap is handled in the following way by *M2FORMAT*. Formatted output lines that exceed the end of line are clipped on word boundaries and continued on the next line at the same indentation level as the preceding line. If the "comment command" `(*.m+*)` is enabled, all line wraps are flagged in the formatted output file with the comment `(*@*)` in column 1. These comment flags can be used to customize the formatting of each line wrap.

1.5 Using M2FORMAT

This section gives detailed information on how to use *LOGITECH M2FORMAT* software. It is divided into three parts that follow the three steps in formatting your source code:

1: Define a specific style.

2: Check for consistency in this defined style.

3: Reformat a compilable *Modula-2* source file.

A simple walk-thru example shows you how to use *M2FORMAT*. In this example, the style features provided in the standard template, TEMPL, are used without modification.

The following *Modula-2* program is used as an example. The template file show in **Figure 1-2** is significantly different from the final version listed in **Figure 1-8.**

```
MODULE CountExample;

FROM InOut IMPORT WriteString,WriteLn,WriteCard;

CONST
MaxCount = 10;
  LongerName = 11;

VAR Count : CARDINAL;

BEGIN
  FOR Count := 1 TO MaxCount DO
      WriteString ("Count = ");
      WriteCard (Count,3);
       WriteLn;
     END;
  END CountExample.
```

Figure 1-2 A Compilable *Modula-2* Program Listing

1.5.1 Modifying the Template File

For this walk-through the template file is used unmodified as provided in TEMPL. See **Section 1.4, Editing the Template File**, for information on how to modify the template file.

1.5.2 Compile the Template File

To run Template Compiler, type:

M2FC ↵

You will see the following screen display:

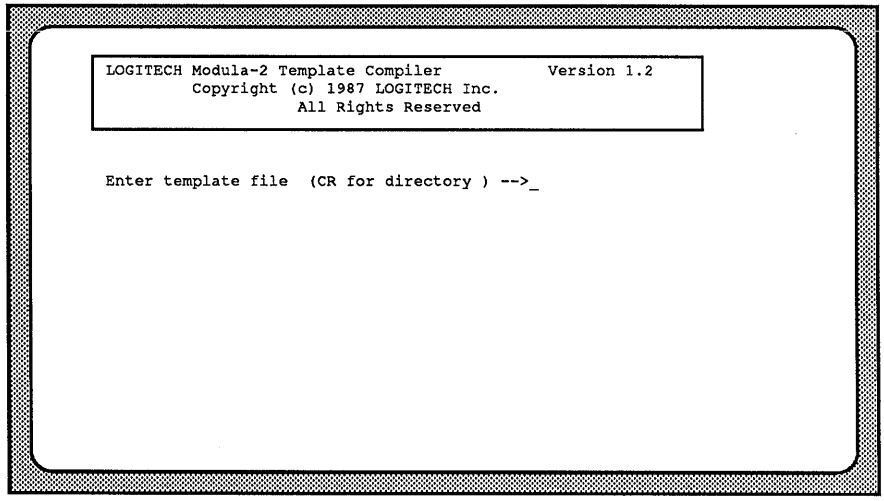


Figure 1-3 Screen Display Observed When Template Compiler Is Executed

Type the name of the template file:

<templatename> ↵ at the **Enter template file** prompt,

or press ↵ for a directory listing. TEMPL is the standard template file.

The directory option lists current directory files alphabetically, with the first file highlighted. Use the ← → ↑ ↓ keys to highlight the file you wish to select. Then press ↵.

Once the template file is selected, the template compiler begins its two-pass compilation of the template file. **Figure 1-4** shows the screen you see if no inconsistencies are encountered in the style features of the template file. The compiler produces an output file, **<templatename>.TMD**, or TEMPL.TMD for the standard template file. The .TMD file must be put in the subdirectory that is set by the environment variable **"M2F"**. A compiled template file must be present for ***LOGITECH M2FORMAT*** to execute correctly.

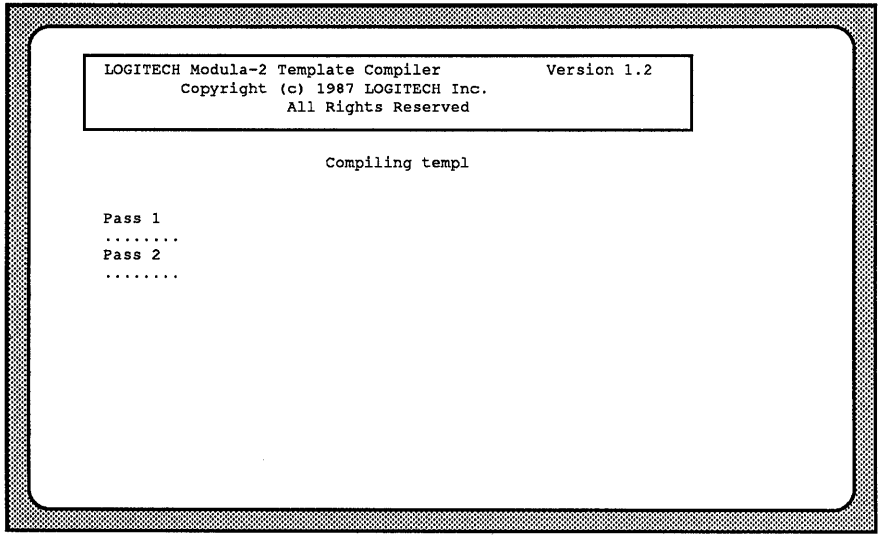


Figure 1-4 Screen Display After Successful Run Of Template Compiler

1.5.3 Format a Modula-2 Source File

To execute *LOGITECH M2FORMAT*, type:

M2FORMAT ↵

The display shown in **Figure 1-5** appears on the screen.

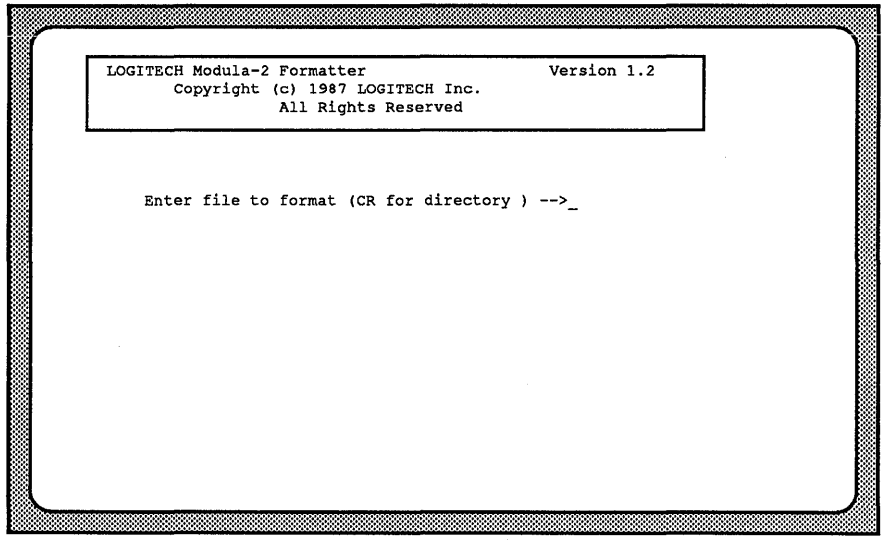


Figure 1-5 First Screen Display When M2FORMAT Is Executed

You may enter the **<filename.ext>** of the file you wish to format or ↵ for a directory listing. Use ←, ↑, →, ↓, keys to highlight the file you wish to select, then press ↵.

Only those filenames with the extensions specified in DEFAULT.M2F will appear in the directory listing (e.g. *.MOD, *.DEF). A filename may include a path designation. If so, then the reformatted file will be saved in the same path. The file to be formatted must not end in the suffix .FMT or .FMD or an error will be reported and the program terminated.

After you enter a valid source filename, the screen in **Figure 1-6** is displayed.

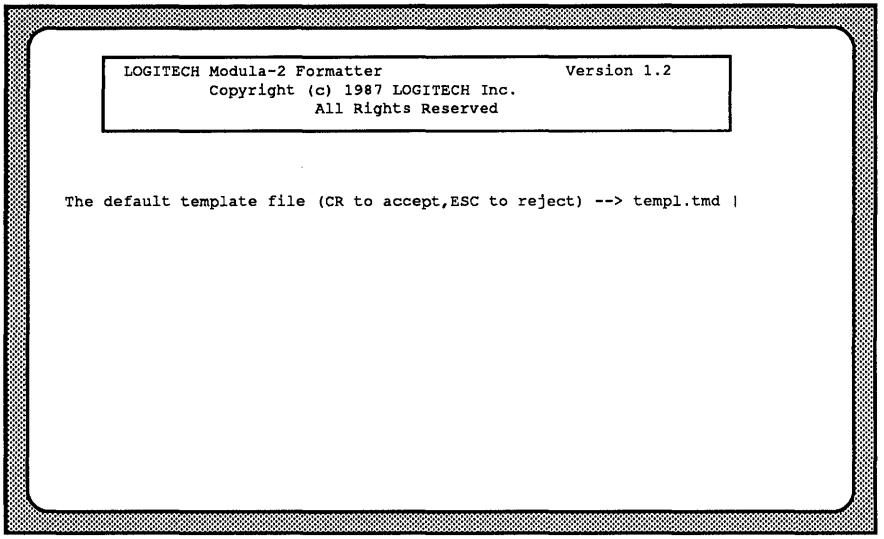


Figure 1-6 Second Screen Display When M2FORMAT Is Executed

You are prompted for the template file. Press [↵] to accept the default template file or [Esc] to list all the .TMD files (in the subdirectory set by the environment variable "**M2F**"). Selection is again done through highlighting and [↵]. If a DEFAULT.M2F file is not available in the subdirectory that is set by the environment variable "**M2F**", you may enter a filename without the .TMD extension, which is then assumed.

Once a valid compiled template file is selected, and if the formatting has proceeded with no errors, *LOGITECH M2FORMAT* displays the screen in **Figure 1-7** . The original source file is unchanged. The formatted file is saved as **<filename>**.FMT for implementation and program modules and **<filename>**.FMD for definition modules.

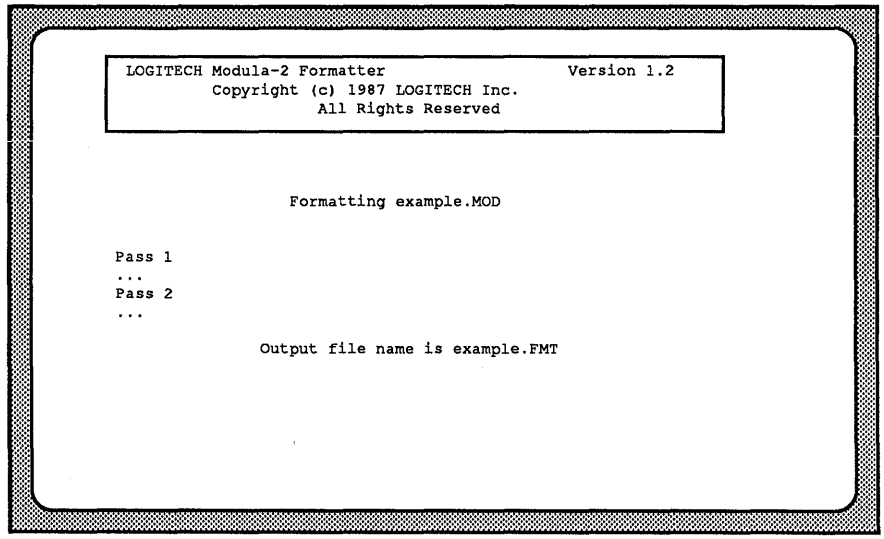


Figure 1-7 Screen Display After Successful Run Of M2FORMAT

The reformatted output in file **<example>**.FMT is shown in listing in **Figure 1-8**, consistent with the style features in the supplied standard template file TEMPL.

```
MODULE CountExample;

  FROM InOut IMPORT
    WriteString, WriteLn, WriteCard;

  CONST
    MaxCount   = 10;
    LongerName = 11;

  VAR
    Count : CARDINAL;

BEGIN
  FOR Count := 1 TO MaxCount DO
    WriteString( "Count = " );
    WriteCard( Count, 3 );
    WriteLn;
  END (* for Count *);
END CountExample.
```

Figure 1-8 Reformatted Example Program (Standard Style)

1.5.4 Specify Arguments To M2FORMAT In The Command Line

To pre-specify all arguments to *M2FORMAT* type:

M2FORMAT <filename.ext> <templatename> ↵

in which case the screens in **Figure 1-5** and **Figure 1-6** are skipped. <filename>.<ext> is formatted using **<templatename>** as template file. This is useful for batch command files as it lets you invoke ***M2FORMAT*** without waiting. If the extension is left off, ***M2FORMAT*** applies .MOD. If the template extension .TMD is left off, ***M2FORMAT*** assumes the extension .TMD.

If only one argument to ***M2FORMAT*** is specified, it is assumed to be the name of the file to be formatted. ***M2FORMAT*** will prompt for **<templatename>** by showing the **Screen Display in Figure 1-6.**

1.6 Editing the Template File

You can edit the template file TEMPL with any suitable text editor. Restrict your editing to horizontal and vertical spacing and comment commands. Do not delete constructs contained in the file.

CAUTION

Keep a backup copy of TEMPL.

1.6.1 Comment Commands

Comment commands or comment selected features are selected with predefined comments in either the template file or the ***Modula-2*** source file that is to be reformatted. Comment selected features in the ***Modula-2*** source file have precedence over the default features specified in the template file. The form of the comment command is (*.x*) with no spaces. The command may be in either upper or lower case. The following comment selectable features are provided. A complete discussion of comment commands is given in **Section 1.7, Formatting Options.**

Command	Description
(*.lx*)	Left Margin - x ranges from **0** to **15**. Default is **0**.
(*.rx*)	Right Margin - x ranges from **60** to **240**. Default is **80**.
(*.e+*)	End Comment On
(*.e-*)	End Comment Off (Default) When selected, comments are automatically placed one space after the reserved word **END** to identify the construct. If a user comment already follows **END**, the end comment is suppressed.
(*.m+*)	Line Overrun Marking On
(*.m-*)	Line Overrun Off (Default) When enabled, line overruns generated in formatted output file are marked with the symbol (*@*) in the first column.
(*.p+*)	Identifier Padding On (Default)
(*.p-*)	Identifier Padding Off If enabled, then identifiers in **VAR**, **CONST**, **TYPE** or **PROCEDURE** headings are padded with blanks so that " = " and " : " symbols are aligned.
(*.f+*)	Formatter Enable (Default)
(*.f-*)	Formatter Disable May be used in your source file to control which sections of the file are to be reformatted or left unchanged.
(*.s+*)	Syntax Checking Enable (Default)
(*.s-*)	Syntax Checking Disable May be used in your source file to control sections of the file that contain non-standard ***Modula-2*** that would normally cause a syntax error during formatting.
(*.h+*)	Hardcopy Enable
(*.h-*)	Hardcopy Disable (Default) When enabled, the output file is created with hardcopy features as defined in the printer configuration file PRINTER.M2F. The hardcopy option may only be selected in the template file.

Figure 1-9 Table Of Comment Selected Features

1.6.2 Generating A New Template File

The template file provided with the software is given in the listing in **Figure 1-10.** The style shown can be modified by changing spacing, newline, or comment selected features.

```
(* comment *)
(*.h-*)
(*.l0*)
(*.r78*)
(*.m-*)
(*.e+*)
(*.p+*)

DEFINITION MODULE defmodule1;

  FROM somemodule1 IMPORT
    importitem1, importitem2, importitem3;

  (* comment *)

  IMPORT
    importitem4;

  EXPORT QUALIFIED
    exportitem1;

  CONST
    const1 = -const2 + ( const3 - const4 ) * const5;
    (* comment *)
    const6 = NOT const7;

  TYPE
    subrange = [ low1 ..  low2 ];
    arraytype1 = ARRAY[ low1 ..  high1 ], [ low2 ..  high2 ] OF type1;
    arraytype2 = ARRAY subrange OF type2;
    opaquetype;
    enumtype = ( enum1, enum2, enum3 );
    settype = SET OF enumtype;
    recordtype = RECORD
                   rec1, rec2, rec3 : type2;
                   CASE variant : type3 OF
                     value1 :
                       caselabel1 : type4; |
                     value2 ..  value3 :
                       caselabel2 : type5;
                   END;
                 END;
    ptrtype = POINTER TO type6;
    proctype1 = PROCEDURE
                  (     type7,
                    VAR type8 ) : BOOLEAN;
    proctype2 = PROCEDURE
                  ( VAR type9,
                       type10 );
    proctype3 = PROCEDURE
                  ( type9,
                    type10 );
    proctype4 = PROCEDURE() : BOOLEAN;
```

Figure 1-10 Listing of TEMPL (cont'd next page)

```
  VAR
    var1 : type11;  (* comment *)
    var2 : type12;

  PROCEDURE procname1
            (     param1 : type13;
              VAR param2 : type14  (* comment in params *) ) : BOOLEAN;

  (* comment *)

  PROCEDURE procname2
            ( VAR parm3 : type15;
                  parm4 : type16 );

END defmodule1.
DEFINITION MODULE defmodule2;

  FROM somemodule2 IMPORT
    (* comment *) importitem5, importitem6,
    (* comment *) importitem7;

  EXPORT QUALIFIED
    (* comment *) exportitem4, exportitem5,
    (* comment *) exportitem6;


END defmodule2.

IMPLEMENTATION MODULE impmodule1;

  CONST
    const8 = const9;
    const10 = const11;

  TYPE
    type17 = type18;
    type19 = type20;

  VAR
    var3  [ 0FH:0FH ] : type21;
    var4,
    var5,
    var6 : type22;


  PROCEDURE outerprocname1
            ( parm5 : type17;
              parm6,
              parm7,
              parm8 : type18 );


    PROCEDURE innerprocname1() : BOOLEAN;
    (* comment *)

    BEGIN (* comment *)
      (* comment *)
      statement1;
    END innerprocname1;
```

Figure 1-10 Listing of TEMPL (cont'd next page)

```
BEGIN
  x1 := -x2 + x3 > ( x4 * x5 );    (* comment *)
  (* comment *)
  set := { setelement1, setelement2 };
  qualset := qualident { setelement1, setelement2 };
  pointer^[index].ident := record.somefield;
  procedurecall( x6, x7 );
  WHILE NOT condition DO
    statement2;
  END;
      WITH record DO
    statement3;
  END;
  LOOP
    statement4;
  END;
  REPEAT
    statement5;
  UNTIL condition;
  IF condition
  THEN
    statement6;
  ELSIF condition
  THEN
    statement7;
  ELSE
    statement8;
  END;
  FOR x8 := x9 TO x10 BY -x11 DO
    statement9;
  END;
  CASE x12 OF
    x13,
    x14 :
      statement10; |
    x15 ..  x16 :
      statement11;
  ELSE
    statement12;
  END;
  RETURN x17;
END outerprocname1; (* comment *)

(* comment *)


PROCEDURE procname3
          ( VAR parm3 : type23;
                parm4 : type24 );

VAR

TYPE

CONST

BEGIN
  statement1;
END procname3;


END impmodule1.
```

Figure 1-10 Listing of TEMPL (cont'd next page)

```
MODULE progmodule1 [ priority ];

  MODULE internalmodule;

  BEGIN
    statement13;
  END internalmodule;

BEGIN
  (* Comment *)
  statement14;
END progmodule1.
```

Figure 1-10 End Of TEMPL Listing

As an example of how the template file may be edited we extract the IF THEN ELSE construct as shown in the listing in **Figure 1-10** and reproduce it in the listing in **Figure 1-11.** We will edit this construct to illustrate methods for changing the formatting style.

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

Figure 1-11 Listing Of IF THEN ELSE Format Template

In **Figure 1-12** the style of the IF THEN ELSE construct has been changed so that THEN occurs on the same line as the condition and the statements are indented four spaces.

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

Figure 1-12 Listing For Revised IF THEN ELSE Format Template

The template compiler is able to detect style inconsistencies in the template file. **Figure 1-13** shows the format of a **REPEAT** construct followed by an **IF THEN ELSE** construct. The style inconsistency occurs because of the different indentation level for the **IF** and for the **REPEAT** constructs. The template compiler will therefore generate the error message shown in **Figure 1-14.**

The ***M2FORMAT Compiler*** error messages identify the line numbers in the template file, the offending tokens, and the two key formatting parameter values for each token. It is clear from **Figure 1-14** that the "spaces" parameter for the **IF** token is inconsistent with that for the **REPEAT** token.

```
REPEAT
  statement5;
UNTIL condition;
  IF condition
  THEN
    statement6;
  ELSIF condition
  THEN
    statement7;
  ELSE
    statement8;
  END;
```

Figure 1-13 IF THEN ELSE Format Template With An Error

```
>> Template conflict
>> Line 154 : Token REPEAT,                    1 line, 0 spaces
>> Line 158 : Token IF,                        1 line, 2 spaces
```

Figure 1-14 Error Message From Template Compiler

As an example of the flexibility that the template file offers within a construct, you have a wide range of choices for spaces and new lines. This is shown in an extreme example in **Figure 1-15.** The style in this example is valid and will compile.

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

Figure 1-15 Listing For Another Valid Formatting Style

1.7 Formatting Options

Formatting options are selected through comment commands. Comment-selected features in the ***Modula-2*** source file have precedence over default features in the template file. The form of the comment command is **(*.x*)** with no spaces. The command can be in either upper or lower case. All comment commands in the ***Modula-2*** source file are passed through to the formatted output.

1.7.1 End Comments

End comments enhance the readability of programs by placing a comment at the close of control constructs (e.g. **END (* while x↔*);**). This option is activated using (*.e+*) in either the template file or the source file. It is deactivated using (*.e-*). When activated, end comments are placed immediately after the keyword **END** on control constructs only if an end comment is not already present.

If the first identifier of the control construct (e.g., **IF first**) is not preceded by a left parenthesis, the identifier is included in the end comment, otherwise it is omitted.

Source File

```
IF first > second
THEN
  statement;
END;
```

Formatted File

```
IF first > second
THEN
  statement;
END(* if first *);
```

Figure 1-16 Simple Expression

Source File

```
IF ( first > second ) AND ( second < third ) THEN
  statement;
END;
```

Formatted File

```
IF ( first > second ) AND ( second < third ) THEN
  statement;
END(* if *);
```

Figure 1-17 Complex Expression

1.7.2 Line Overruns

Line overruns are handled automatically. Lines that exceed end of line are wrapped to the next line on a word boundary and are placed directly under the first word on the preceding line. There are however, cases where line overruns are handled specially. Here are several examples. In complex expressions, the innermost expression (i.e. lowest nested parenthesis) is never broken across line boundaries. This is shown in **Example 1, Figure 1-18.** Parameter lists of a procedure call are also handled specially as in **Example 2, Figure 1-19.**

Line overruns may be marked with the comment (*@*) by invoking the comment command (*.m+*) in either the template file or the source file. The comment command (*.m-*) disables line overrun marking.

Source File

```
IF ( first > second ) OR ( third  > forth ) OR ( fifth > sixth + x )
                                                    ^Right margin
```

Formatted File (with m+ option)

```
IF ( first > second ) OR ( third  > forth ) OR
(*@*)    ( fifth > sixth + x )
```

Formatted File (with m- option)

```
IF ( first > second ) OR ( third  > forth ) OR
   ( fifth > sixth + x )
```

Figure 1-18 Example 1, Expression Overrun

Source File

```
        procedurecall( parameter1, parameter2, parameter3, parameter4 );
                                                                ^ Right margin
```

Formatted File

```
        procedurecall( parameter1, parameter2, parameter3, parameter4 );
```

Figure 1-19 Example 2 Procedure Call Parameter Overrun

1.7.3 Padding

(*.p+*) in either the template or source file turns on padding of identifier lists. The command (*.p-*) turns off padding. Padding is recommended only for formatting styles where identifier lists consist of a single line for each identifier as shown in the example. When padding is on, identifiers line up on the " : " or " = " signs as shown in **Figure 1.20.**

Padding off:

```
TYPE
  typel = RECORD
            first : INTEGER;
            second : REAL;
          END;

  thesecondtype = REAL;

VAR
    first : INTEGER;
    secondandthird : INTEGER;
```

Padding on:

```
TYPE
  typel          = RECORD
                     first  : INTEGER;
                     second : REAL;
                   END;

  thesecondtype = REAL;

VAR
    one           : INTEGER;
    thisisabigone : INTEGER;
```

Figure 1-20

1.7.4 Hardcopy Switch

LOGITECH M2FORMAT can be configured to produce output files that can be copied to a printer for hardcopy. Hardcopy is selected by inserting the hardcopy switch **(*.h+*)** in the template file. When hardcopy is enabled, the file PRINTER.M2F must be present in the directory set by the environment variable "**M2F**". PRINTER.M2F contains configuration information used for hardcopy. The hardcopy flag may only be used in the template file and is not permitted in the source file to be formatted. For additional information on hardcopy see **Section 1.8 Hardcopy Features.**

1.7.5 Formatting Switch

Formatting may be selectively turned on and off within a source file by using **(*.f+*)** or **(*.f-*)**. This lets you selectively retain existing format within a file, such as a table in comment blocks. *M2FORMAT* requires correct syntax in your input program even when formatting is disabled.

1.7.6 Syntax Checking Switch

M2FORMAT syntax checking may be selectively turned on and off within a source file by using **(*.s+*)** or **(*.s-*)**. This lets you use *M2FORMAT* on programs with non-standard *Modula-2*. Any section of code where syntax checking is disabled must result in a syntactically correct program if that section were to be removed from the program as shown in the example. The section of code is passed through to the formatted output file as is from your input file as in **Figure 1-21.**

```
VAR
  MyInt : INTEGER;
  (*.s-*)
  Buff :  Byte ABSOLUTE Infile;
  (*.s+*)
  Fileptr : INTEGER;
```

Figure 1-21 Modula-2 Code With Non Standard Syntax

1.7.7 Margins

Left and right margins may be selected using **(*.lx)** and **(.rx)** in either the template file or your input file. The valid range for the left margin is **0-15** and the right margin is **60-240**. An attempt to modify the margins beyond the valid range will result in an error.

1.8 Hardcopy Features

1.8.1 Overview of Hardcopy Features

LOGITECH M2FORMAT can produce printable output by setting the hardcopy switch (`*.h+*`) in the template file. When hardcopy is enabled, the file PRINTER.M2F must be present in the directory set by the environment variable "**M2F**". PRINTER.M2F contains configuration information used for hardcopy.

You can set the number of physical lines per page, to accommodate different printers. An optional page heading containing the file name, page number, date and time of formatting is placed on the third line of the page. The listing starts on line six and continues to a bottom margin of two lines.

To minimize the effects of page breaks, set the number of lines before the bottom margin, this should cause the next major construct to start on the following page. Major constructs are defined by the keywords **BEGIN, WHILE, IF, ELSE, ELSIF, CASE, REPEAT, LOOP, WITH, FOR, PROCEDURE, VAR, TYPE,** and **CONST.**

Most printers have features such as bold or italic text that are selectable using special escape sequences. A set of six escape code sequences (which you can redefine) are used to select modes for printing identifiers, keywords, procedure and module headings, and comments. The listing in **Figure 1-22** contains an example of a program module that has comments, module heading, and **bold procedure headings.**

```
File: example.mod    Page 1    11/21  9:35

PROGRAM Example;

(* program to show the use of printer attributes *)

FROM InOut IMPORT
  (* proc *) WriteString, Writeln;


PROCEDURE WriteMessage
            (s : ARRAY OF CHAR);

BEGIN
  WriteString (s);
  WriteLn;
END WriteMessage;

BEGIN (* main program *)
  WriteMessage ("hello world");
END Example 1.
```

Figure 1-22

1.8.2 Configuring PRINTER.M2F

PRINTER.M2F contains user-defined information for configuring hardcopy. The file layout is shown below in **Figure 1-23.**

```
number of lines per physical page
bottom number  of lines for major construct to start on
header enable (1=header on, 0=header off)
escape sequence to be sent at beginning of output file
escape sequence to be sent at end of output file
escape sequence to be sent at beginning of the module heading
escape sequence to be sent at end of module heading
escape sequence to be sent at beginning of comment text
escape sequence to be sent at end of comment text
escape sequence to be sent at beginning of keyword
escape sequence to be sent at end of keyword
escape sequence to be sent at beginning of identifier
escape sequence to be sent at end of identifier
escape sequence to be sent at beginning of procedure heading
escape sequence to be sent at end of procedure heading
```

Figure 1-23

The first line in the file is the number of lines per physical page in the range of 50 to 120, which is set to 66 for most printers.

The second line of the file is the number of lines from the bottom of the page to start major constructs on the next page. (**BEGIN, WHILE, IF, ELSE, ELSIF, CASE, REPEAT, LOOP, WITH, FOR, PROCEDURE, VAR, TYPE,** and **CONST**)

The third line selects whether or not a page header is placed at the start of each page.

The forth through fifteenth lines contain escape code sequences to be sent before and after certain elements of ***Modula-2***.

Escape code sequences are one per line, and must be printable ASCII characters. Non-printable ASCII may be encoded as three decimal digits, after a backslash character, representing the decimal character value of the desired character. For example,

```
\027@x
```

sends an escape character followed by an @ and an **x**. The character value must be between **0** and **255**, or an error will be reported.

Escape code sequences sent at the beginning and end of the formatted file are useful for sending formfeeds (`\012`) at the beginning or end of listings, or any global printer configuration commands. If no escape code sequence is desired for any of the constructs, that line must contain a single backslash. Other escape code sequences are useful for highlighting procedure headings, comments, keywords or identifiers. Normally, you enable a certain printer option at the beginning of certain element of ***Modula-2***, then disable the printer feature after the feature.

Depending on the capabilities of the printer, care must be taken when selecting printer control as there are times when escape sequences overlap. For example, assume that boldface type is selected for procedure headings and italics for identifiers. When hardcopy is enabled, the segment of the ***LOGITECH M2FORMAT*** output file is shown in **Figure 1.24.** Escape sequences are shown in brackets **< >**. If the escape sequence selected for italics is not independent of the selection of bold (e.g. the printer can't print bold italics), the rest of the procedure heading may not be bold.

```
<bold on>PROCEDURE  <italic on> procname <italic off> ( );<bold off>
```

Figure 1-24 Segment Of M2FORMAT Output File With Hardcopy Enabled

Comments can be added to the file to increase comprehension of the file. Comments are indicated by a pound sign (#) in the first column of a line. The comment extends to the end of the line. An example of a printer configuration for a ***Hewlett-Packard Laser Jet*** is shown in **Figure 1-25.**

```
########################################################
# printer config file for laserjet for bold comments,
# procedure and module headings
########################################################
# physical page length
60
# number of line that cause major constructs to start on the next page
3
# header on/off (on = 1, off = 0)
1
#escape sequence sent at the start of a file
\
#escape sequence sent at the end of a file
\012
#escape sequence sent at the start of a module heading
\027(s5B
#escape sequence sent at the end of a module heading
\027(s0B
#escape sequence sent at the start of a comment
\027(s5B
#escape sequence sent at the end of a comment
\027(s0B
#escape sequence sent at the start of a keyword
\
#escape sequence sent at the end of a keyword
\
#escape sequence sent at the start of an identifier
\
#escape sequence sent at the end of an identifier
\
#escape sequence sent at the start of a procedure heading
\027(s5B
#escape sequence sent at the end of a procedure heading
\027(s0B
```

Figure 1-25 Printer configuration for a Hewlett Packard LaserJet printer

1.9 Syntax Extensions

LOGITECH M2FORMAT currently recognizes and formats standard *Modula-2* source code files as defined by the *Modula-2* book by Niklaus Wirth *Programing in Modula-2* third edition. In addition *M2FORMAT* recognizes the *LOGITECH Modula-2* language extensions of the compiler, for example.

```
VAR
  PortAddress  [0FH:0FH] : CARDINAL;
```

Figure 1-26 Example of LOGITECH Modula-2 language extension

1.10 Formatting Resolution

In general, the template lets you specify the formatting style only once for each construct in the ***Modula-2*** language. However, to increase formatting flexibility, there are certain constructs in the template that are repeated. The various constructs that allow additional resolution are now discussed.

1.10.1 Identifier Lists

The identifier list construct is used extensively in ***Modula-2*** for variable declarations, enumerated type declarations, record field declarations, procedure declaration parameters, procedure call parameters, and case statement labels. The template lets you specify the formatting style of each of these identifiers separately.

1.10.2 Procedure Declarations

Procedure declarations allow for extensive flexibility based on the number and type of parameters. Procedure declarations can be of three types:

1) a procedure declaration with parameters that are all passed by value (no **VAR**'s).

2) a procedure declaration with at least one parameter passed by reference.

3) a procedure declaration with no passed parameters. Examples of each type are shown in the listing in **Figure 1-27**.

```
PROCEDURE PassByValue ( x : INTEGER; y : REAL );
PROCEDURE PassByReference ( VAR x : INTEGER; y : REAL );
PROCEDURE NoParameters ( );
```

Figure 1-27 Three Types of Procedure Declarations

The increased formatting resolution lets you specify details such as the parameters aligned regardless of whether reference or value is used as seen in **PROCEDURE PassByReferenece.**

1.10.3 Nested Procedures

Procedures and functions are classified as being either first level (visible from the main body of a program) or as nested procedures. ***LOGITECH M2FORMAT*** lets you specify the indentation level and number of blank lines of nested procedures relative to the parent procedure.

1.10.4 Import Lists

In recognizing an emerging style often used in import lists, ***M2FORMAT*** allows different formatting for import lists that used comments to specify what type of item is being imported. This style is encapsulized in the definition module **defmodule2** in the template source file.

1.11 Error Messages

Error messages that occur during the execution of the template compiler are indicative of inconsistencies in formatting style. Check the indicated style features for the inconsistency and correct them.

Errors that occur during the execution of ***LOGITECH M2FORMAT*** are indicative of an incorrect input source file. Error comments will assist in identifying syntax errors in the source file. Check to see that the source file is a compilable ***Modula-2*** source file and correct the syntax errors. The line number indicated represents the approximate location of the syntax error. Syntax errors cause ***M2FORMAT*** to terminate immediately.

If comment commands have been incorrectly entered into the source file, an error will occur (e.g., `(*.p*)` is incorrect). Incorrect comment commands generate an error message but do not terminate ***M2FORMAT***.

If PRINTER.M2F is not present or it contains invalid configuration data, an error message will be generated and ***M2FORMAT*** will be terminated.

Notes:

Chapter 2
The LOGITECH Linker

The ***LOGITECH Linker*** combines separately compiled modules into a single executable file. It takes the object (.OBJ) files of the module to be linked as input, and produces an executable (.EXE) or an overlay (.OVL) file, and a map (.MAP) file. The .MAP file provides information about the associated .EXE file. This .MAP file must be present to use the the ***RTD*** and ***PMD*** debuggers.

The first .OBJ file name the ***Linker*** accepts defines the name of the .EXE output file. The names of the modules to be imported are encoded in the .OBJ file generated by the ***LOGITECH Modula-2 Compiler***. This means that after you give the name of the first .OBJ file to the ***Linker***, the ***Linker*** will automatically search for other .OBJ and .LIB files to be imported.

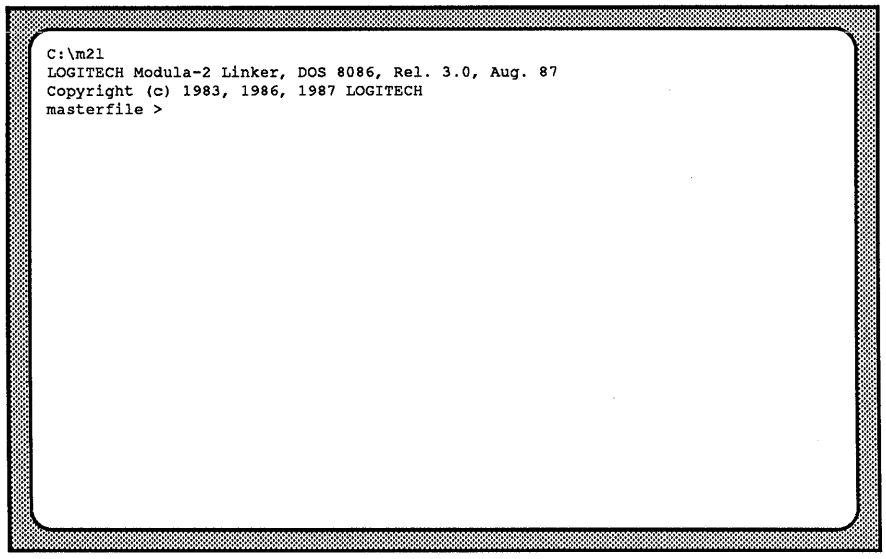
Parameters may be given on the command line for various enhancements. **/OPT**, for example, lets the ***Linker*** combine only the necessary procedures when creating an output file. Most standard linkers put more procedures in the output file than is necessary. However the ***LOGITECH Linker*** is able to combine the minimum necessary procedures for its output file. This creates a smaller, more efficient executable file.

2.1 How to use the Linker

Step 1: To load the ***LOGITECH Linker*** type:

M2L ↵

You will see a screen resembling this:



Step 2: Enter the filename and any options.

(See section 2.5 Linker Options).

Compiled modules which are ready to be linked use .OBJ as the default extension. The default drive is the current disk drive.

If you use the default options, the ***Linker*** automatically links all other necessary modules. It also lists all imported modules and their corresponding file names.

You can specify the file name on the ***DOS*** command line directly after the command to load the ***Linker***. The ***Linker*** will then display the banner and begin linking.

2.2 Search Strategy

2.2.1 Object files

By default, object files use the extension .OBJ. To know which additional files are to be used during linking, the ***LOGITECH Linker*** finds references that are imbedded in the .OBJ file which is being linked. Files which are needed for linking have an extension of .OBJ or .LIB. Object files compiled with the ***LOGITECH Modula-2 Compiler*** contain information about the imported modules that need to be imported into the final .EXE or .OVL file.

The object files are searched for in the following order:

1: In the current directory.

2: In the directory where the master (main) file came from. This path is called the master path.

3: In the directories specified by the environment variable **M2OBJ**.

Each time a directory path is specified on the command line, or when the ***Linker*** asks for a file it does not find, the directory path is added to the begginning of the list of directory paths specified by the environment variable **M2OBJ**.

If the ***Linker***, after the search, does not find the file it assumes that the file is defined in a library file. (See **Section 2.2.2 Library Files**, below).

When the ***Linker*** asks for an object file, the request is repeated until an appropriate file is found or [Esc] is pressed. [Esc] means that the file is not available.

2.2.2 Library files

A library file is a collection of different object files. Library files use the extension .LIB. They can be created using the ***Microsoft Librarian*** utility (they also follow the ***Microsoft*** format for library files).

The *LOGITECH Linker* recognizes four predefined names for library files:

M2LIB.LIB	(contains the standard ***Modula-2*** library)
M2RTS.LIB	(contains the run-time system)
M2REAL.LIB	(contains the reals library)
M2USER.LIB	(contains a library which you can define)

As some object files have not been found, the ***Linker*** will scan the library files it knows, and extract the missing .OBJ file from them.

The known libraries are the four libraries defined above by default. You can use your own libraries in one of the following ways:

- Copy your private library to M2USER.LIB (which is defined for this purpose).
- Give the name(s) of your private libraries on the command line of the ***Linker***, preceded by the switch **/LIB** (or **/L**) as in the following example :

  m2l obj1 obj2 /LIB mylib1 mylib2 mylib3

- If the ***Linker*** does not find an object file in a library, it prompts you for it. You can either give a file name with no extension and the ***Linker*** will assume this new name is the name of an object file, or you can give a file name with the extension .LIB. The ***Linker*** will assume the given name is the name of a library file. The file you specify will be added to the list of the other library names, and the missing object will be searched for.

The library files are searched for as follows:

1: The current directory is searched.

2: In the directory where the master (main) file came from. This path is called the master path.

3: The directory path specified by the environment variable **M2LIB** are searched.

2.2.3 Output files.

After successful linkage, the ***Linker*** writes all output files (.OVL, .EXE and .MAP formats) in the current directory.

2.3 Temporary Files

If the ***LOGITECH Linker*** runs out of memory during the linking process, it creates a temporary file on disk on which it stores data. This frees the memory for the linking process.

These temporary files are automatically erased when the linking process is complete.

2.4 MS-DOS Environment

The ***Linker*** uses four variables as environment parameters.

To create or modify these variables using the ***DOS*** command, type:

SET <variable> = <value>

Variables	Specify path for
M2OBJ	Object (.OBJ) files
M2LIB	Library (.LIB) files.
M2MAP	Map (.MAP) files.
M2TMP	Temporary files (can be used with RAM disk).

2.5 Linker Options

The ***LOGITECH Linker*** accepts several options, from any position on the command line.

Three kinds of option are accepted on the command line :

/ Those that begin with a forward slash " / ". (Switch options).

@ Those that begin with " @ ". Tells the ***Linker*** to use the following filename as the name of a command file that it must read. (Must be followed by filename).

\+ And " + ". Tells the ***Linker*** to continue the input of the command line on a new line. This is useful for a long command line. (Stand alone character).

The following options can be used :

(note : the case of the characters has no effect).

/BAT /B Tells the ***Linker*** it is running in a batch file. Ordinarily the ***Linker*** tells you to enter a character when it prompts the list of undefined symbols or the list of unreferenced procedures.

/CASE (/C) Ignore the case of the characters in symbols.

By default, the case of the character is checked when comparing symbols. If this switch is present, the ***Linker*** will not check the case, i.e. " a " is equivalent to " A ".

/LIB (/L) The following names are library file names.

On the command line, all filenames are assumed to be the names of .OBJ files. If this switch is present, all the filenames that follow are assumed to be the names of .LIB files.

/LINE (/LI)

List line numbers in the .MAP file.

In order to produce symbolic information for debuggers like ***SYMDEB***, ***CODEVIEW*** (Copyright © **Microsoft**) or ***PFIXPLUS***, (Copyright © **Phoenix Software Associates Ltd.**) this switch tells the ***Linker*** to put source line numbers on the map file.

/NOM

The ***Linker*** generates a .MAP file by default. This switch tells the ***Linker*** not to generate such a file.

/MAP (/M)

Produce a .MAP file (default).

Forces the ***Linker*** to produce a map file. The full syntax is:

/MAP [= <filename>]

If the " = "sign is present, the filename that follows is assumed to be the name of the map file to be created.

If the equal sign is not present, the master name is used as filename.

/MS

Produces a CMD file.

The ***LOGITECH Linker*** will create a file CMD, on which it writes the names of the object file. This file can then be used with the standard **Microsoft *Linker***.

If the ***Linker*** generates such a command file it will not generate (MAP, OVL or EXE) file(s).

/NDL

No Default Library search.

The ***Linker*** uses information encoded in the object file to know the name of eventually used library. This switch tells the ***Linker*** not to use this information.

/OPT (/O)

Deletes unreferenced part of code.

Tells the ***Linker*** to check for unreferenced part of code.

The ***Linker*** will then prompt the name of the deleted procedures. These procedures will then not be present in the output file.

/OPTQ (/OQ) Deletes unreferenced part of code with query.

Same switch as **/OPT**. The difference is that the ***Linker*** prompts the name of each unreferenced procedure and asks the user if he will need this procedure in the output file.

/OUT Specify the name of output (.EXE or .OVL) file

Lets you specify the name of the output (EXE or OVL) file. The syntax is:

/OUT = <filename>

/PACK (/P) Produce a packed .EXE file.

This switch forces the ***Linker*** to pack the .EXE output file. .OVL files are automatically packed. This effects the size of the resulting output file only, not the memory the output file uses when it is executed.

/STACK (/S) Specify the stack size.

Lets you change the size of the stack defined for the output file (works only for EXE file, since OVL files have no stack). The default size for a Modula-2 program is 8000 bytes. The syntax is:

/STACK = value

Value must be in the range of [512 ... 65535].

/WAIT (/W) Wait before creating a file.

The ***Linker*** asks you to press a key before creating an output file. This lets you change the disk in the disk drive.

NOTE

If the ***Linker*** has created a dump file on a floppy disk, do not remove this disk until the end of the link process!

2.6 How to Link an Overlay

During its execution, a program can load overlays.

LOGITECH Modula-2 overlay files, with an extension of .OVL, can be generated only by the *LOGITECH Linker* (and not by a *DOS* linker).

An overlay can reference modules which are part of other overlays. To know the names of the modules and objects which are declared in other referenced overlays, the .MAP files of these overlays must be given to the *Linker*.

The syntax of the command line is the following :

```
M2L <objectfilenames> (<mapfilenames>)
```

Any switch can be used on the command line.

The *Linker* then creates a .OVL file, linking all the object files specified in the command line. It will then link the imported object files that are not declared in the .MAP files.

The .MAP file generated when linking an overlay can be used again to create another overlay. When creating the .MAP, the *Linker* writes the name of the other imported map, so you have only to specify the name of the first imported map, and the *Linker* will automatically find the name of the others.

Example :

In this example the program uses the following strategy:

The first part of a program is the "**Base**" which then loads the "OVERLAY1" overlay. The **Base** part of any program is the **main** .EXE file of the program. For this example, the **Base** of the main .EXE file will be named "BASE".

While overlay "OVERLAY1" is executing, it will load another overlay "OVERLAY2".

When you run BASE.EXE it executes the first overlay (.OVL) file, which we call here OVERLAY1. When OVERLAY1 is running, it can load OVERLAY2.

To link all the components together, follow this procedure:

Step 1: Link the BASE module:

M2L BASE [↵]

A .MAP file is generated by default.

Step 2: Link the first overlay file:

M2L OVERLAY1 (BASE) [↵]

With the BASE, a .MAP file is generated by default:

Step 3: Link the second overlay file(s):

M2L OVERLAY2 (OVERLAY1) [↵]

This overlay might reference OVERLAY1 and BASE. However, since OVERLAY1 already references BASE, one has only to give the name of the first referenced .MAP, which is OVERLAY1.MAP:

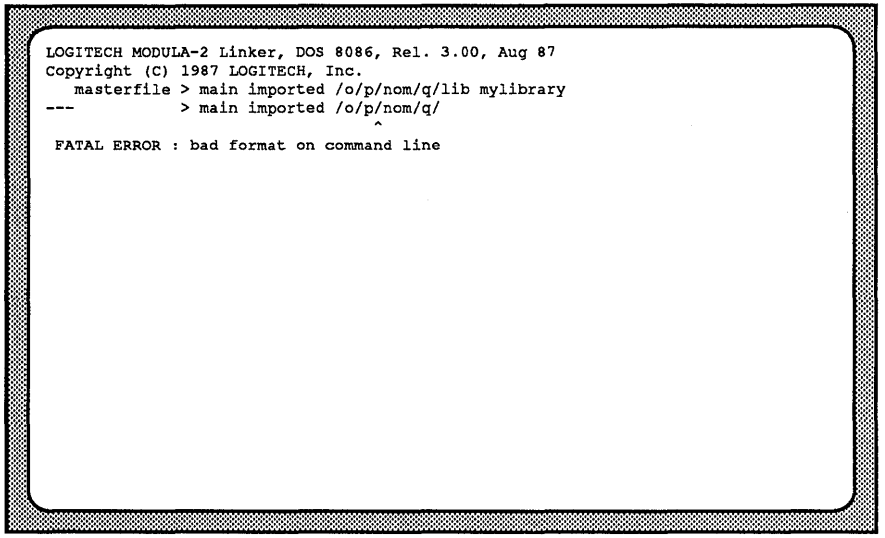
Switches can be set anytime. For example, try to optimize (**/OPT**) OVERLAY2. You can also try to optimize OVERLAY1 or BASE, but be careful, the ***Linker*** might delete procedures which are referenced by an overlay! To avoid this, use the switch **/OPTQ**, and specify the ***Linker*** procedures which it can not delete.

2.7 Linker Error Messages

2.7.1 Common Errors

bad format on command line

Misspelled switches or unrecognized characters have been given on the command line. The ***LOGITECH Linker*** rewrites the typed command line showing where it has detected an error:



version conflict in module

A version conflict has been detected in one of the ***Modula-2*** object files. This error arises typically when a **DEFINITION** module has been recompiled and the **IMPLEMENTATION** part has not.

no object file linked

The ***Linker*** could not link any files.

more than one entry point in the program

The object files linked together define more than one entry point. Since the entry point of a program is the point where *DOS* should start its execution, only one entry point must be defined.

master file of overlay is not a main

Each overlay is defined by its main module. The main module is a standard *Modula-2* main module without **DEFINITION** part. This error occurs when the first object file given on the command line is not a main *Modula-2* module.

no entry point in program

There must be one entry point for any program.

undefined symbol

The *Linker* could not resolve all the external references. This occurs when some object files are missing, or when there is a version conflict between modules.

invalid library

One of the files given to the *Linker* as a library file is not a *Microsoft* library file.

source map file not found

One of the .MAP files given to the *Linker* has not been found.

incorrect map file

The .MAP file given to the *Linker* is incorrectly formatted.

Name of main program not found in map

(.MAP file maybe corrupted).

When using a .MAP file to find the names of the modules which are part of another layer, the *Linker* could not find the main modules of the layer corresponding to this .MAP.

can not open output map file (disk full ?)

can not open output file (disk full ?)

Open failure. Occurs if the disk is full or write-protected.

too many nested overlays

An overlay can be loaded by another overlay, this last overlay can also be loaded by another overlay, and so forth. The maximum number of nested overlays is **255**.

overlay without entry point.

Occurs if the first object file given on the command line (which will define the overlay) is not a main *Modula-2* module.

bad key in map

The .MAP is corrupted or the object files described by it have not been generated by the *LOGITECH Compiler*.

insufficient memory to link the program

The program being linked is too big to be linked. Try to recompile some files with the *LOGITECH Compiler* with the option **/NOSY**. This will generate less symbols in the object file.

disk full

Occurs if the disk is full or write-protected.

more than one stack segment

A program may have only one stack segment.

symbol defined twice

Each symbol defined by the object files can occur only once.

can not open temporary file

As it is running out of memory, the *Linker* copies part of its data to disk. This error occurs when it is unable to open the open files, either because the disk is full, write-protected, or the path specified in the **M2TMP** environment variable is not good.

2.7.2 Special Errors

Special errors occur when linking object files generated by other language compilers or by assemblers. They should not occur if the object files being linked have been created by the ***LOGITECH Compiler*** or are part of the ***LOGITECH Modula-2 Library***.

NOTE

The ***LOGITECH Linker*** will usually link object files generated by the ***LOGITECH Compiler*** with object files generated by assembly language if there is no use of **GROUP** statement in the assembly language source file.

`bad record in object file`

The ***Linker*** can not recognize one or more records in the object file being linked. The object file is either corrupted or does not follow the Intel Standard Format for object files.

`unknown record in object file`

The ***Linker*** has detected a record in the object file being linked that is not implemented, (i.e. the ***Linker*** does not interpret the records).

`not implemented kind of fixup.`

Occurs if an object file was not generated by the ***Modula-2 Compiler***.

`segment size greater than declared`

`unable to pack output file`

`overflow in address computing`

`underflow in address computing`

`offset in removed block`

All these errors come from corrupted .OBJ files, or .OBJ files that have not been generated by the ***Modula-2 Compiler***.

2.8 Overlays

Overlays are basically equivalent to standard ***Modula-2*** programs. They are built from a main ***Modula-2*** module which imports other ***Modula-2*** modules.

The difference is that the modules that are imported by an overlay are not necessarily part of the overlay itself, but can be part of another overlay. An overlay can make reference to other overlays, using the exported identifier of the modules which are part of other overlays.

Overlays are loaded and unloaded during the execution of the **Base** and overlay programs which use them,making it possible for overlays to share memory. They can also be nested, in the sense that one overlay can load another overlay.

The first program of an application (with the extension .EXE) is called the **Base**. Since the **Base** also behaves like an overlay, (it can be seen as an overlay of the ***DOS*** interpreter which invokes it), features applying to overlays also apply to the **Base**.

2.8.1 Creating an Overlay

A ***LOGITECH Modula-2*** overlay file has a .OVL extension. It can be created only by the ***LOGITECH Linker***. Since an overlay can reference other overlays (for example the overlay which will load this overlay (may be the **Base**)), the ***Linker*** needs to know which identifiers are exported from these referenced overlays. This information is given to the ***Linker*** by the .MAP file generated as the ***Linker*** created these other overlays (or the **Base**).

The syntax on the Linker's command line is the following :

```
m2l <object_file> ( <map_file> ) /anySwitch
```

The ***Linker*** first links together the modules specified in the list of the .OBJ files on the command line. Then, it tries to link all the other imported files, checking each time if those files are not already defined in the .MAP file (i.e. if those modules are not already part of other overlays). If those modules are already defined in some other overlay, the ***Linker*** will put some special information in the output file in order to permit the new overlay to access those modules. This means that their code will not be written in the new overlay.

2.8.2 The Overlay Manager

The **Overlay** manager is defined in the **M2LIB** module library **Overlay**. This module provides procedures to call, load, or unload overlays. It also provides a way for you to control memory allocation.

2.8.3 Loading an Overlay

Two procedures in the **Overlay** manager loads the overlays. The routine you use will specify the nature of the overlay being loaded:

PROCEDURE CallOverlay (fileName, errorCode, status);

loads the overlay specified in **<fileName>**, starts its execution, and then unloads it automatically. Overlays which are called through this routine are called "**SUBPROGRAM**". After the overlay has been unloaded, control is returned to the program which issued the call.

PROCEDURE InstallOverlay (fileName, errorCode, status):OverlayId;

loads the specified overlay, starts its execution and then give back the control to the calling program. Overlays loaded through this routine are called "**RESIDENT OVERLAYS**". Their main feature is that they are not automatically unloaded from memory, but instead the code and data of the overlay becomes logically part of the program in which the call to **InstallOverlay** was issued. After the termination of a resident overlay, the code and data can still be used by the application.

PROCEDURE DeInstallOverlay (fileName, errorCode, status):OverlayId;

unloads explicitly a resident overlay.

2.8.4 Execution of the Overlay

Once it has been loaded, the code of the main module of the overlay is executed, as in a standard ***Modula-2*** program. The execution is terminated as the last statement of the main module is executed, or in the case of a run-time error.

2.8.5 Termination of the Overlay

2.8.5.1 Termination of a Subprogram

When a subprogram is terminated, it is unloaded from memory and control is returned to the calling program.

Two pieces of information about the overlay are given back to the calling program:

1: The loading status of the overlay:

The loading status gives information about the loading of the overlay. If the loading was successful (the overlay could be found on the disk, there was enough free memory to load it...), the loading status has the value **Done**.

2: Execution status.

If the value of the loading status is **Done**, the execution status gives information about the way the overlay was terminated. Normally this status has a value of **Normal** (if the execution was successful). In the case of unsuccessful termination, the possible values of the status is defined in the **RTSMain** (i.e. range error, divide by 0, stack overflow, ...)

2.8.5.2 Termination of a Resident Overlay

In the case of loading errors and execution errors, resident overlays behaves like subprograms, i.e. they are unloaded from memory and control is given back to the calling program, returning the same information about the loading status and the execution status.

If the resident overlays can be successfully executed, they become logically part of the calling program and control is given back to the calling program.

A resident overlay can explicitly be unloaded from memory using the procedure **DeinstallOverlay (OverlayId)** provided by the overlay manager.

If a resident overlay is not unloaded explicitly by the application, it will be unloaded when the overlay to which it has become logically part is unloaded.

2.9 Accessing Overlays from within a Loaded Overlay

2.9.1 Subprogram

Subprograms can not be accessed from previously loaded overlays.

Subprograms behave like procedures. Subprograms can still access the subprogram which have been previously loaded (like a procedure can access the data of the procedure in which it is nested), or they can themselves call other subprograms. Once a subprogram is terminated, the code and data are unloaded from memory, an cannot be used any longer.

2.9.2 Resident Overlays

Resident overlays provide a way to load code for procedures during the execution of an application, and then explicitly unload them when they are no longer needed.

Typically, a resident overlay will import procedure variables from modules defined in previously loaded overlays. As the resident overlay is initialized, it changes the value of these procedure variables to some procedure it itself defines. When the resident overlay is initialized, its code and data can still be accessed from the rest of the application, using these procedure variables.

2.9.3 Termination Procedures.

The module **RTSMain** provides a way to install termination procedures using the procedure **InstallTermProc (PROC)**.

A termination procedure installed using **InstallTermProc** will be executed upon the unloading of the overlay in which the call to **InstallTermProc** is issued. Termination routines installed in the **<base>**.EXE are executed upon termination of the application.

A typical use of termination procedure is to release resources owned by the overlay. As an example, if a module which handles file operations (as **LogiFile** or **FileSystem**) is unloaded from memory (since the overlay in which it is defined is unloaded), one should close all the files opened using this module which are still open to ensure correct behavior of the application.

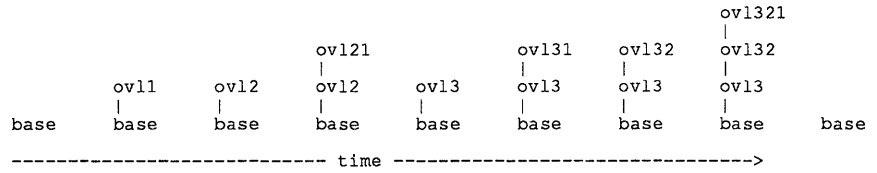
2.9.4 Initialization Procedures.

The module **RTSMain** provides a way to install Initialization procedures using the procedure **InstallInitProc (PROC)**.

Initialization procedures are procedures which will be called upon the loading of an overlay. Each time an overlay is loaded, all the initialization procedures installed will be executed.

2.9.5 An Example

Consider an application that consists of a base that loads several overlays as subprograms, one after the other as well as on top of each other:



The base contains the common module **FileHandler.**

Shared Resource

The module **FileHandler** installs a termination procedure that closes all open files upon termination of the application. This means that files that have been opened for example in OVL1 but that have not been close there explicitly, will be left open until some other overlay closes them, or until the application terminates.

Thus, files or any other resources generated by other modules, can be shared over overlay boundaries.

Up to this point, the implementation part of the module FileHandler could be the following:

```
IMPLEMENTATION MODULE FileHandler;

FROM RTSMain IMPORT InstallTermProc;

CONST MaxFile = 255;

TYPE  FileDesc    = RECORD
                      used   : BOOLEAN; (* TRUE if this entry is used   *)
                      handle : INTEGER; (* for instance DOS file handle *)
                    END;

      FileTable   = ARRAY [0..MaxFile-1] OF FileDesc;

VAR fileTable : FileTable;

PROCEDURE OpenFile (VAR f : File);

  (* opens a file and records it in a table  *)

  VAR i : CARDINAL;
BEGIN
  (* search for a free entry in the table *)
  i := 0;
  WHILE (i < MaxFile) & fileTable [i].used DO
    INC (i);
  END;

  IF i < MaxFile THEN     (* a free entry exist *)
    fileTable [i].used := TRUE;
    DOSCALL (OPEN,.....); (* now the file has been opened *)
  END;
END OpenFile;

PROCEDURE CloseAllFiles;
  VAR i : CARDINAL;
BEGIN
  (* called upon the termination of the application *)
  (* close all open files *)
  FOR i := 0 TO MaxFile-1 DO
    IF fileTable [i].used THEN
      DOSCALL (CLOSE,....);
    END;
  END;
END CloseAllFiles;

VAR i : CARDINAL;

BEGIN  (* FileHandle *)
  FOR i := 0 TO MaxFile-1 DO
    fileTable [i].used := FALSE;
  END;

  (* now install the termination procedure (CloseAllFiles) *)

  InstallTermProc ( CloseAllFiles )

END FileHandler.
```

Private Resources

Any module that does not want its resources to be shared over overlay boundaries, has to make sure that a termination routine is called for every overlay. This can be achieved by using the initialization routines.

In our example, the module **FileHandler** will have to install an initialization routine in its own initialization code (i.e. the module body). This initialization routine will be called for every overlay that will be loaded, and its execution is considered to be part of the currently loaded (i.e. the new) overlay. A termination routine to close the open files of this overlay can be installed now, inside the initialization routine.

Usually, it will be the same termination routine that is called on all levels. This means that the installing modules must have other means to identify, for example the files that have been opened in a specific overlay. This information can be obtained from the variable **RTSMain.currprocess^^.termOverlay** which is of type **OverlayPtr**.

According to this, the new implementation part of the module FileHandler would be the following:

```
IMPLEMENTATION MODULE FileHandler;

FROM RTSMain IMPORT InstallTermProc, InstallInitProc, currprocess,
                    ProcessDescriptor, OverlayPtr;

CONST MaxFile = 255;

TYPE  FileDesc    = RECORD
                      used   : BOOLEAN; (* TRUE if this entry is used    *)
                      handle : INTEGER; (* for instance DOS file handle *)
                      owner  : OverlayPtr; (* the specifi overlay were  *)
                                           (* the files has been opened *)
                    END;

      FileTable   = ARRAY [0..MaxFile-1] OF FileDesc;

VAR fileTable : FileTable;

PROCEDURE OpenFile (VAR f : File);

  (* opens a file and records it in a table  *)

  VAR i : CARDINAL;
BEGIN
  (* search for a free entry in the table *)
  i := 0;
  WHILE (i < MaxFile) & fileTable [i].used DO
    INC (i);
  END;

  IF i < MaxFile THEN      (* a free entry exist *)
    fileTable [i].used := TRUE;
    DOSCALL (OPEN,.....); (* now the file has been opened *)
    fileTable [i].owner := currprocess^^.termoverlay;
  END;
END OpenFile;
```

(Cont'd on next page)

```
PROCEDURE CloseAllFiles;
  VAR i : CARDINAL;
BEGIN
  (* called upon the termination of any overlay *)
  (* close all open files that were opened in the terminating overlay *)
  FOR i := 0 TO MaxFile-1 DO
    IF fileTable [i].used &
       (fileTable [i].owner = currprocess^^.termoverlay) THEN
      DOSCALL (CLOSE,....);
    END;
  END;
END CloseAllFiles;

PROCEDURE InitProcedure;
  (* called before the initialization of any new loaded overlay *)
  (* install the termination procedure 'CloseAllFiles' in the   *)
  (* new loaded overlay                                         *)
BEGIN
  InstallTermProc ( CloseAllFiles );
END InitProcedure;

VAR i : CARDINAL;

BEGIN  (* FileHandle *)
  FOR i := 0 TO MaxFile-1 DO
    fileTable [i].used := FALSE;
  END;

  (* now install the termination procedure (CloseAllFiles) *)

  InstallInitProc ( InitProcedure )

END FileHandler.
```

Another way to generate private resources is to link the corresponding module with those overlays where the resource has to be private, and take the one linked with the **Base** for those where the resource is supposed to be shared.

2.9.6 Creating PROCESSES in Overlays

PROCESSES can be created using the standard procedure **NEWPROCESS** in subprograms (but not in resident overlays). These processes can load other overlays, when they have the control.

Upon the unloading of the overlay, processes which were created in the overlay itself are destroyed. It is also no longer possible to **TRANSFER** to such a process. Transferring to a dead process will usually hang the system. If such a process has loaded overlays, these overlays will be unloaded from memory.

Notes:

Chapter 3
The Symbolic Run-Time Debugger

LOGITECH Modula-2 works with two complementary debuggers: the ***Symbolic Post-Mortem Debugger*** (referred to as the ***LOGITECH PMD***) is described in the ***LOGITECH Modula-2 User's Manual.*** The ***Symbolic Run-Time Debugger*** which is described in this chapter, is referred to as the ***LOGITECH RTD***, or simply the ***RTD.***

The ***RTD*** lets you watch how a program runs.

You can execute the program step by step. After each step, you may then inspect the data and the current status of the program. There are several ways you can step through the program. Depending on the situation, you may execute larger or smaller steps. You can also modify the values of the variables the program uses.

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

3.1 LOGITECH RTD Files

On the ***RTD*** diskettes you will find the file RTD.ARC. When you extract the files as explained in the **Installation Chapter**, or run INSTALL, you will obtain the following:

RTDPAR.CFG
RTD.EXE
RTDINIT.OVL
RTDM2.OVL
RTDOVLAY.OVL

The ***RTD*** will also use the following files from the ***PMD***:

MDA.CFG
CGA.CFG
EGA.CFG
DB.CFG
DB.HLP

The file extensions stand for:

.CFG	Configuration file
.HLP	Help file
.EXE	***RTD*** executable file
.OVL	***RTD*** overlay file

3.2 The RTD and Your Hardware

The *LOGITECH RTD* works on *IBM PC* and compatible computers. It is advised to run the *RTD* on a **512**K or **640**K system and with a mouse. The debugger can work with a MDA, CGA or EGA video controller. With a CGA or EGA controller, the *RTD* can have windows with colors. With an EGA controller, the *RTD* can have a screen with **43** lines.

The *RTD* can also be used without a mouse. If a mouse is used, it is important to use recent mouse drivers (*LOGITECH Mouse driver Version 3.20* (or higher), *Microsoft Mouse driver Version 6.00* (or higher)).

3.2.1 Memory Requirements and Swapping

The *RTD* requires approximately **275** Kbytes of memory to run. The remaining memory can be used by the program being debugged. In order to determine the maximum size of the program that can be debugged, you have to add the size of your *DOS*, and the size of every other resident program (Mouse Drivers,...) you have loaded: For example:

RTD	**275** Kbytes
Dos 3.2	**20** Kbytes
LOGITECH Mouse driver	**9** Kbytes
	304 Kbytes

304 Kbyte is the memory used before your application can be loaded. The requirement of **275** Kbytes includes only the *RTD*.

With the Big swap option, it is possible to enlarge the memory space available to the program being debugged. The *RTD* requires as little as **125K** to run. **Refer to Section 3.6 RTD Options** for details.

When you specify the a swap option, parts of the *RTD* are loaded into memory as needed. When the program has been stopped, the program is swapped out to disk. It will be swapped back into memory as soon as you resume execution.

When you choose the Big swap option, the debugger creates the two swap files RTDSWAP.RTD and RTDPROG.RTD. They will be created in the current directory of the current drive if the "**V**" option is off, or they will be created in the virtual drive if the "**V**" option is on. Both files have a fixed size of approximately **150K** bytes. Therefore, when using the Big swap option you should make sure **300K** bytes of disk space are available.

3.3 How to Run the Run-Time Debugger

To run the *LOGITECH RTD*, type:

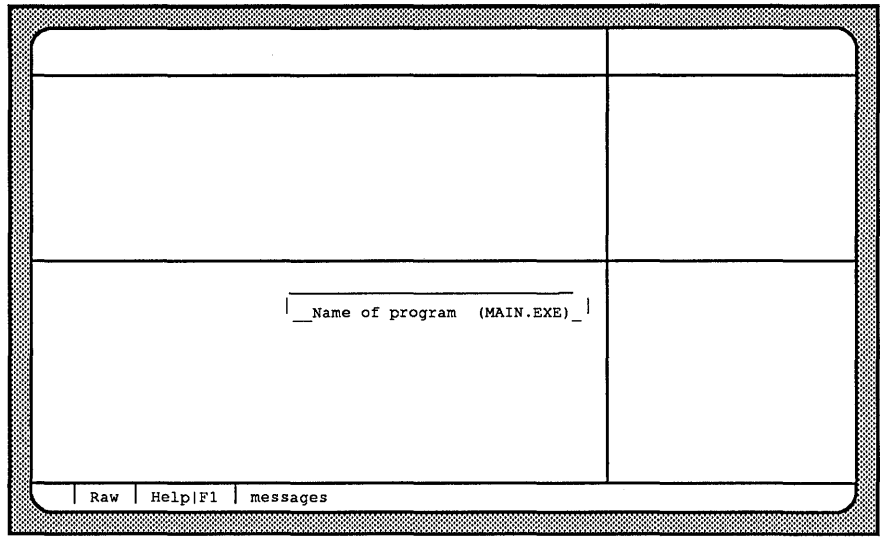
RTD ↵

The debugger responds with a sign-on message:

```
MODULA-2/86 Run-Time Debugger
```

followed by the version number and a copyright notice.

You will see the following screen prompting you for the name of the program you wish to debug:



Enter the name of the executable file (<**filetype**>.EXE) followed by ↵. The ***RTD*** will then load your program into memory, and update each window on the screen with the appropriate information. 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. **Refer to Section 3.4.3 and Section 3.8 for more information on Go commands.**

The ***RTD*** is started by just typing **RTD** [↵]. The name of the file to debug can be specified on the command line or added when the debugger prompts.

RTD options should appear just after the name of the debugger. If the application debugged in the ***RTD*** wants options, just type the options after the name of the application.

RTD Version 3.0 uses the .MAP file of the program.

IMPORTANT !!!

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

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

3.3.1 Programs Taking Command Line Arguments

With the ***LOGITECH RTD***, you can also debug programs that accept command line arguments. When the ***RTD*** asks for the program to be debugged, enter the arguments in the usual way. For example:

Assume the program "**mycopy**" is normally started under ***DOS*** by entering:

<mycopy> <file1> <file2> [↵]

With the ***RTD***, the following will start the program in the same way:

RTD <mycopy> <file1> <file2> [↵]

3.4 Control of Program Execution

There are two ways you can control the program being debugged. One is to set breakpoints on specific statements in 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 you can inspect and modify the content of variables in any part of the program. You can also examine any process, and you can view or change the data of any module or any active procedure.

3.4.1 Breakpoints

One way to monitor program execution is to tell the ***RTD*** at which points to stop. These are called breakpoints. When the program executes a statement at a breakpoint, the program stops and you may examine the data structures and the status of the program.

You may set a breakpoint on any program statement. The ***RTD*** 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. Each time you set a breakpoint, the debugger prompts you to specify a limit for the occurrence counter. This 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: Set the limit of the counter for a particular breakpoint to five. The ***RTD*** 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 additional time this breakpoint is encountered.

3.4.2 Step Mode

You can also tell the ***RTD*** 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 can examine its current status and data.

3.4.3 Overview of the Run-Time Debugger Commands

Six commands clearly distinguish the ***RTD*** from the ***PMD***. These commands let you control program execution by stopping at specific points in the program. Whenever you stop the program, you can examine its current status, as well as display and modify its data. In this way you can determine more specifically the location and cause of problems in your program.

The six 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**.

GB **Go Breakpoint**

Stop at the next breakpoint or overlay loading.

GS **Go Statement**

Stop at the next statement, breakpoint, or overlay loading.

GP **Go Procedure**

Stop on the next procedure call, breakpoint, or overlay loading.

GR **Go Return**

Stop on the return from the current procedure, or the next breakpoint, or overlay loading.

GE **Go End**

Execute the program until the end, ignoring breakpoints, but not the overlay loading.

GF **Go Flat**

Same as **GS**, but it steps over a procedure call without going into the procedure.

3.4.4 Run-Time Errors

When a run-time error occurs in the program being debugged or when the program calls the standard procedure HALT, the ***RTD*** gains control, updates the windows and displays an error message. No memory dump (**<filename>** .PMD) is generated. The ***RTD*** also indicates in the Call window the cause of the run-time error. You can now inspect the program and you can resume execution. It will continue with the termination routines of the terminated layer.

3.4.5 Stopping Programs During Execution

A program being debugged, with or without the ***RTD***, should import the **Break** module, 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 ***RTD*** handles this situation in the same way as when a run-time error occurs, and you can inspect the status and the data of the program as they were, when Ctrl-C or Ctrl-Break was typed.

If a program that does not contain the **Break** module is stopped by Ctrl-C or Ctrl-Break, the ***RTD*** will not be able to analyze the current program state. However as in the case of run-time errors you can follow the termination of the current overlay.

3.4.6 Debugging Programs that Use Overlays

When an overlay is called, the ***RTD*** stops the execution when it is loaded, but before it has started execution. This is similar to what happens when you start debugging a program. The windows get updated when the overlay has been loaded. You may then set breakpoints or start the execution of the overlay in step mode.

For all these files, the search strategy is the same: first the debugger looks into the current directory and then in the directory from where the program to debug was loaded.

3.5 RTD Configuration

The ***RTD*** reads files during its initialization. It reads a file which contains the layout of the screen, a file which contains information displayed in the help window, and then it reads overlays.

3.5.1 Screen configuration

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

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

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

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

3.5.2 On-line Help

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

NOTE

For both DB files, the search strategy is the same: first the debugger looks into the current directory and then in the directory from where the ***RTD*** was loaded.

3.6 Run-Time Debugger Options

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

RTD/Q/S ↵

when starting the ***RTD***.

The ***RTD*** accepts file-related, memory-related, mouse-related, and screen-related options.

3.6.1 File-related Option

/Q

(default : **/Q-**)
Query

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

3.6.2 Memory-related Options

These two options let the debugger and the application share the same memory area. The debugger uses about **275**K without either swap.

/S (default : **/S-**)
Small swap — uses **225**K

/B (default : **/B-**)
Big swap — uses **125**K

For a faster swap, use the following:

/V (default : **/V-**)
Virtual disk

Saves swap files on drive **D:**. In a computer with ***AboveBoard*** memory, you can use **VDISK** as virtual disk **D:**. Other drives may be specified in the RTDPAR.CFG file.

/L (default : **/L-**)
Large

Enlarges the internal workspace of the ***RTD***. 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 with many modules, the default workspace of the ***RTD*** may be too small. This would lead to a stack or heap overflow in the debugger itself. You can specify the amount of workspace the ***RTD*** has to use in the file RTDPAR.CFG.

3.6.3 Mouse-related Options

/M (default : **/M-**)
Mouse

Use this only when the application uses an old mouse driver. The reason is that the old mouse drivers are unable to switch context so that two applications can use the mouse simultaneously. If **/M+** is used, the mouse is not used by the debugger, which lets the application use the mouse without conflict.

3.6.4 Screen-handling Options

/G (default : **/G-**)
Graphics

Use this for conflict between the debugger screen and the application screen. Try the application with **G-** (default). If a problem occurs (messy screen), restart the debugger with **G+**.

The controller can be MDA, CGA, or EGA. If EGA is used, it can be used in MDA mode with a monochrome display (see EGAm, below), in CGA mode with a color display (see EGAc) or in EGA mode with a color display (see EGAe).

Two methods are used for screen saving/restoring. With **G-**, the debugger writes its output in one page and the applications output in another page. **G+** saves the application/debugger screen to disk. **/V** saves these files on the same virtual drive as the swap files.

Sometimes the debugger detects a method that won't work with the screen used. Then you must try **G-** and then **G+** if you don't want this kind of problem.

MDA: **G+** is forced by the debugger

CGA: **G-** uses page **1** and the application may use pages **0**, **2**, **3**.
G+ allows the application to use all the pages.

EGAm: same as CGA

EGAc: same as CGA

EGAe: **G+** is forced by the debugger.

3.6.5 RTD Option File

You can change the ***RTD*** option default values by modifying a text file named RTDPAR.CFG. Keep this file in the directory where RTD.EXE resides or in your work directory. You can also put it into any sub-directory that is identified by the environment variable **M2CFG**.

Typing options on the command line will overwrite the values specified in RTDPAR.CFG.

In RTDPAR.CFG there are two additional options not available on the command line:

A memory-related option:

/W=KK Workspace
(default : **/W=32**)

Specifies the ***RTD*** workspace, expressed in KBytes from **16-64** KBytes. **/W** is only meaningful if the **/L** option is on.

A file-related option:

/P="CCC" Path
(default : **/P="D:"**

Specifies the drive and path of the directory where the ***RTD*** saves internal temporary files. Applicable only if the **/V** option is on.

/D (default : **/D+**)

Tells the debugger not to maintain the application screen. Use this switch if you have an application without any input/output or do not want flashes while screen switching (**/G-**) or have time spent while saving/restoring screen (**/G+**). See **/G** option for more details).

With **D-**, the debugger writes on page **0** with a MDA screen or in page **1** with a CGA or EGA screen. In this case, it will not switch to page **0** when the application runs and you will see something only if the application writes in the current page. If the application switches to another page, the debugger will not show anything. If **/D-** is used, the command **=A** is not available.

3.7 User interface

3.7.1 Windows

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

The windows can have two states: opened or iconized. An opened window shows its contents. An iconized window is displayed as a label on the last line of the screen.

In the following text an open window may be referred to as "window", and an iconized window as "icon".

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

One window is always active (this means that the menu called is connected to the active window). It is also possible to activate an icon so that its menu is available. The activation of the icon does not open it as a window.

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

Window functions are:

- activate a window
- scrolling of the contents
- color modification:
    - for the borders
    - for the window contents
    - for the menus
- size modification:
    - window borders moved (because they share a screen, the motion of a border modifies the size of adjacent windows)
    - window modified to fill the whole screen (zoom)
    - window iconized (shrink)
    - window swapping (to exchange the position/size of two windows). This command can also be applied between an icon and a window, but not between two icons.

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

3.7.2 Mouse Functions

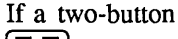
The mouse button is context sensitive. The table below describes these meanings.

Cursor position			
cursor on a window's left border	scroll up	vertical absolute position	scroll down
cursor on a window's bottom border	scroll left	horizontal absolute position	scroll right
cursor on a window's bottom left corner	move left/bottom border	------	------
cursor inside a window	simple click: select double click: carry out the most probable action on the selection	call window's manipulation menu	call window's specific menu
prompt	terminate user entry	escape prompt	escape prompt
menu	execute the highlighted action	execute the highlighted action	execute the highlighted action

Note:

Scroll functions: Similar to those in *LOGITECH POINT* or *Microsoft Word.*

Attempting to scroll beyond the ends causes a beep.

If a two-button mouse is used,  can be emulated by  .

Click:  inside the window you wish to select. If you click  on an icon, it does not expand the icon but lets you access its local menu.

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

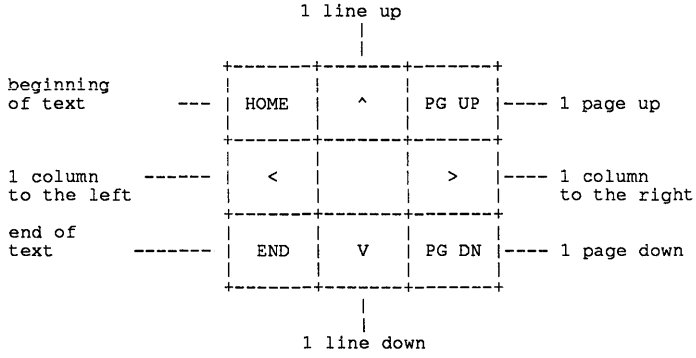
eXchange windows: First select a window then point at the other window with the mouse and select the e**X**change command in the window menu. This command moves the active window in the selected window.

The other window commands are available via the menu.

3.7.3 Keyboard Functions

3.7.3.1 How to scroll

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



TAB : 1 page right

SHIFT-TAB : 1 page left

3.7.3.2 Select a window object

To select any window object, move the cursor above the object and press Spacebar or ↵ .

To activate a window, use a window activation command.

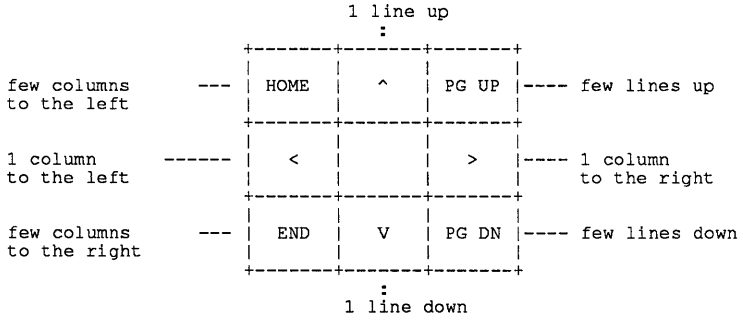
3.7.3.3 Call the menu

[F10] displays or erases the menu.

[↵] or [Spacebar] validates the selected item in the menu.

[Esc] leaves the menu without performing the action.

Menu bar The menu bar can be moved up or down. The cursor laypad is mapped as follows for bar moving:



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

◆ means [Alt]
^ means [Ctrl]
the musical note sign means [■ ❑ ❑] double click.

Off-menu, [Esc] purges the keyboard input buffer. Wrong keystrokes are beeped, if beep is ON. [↵] and [Spacebar] activate the selected command.

3.7.3.4 Respond to a prompt

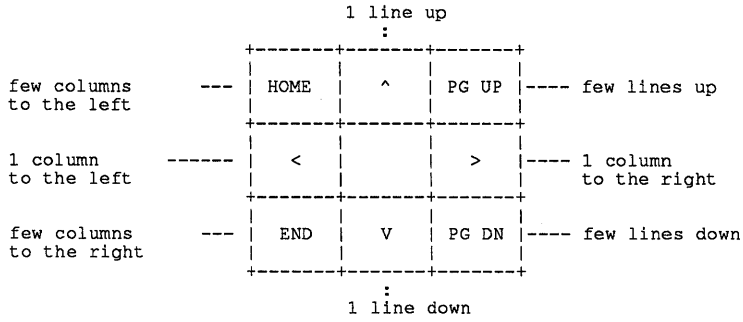
[↵] or [Spacebar]	validates the characters entered to the debugger.
[Esc]	aborts the input processing.
[Ctrl]-[X]	erases all the input characters.
[← Back]	erases the last character input.

NOTE

[■□□] acts as [↵].
[□■□] and [□□■] act as [Esc].

3.7.3.5 Move the mouse cursor with the keyboard

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



3.8 Windows and Commands

The *LOGITECH RTD* displays these windows:

Call:	Show the calling chain of your program
Module:	Show list of the modules in memory
Text:	*Modula-2* source file
Data:	Data defined in a module or procedure
Raw:	Direct access to the memory
Help:	Displays the contents of DB.HLP (help file)
Message:	Displays messages of the *RTD* (like which file is accessed)
Application:	Screen of the application (applicable only if the **/D** option is on)

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

Quit command

Q This command lets you quit the *RTD*. If the screen layout was changed during the session and not saved, you can save the configuration at that time.

Go commands

GB **Go Breakpoint**

Instructs the debugger to execute the program until the next breakpoint or the next overlay loading.

GE **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:

GF **Go Flat**

Same as GS but it steps over a procedure call without entering the procedure.

GP **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. Note: at this point, some procedure parameters may not be initialized.

GR **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.

GS **Go Statement**

Instructs the debugger to execute the program until the next statement, or to the next breakpoint, or to the next overlay loading.

If the program to be debugged contains more than one process, the step mode is only applicable to one process at a time.**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.

The ***RTD*** always shows the stack of the process where the breakpoint was incountered.

Window activation commands:

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

[=][C]	Selects **C**all window
[=][M]	Selects **M**odule window
[=][D]	Selects **D**ata window
[=][T]	Selects **T**ext window
[=][R]	Selects **R**aw window
[=][H]	Selects **H**elp window
[=][S]	Selects me**S**sage window
[=][A]	Selects **A**pplication window

This is also possible either by clicking into the window or via menus. The fact to activate an icon does not open it but lets you access its menu. To open an icon can be done by a window manipulation command.

[F1] opens the help window (full screen). [F1] exits the help window.

Window manipulation commands:

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

Zoom [Alt]-[Z] expands a window to full screen. It also returns the window its original size.

Windows can have two states: opened or iconized. A window is iconized (or shrunk [Alt]-[S]) when only its label is visible on the last line of the screen. You can open an icon with one of the two commands: vertical expand [Alt]-[V], horizontal expand [Alt]-[H]. This distinction occurs because of the window allocation policy: a window can be opened only in an already visible window.

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

Configuration commands:

Alt-T	Lets you access a menu by which you can change colors:
	Alt-B, Alt-G — Border colors.
	Alt-L, Alt-O — Menu colors.
	Alt-C, Alt-D — Window content colors.
	Alt-N — Turns a bell on/off.
Alt-F	Lets you save the configuration. If you do not save after a change, you will be prompted when you will leave the debugger.

Low level commands:

Alt-R	Lets you redraw all windows.
Alt-P	Lets you center the current selection.

Menu commands:

F10	Invokes and exits the menu (toggle).

3.8.1 Call Window

The Call window displays the chain of procedure calls of a process. The Call window displays: `"No call chain"`, before you have started the program with the **Go** command because no procedure of the program is active.

Local commands in the Call window:

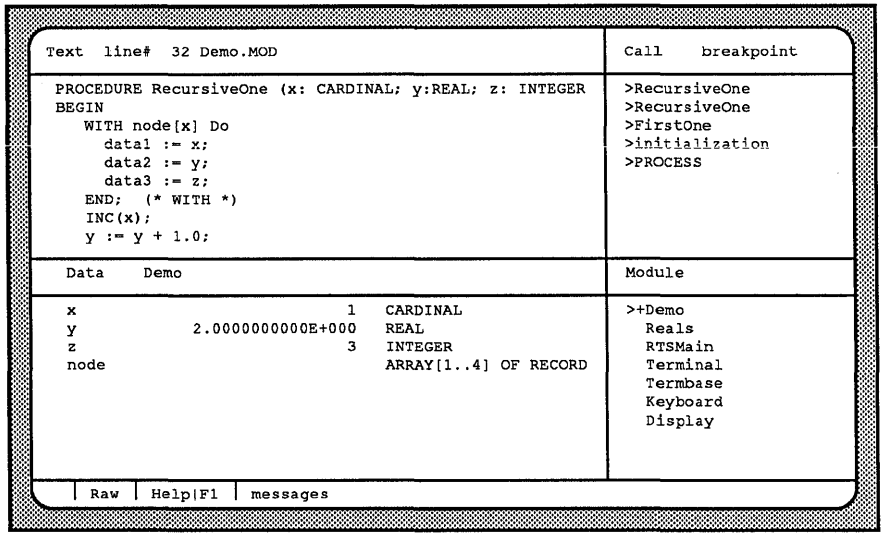
Address:	Gives the address and line number of the executed statement.
Examine break process:	Updates all windows with the information related to the process running when the program stopped.
Data:	Updates the data window with the data of the selected element (**PROCEDURE** or **PROCESS**).
Text:	Updates the text window with the text of the selected element (**PROCEDURE** or **PROCESS**).
Both:	Executes commands Data and Text or is equivalent to the double click on the selected item.

NOTE

You can see the contents of the **PROCESS** only if RTSMAIN.REF is available.

The following example shows the default setup of the windows on the screen.

The application screen will substitute this when either (=) (A) is entered or when such a window is selected via menus.



Sample Screen 3-1

3.8.2 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

Find:	Lets you search for a module. Wildcard * and ? characters are accepted with the same syntax as ***DOS***. You can reflect the next module name matching the input pattern by selecting the find command again and pressing [↵] .
Address:	Gives the data and the code addresses of the module, and updates the raw window..
Step mode **E**nable:	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.
Step mode **D**isable:	Disables the step mode in the selected module. For all modules of the system library, the step mode is disabled by default.
Step mode **A**lldisable:	Disables the step mode in all the modules.
Data:	Updates the data window with the data of the selected element.
Text:	Updates the text window with the text of the selected element.
Both:	Executes commands Data and Text or is equivalent to the double click on the selected item.

3.8.3 Data Window

The Data window displays the variables and/or parameters of the selected procedure or module.

Local Commands in the Data Window

Son and Father: Displays the data structure beneath the current level for the selected item. If the selected item is an array, the Son command displays the values of the elements of the array. If the selected variable is a record, the Son command displays the names and values of the record fields. Likewise, local modules are shown as data of the embedding module. You can also examine the content of the process descriptor by entering the Son command when a variable of type "PROCESS" is selected. In addition, this command can be used to follow linked lists when you select a variable which is a pointer or is of type "ADDRESS". The double click applies these functions. The command Son is applied when an element is "double clicked". The command Father is applied when the path on the top of the window is "double-clicked". The command Son can be used on a variable of type PROCESS only if the file RTSMAIN.REF is available.

eXchange: Lets you switch from procedure local data to module global data and vice versa.

Right/Left: These commands are only applicable when the selected data item is an element of an array, or part of an element of an array. The Right and Left commands select the element with the next higher or lower index in the array. The current level is not changed by these commands. If the array elements are records, the record field selected is not affected.

Index: Lets you select randomly an element of an array by giving the value of its index.

Type transfer: Lets you change the type of a displayed variable. You can use a predefined or user-defined type. If no type is given, the variable is displayed with its original type. The Type transfer is allowed only if the type of the variable and the new type are of the same size. If you use a type of your own, the debugger prompts you for the module defining it. A type-changed variable is marked by a "**T**".

Variable: Returns to the first level of the selected procedure or module. The first level shows the variables of the procedure or module. The Variables command can be used after you have repeatedly entered the Son command and wish to return to the first level directly, without repeatedly entering the Father command.

Examine PROCESS: This command can be used when you select a variable of type **PROCESS**. Otherwise, the ***RTD*** prompts you to introduce the address of the process descriptor - the content of a variable of type **PROCESS**. The Examine command displays the call chain of the process to be examined. Enter the call window command Examine break process to show the Call window of the process that was running when the program stopped. Checks to see if the process is initialized (a word with a special pattern is in all process descriptor).

Address: Displays the address of the selected data item and updates the Raw window.

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

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 **:<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 a **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 ***RTD*** 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.

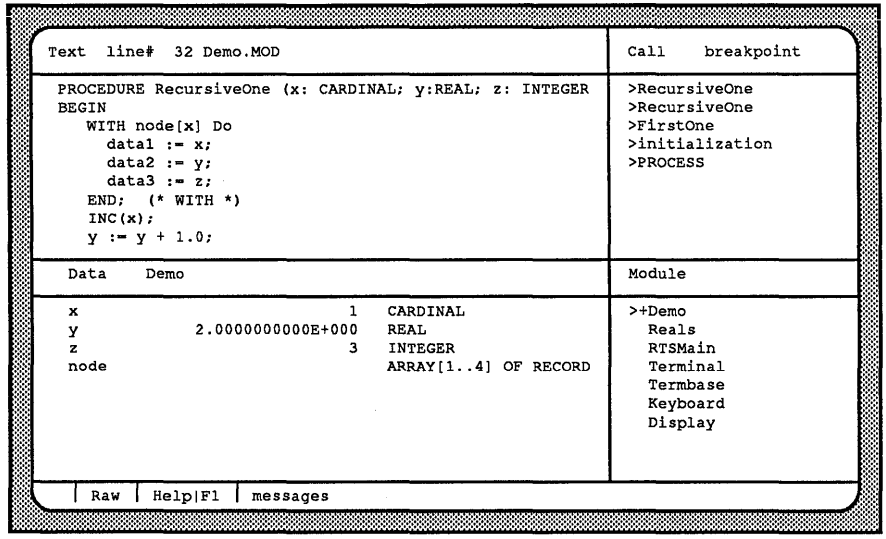
ENUMERATION

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.

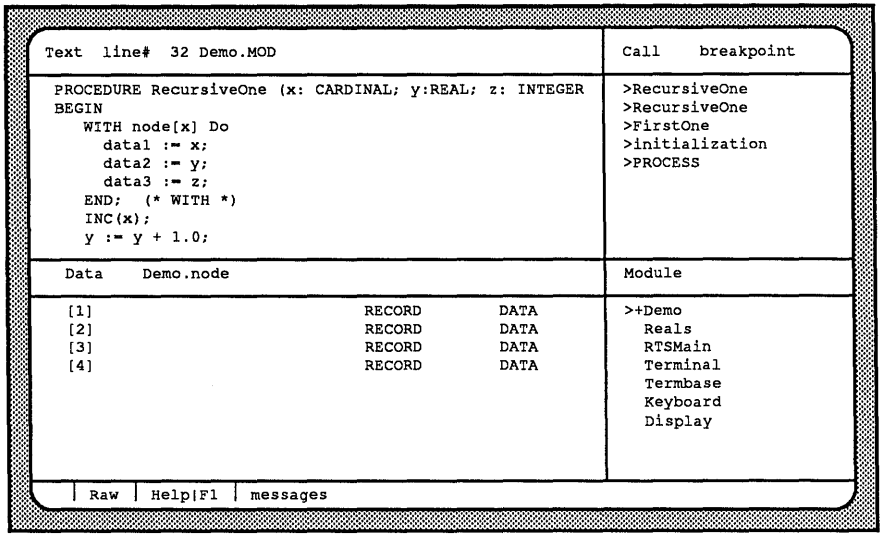
NOTE

Modify is not allowed on **LONGINT** and **REAL**.

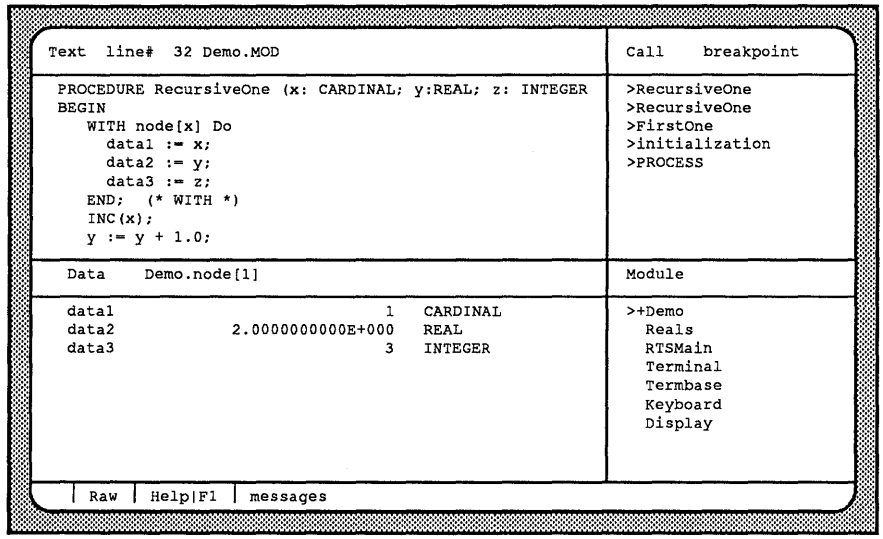
The following sample screens show the path you follow to modify the content of an array element with a record structure. First, you invoke the Son command to view the elements of the variable "**node**" of the module "**Demo**" (**Sample Screens 3-2 & 3-3**). Next, you again invoke the Son command to display the fields of the record "**node[1]**", and the value and type of each field (**Sample Screen 3-4**). Finally, you modify the value of the first field which is of type **CARDINAL**. You invoke the Modify command (**Sample Screen 3-5**) and enter a **6** to change the value from **1** to **6**. **Sample Screen 3-6** shows the modified data.



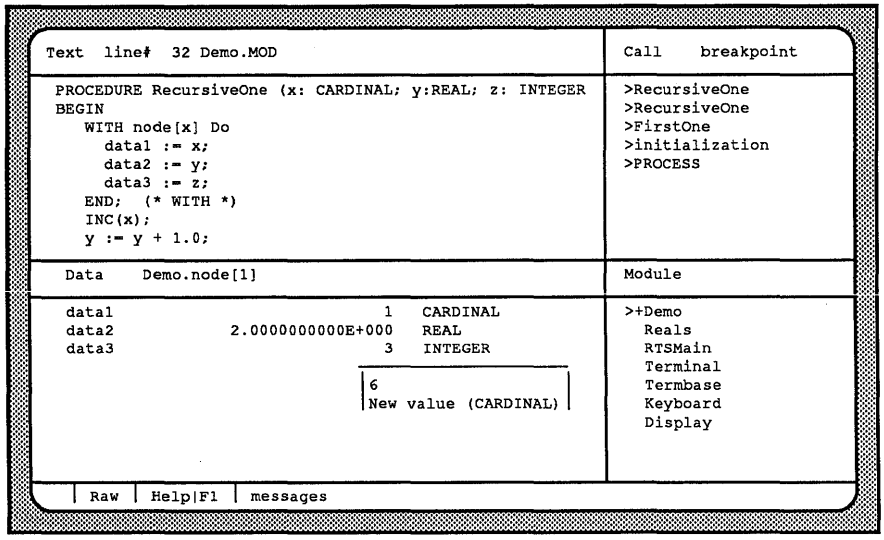
Sample Screen 3-2



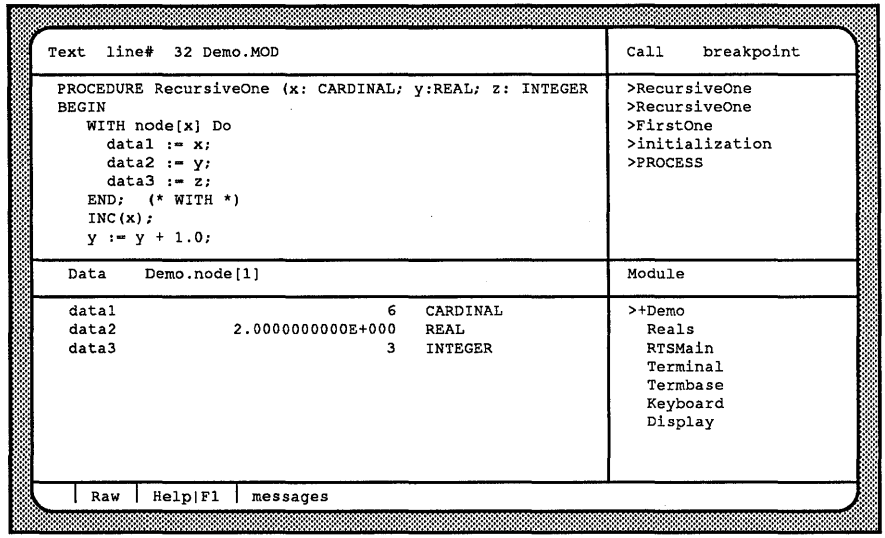
Sample Screen 3-3



Sample Screen 3-4



Sample Screen 3-5



Sample Screen 3-6

3.8.4 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

Find:	Prompts for a **PROCEDURE** or local module name (case sensitive), or a line number and, if found, the ***RTD*** sets the selected position to this **PROCEDURE**, module, or line number. This command allows you to enter either a line number or a procedure name. Wild characters are accepted (" * ", " ? ").
eXchange:	Lets you switch from MOD to DEF or DEF to MOD.
Address:	Show the code address of the selected source line and updates the Raw window.
Set bpt:	Sets a breakpoint in the selected line. If more than one statement is on the line, the ***RTD*** prompts you to indicate on which statement you wish to set the breakpoint. It also prompts you to set a limit for the occurrence counter associated with the breakpoint. You may type [↵] for the default value for this limit which is one. If a breakpoint is already set on the selected statement the ***RTD*** replaces the old value of the occurrence counter with the new one.
Clear bpt:	Removes a breakpoint on the selected line. If more than one statement is on the line, the ***RTD*** prompts you to indicate from which statement you wish the breakpoint to be removed.
Kill all bpt:	Removes all breakpoints from the program.
Go Line:	Lets you execute till the selected line is executed. A double click on the target line applies this function.

3.8.5 Raw Window

The Raw window displays the memory contents around a given address. The initial address of the selected memory location depends on the window from which you invoke the Raw window. The values are set the same way as in the ***PMD***.

Local Commands in the Raw Window

Address:	Lets you enter the address of data to be displayed.
Son:	Takes the contents of the selected memory location as the new selected address. You typically enter this command to follow a linked list. (dereferencing)
Examine PROCESS:	Assumes the memory contents at the selected address is of type "**PROCESS**" that is a pointer to a process descriptor. The Examine command displays the Call window of the process. The call window command Examine break process can be used to show the Call window of the process that was running when the program stopped. It also checks the check word of the process descriptor to check if a valid process is selected.
Modify:	Allows you to modify the memory contents at the selected address. The ***RTD*** asks you for the new vale 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. **REAL** and **LONGINT** cannot be modified
In/**O**ut **B**yte/**W**ord:	Used to read in and write out data through an I/O port. The ***RTD*** asks you to enter the address of the serial port which will be used.
Hexadecimal:	Decimal to hexadecimal conversion.
Decimal:	Hexadecimal to decimal conversion.

Various display modes:	**byte, word, address, char, text, cardinal, integer, longint, real**
#Byte:	**BYTE** (hexadecimal) format.
#Word:	**WORD** (hexadecimal) format. (default)
#Address:	**ADDRESS** (hexadecimal) format.
#Char:	**CHAR** (octal) format. Non-printable characters are displayed as octal numbers.
#Text:	**TEXT.** Non-printable characters are displayed as *IBM PC* extended characters.
#Integer:	**INTEGER.**
#Unsigned:	Unsigned **CARDINAL** format
#Real:	**REAL.**
#Longint:	**LONGINT** format

3.8.6 Message Window

Version: Lets you display the version of the debugger.

3.8.7 Application Window

The Application window shows the output of your application. It is a full screen window.

3.8.8 Markers

> (the greater-than sign) is used in the ***RTD*** as an execution marker to indicate active code. It appears in the Call, Module and Text windows.

In the Call, Module and Text windows, certain lines are marked with an (*) 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 where a program stops is marked with a pound sign (#) which replaces the asterisk.

In the Module window, the ***RTD*** marks modules where the step mode is enabled with a plus sign (+) preceding 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 selected.

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.

3.8.9 Selecting an Item for Display

The ***RTD*** displays the position of the selected item highlighting the proper line. You may select a different item using the cursor keys or the mouse.

3.8.10 Relation between Windows

The ***RTD*** displays different windows at the same time. This impacts what is shown and how selections are made. The Call and Module windows are mostly used for selecting text and/or data. You can select an element and use the double click (or the menu) to update the other windows.

3.8.10.1 Update made from the Call window

When a windows update is requested from the Call window, the ***RTD*** shows the data and/or the text of the selected procedure. The Raw window shows the contents of the stack.

3.8.10.2 Update made from the Module window

When a windows update is requested from the Module window, the ***RTD*** displays the data and/or the text of the current module. The Raw window shows the global data area.

3.8.10.3 Update from the Data and Text windows

You can modify the contents of the Raw window by using the command Address of the Data or the Text window.

3.9 Consistency Checks

The *RTD* does three consistency checks:

- between the code in memory (PMD file or EXE file) and the MAP file. This check is made by using keys stored in the code and referenced by a $OK label in the map.
- between the code in memory and the REF file. This test is made by using the keys stored in the code and a key stored in the REF.
- between the REF file and the MOD file. This check is made by using the date of the MOD (i.e. the date of the source file when it was compiled) stored in the REF file. If this date is not the same as the date of the MOD file read by the debugger, an inconsistency is signaled.

The inconsistency between the .MAP and the code in memory is very dangerous. It is extremely probable that all the information known from the .MAP is wrong. For this reason, the debugger does not display any symbolic information. It is strongly advised to relink your application.

An inconsistency between the .REF and the code in memory will make trouble only for the corresponding module. Depending on the changes made, only the data can be wrong or only the position displayed in the Text window or both can be wrong. It is advised to recompile this module and to relink the application (with a .MAP !)

An inconsistency between the .MOD and the .REF will make trouble when displaying the statement executed in the Text window. It is also advised to recompile this module and to relink the application.

NOTE

An additional check is made when a process descriptor or when an overlay descriptor is accessed (an overlay descriptor is also allocated for the main program). A field is initialized in the both descriptors with a specific value. If this value is not found, an error is signaled. This error means that you tried to analyze an incorrect part of the memory or that the memory was destroyed.

A test is also made when a .EXE is loaded to check if it is a ***Modula-2*** program. The debugger can debug programs only if the main is in ***Modula-2***.

3.10 Messages

******* Unexpected breakpoint**

An unexpected bpt was encountered. For example, this is the case when the application does a SWI(3).

Already as an icon

Already at top level

ASSERT: message

An internal error is detected into the debugger. The execution stops.

Beginning of this ARRAY

Call list incomplete (BP chain invalid)

A problem was encountered while reading the stack (memory destroyed ?)

Call list too long (>32)

This is a warning message which tells you that not all procedures on the stack are visible in the call list.

Can't be iconized

Can't expand in an icon

Cmd not allowed (use zoom)

Cmd not allowed on a window

Cmd not allowed on an icon

Cmd not valid in DEF MODULE

Color changed

Config changes NOT saved

You did not save the last screen configuration.

DEF file not found

End of this ARRAY

Go to line bpt, next bpt or overlay

Go to next bpt or overlay loading

Go to next PROCEDURE, bpt or overlay

Go to next statement, bpt or overlay

Go to next statement, proc, bpt or ovl

Go to return from PROC, next bpt or ovl

Go to the END of the pgm or overlay loading

Help file not found

Incorrect MAP file

An inconsistency is detected between the .MAP file and the code. This message appears and remains in the windows to warn you.

Incorrect REF file

An inconsistency was detected with the .REF file. A pop up window displays the reason the first time the inconsistency is detected. This message remains in the window to remind you.

Invalid call list

This message is displayed in a pop-up window when a problem is encountered while reading the call list (memory destroyed ?)

Invalid descriptor

The overlay descriptor has no valid check word (memory destroyed ?)

Invalid PROCESS descriptor

The process used as a parameter of the last command has no valid check word (memory destroyed ?)

Invalid process

The dump cannot be analyzed, the process descriptor of the process which crashed is invalid (memory destroyed ?)

MOD file not found

MODULE not found in list

New value out of range

No breakpoint to clear

[Text window, Clear breakpoint command]

No breakpoint is set at the selected statement, therefore it cannot be removed.

No call list

No data (unknown PROCEDURE)

No data in this element

No data in this local MODULE

No data in this PROCEDURE

No global data in this MODULE

No global or local data

No modif allowed in this mode

No modif till passed BEGIN

On a BEGIN, the space for the local variable is not allocated. For this reason it is not allowed to modify the variable.

No PROCESS during loading

Before the execution, the process descriptor is not initialized: no call chain can be shown.

No selected PROCEDURE

No statement in this line

This message appears for lines which state as labels (**LOOP**, **REPEAT**, etc...) and for removed procedures.

[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**" (except program and procedure **END**'s), "**IF**", "**CASE**", "**LOOP**" or similar is considered to contain no statement.

No text associated

No text for a PROCESS

Not enough memory

Memory problem. Use Small or Big swap option.

PROC/MODULE isn't in this text

REF file can't be re-opened

REF file not found

Swapping of icons not allowed

This border can't be moved

This data can't be modified

[Data window, Modify command]

This data cannot be modified. Usually this is for hidden types. In this case you should use the Son command to see the effective type and then you can modify the variable.

This data isn't an ARRAY

This data isn't structured

This data is of its original TYPE

To modify data use Son cmd

Too many MODULEs (> 256)

The debugger cannot debug programs with more than **256** modules.

Too small for expanding

TYPE not found in given MODULE

TYPE sizes differ

Wrong version of REF file: Bad structure

The REF file does not have the correct version (recompile the module and relink the application) or is too big.

Wrong version of REF file: Different from OBJ

An inconsistency was detected between the .REF and the code in memory.

Wrong version of text file

An inconsistency was detected between the .MOD and the .REF file.

Out of dump

Out of dump : = NIL

Out of dump : > 1 MB

Notes:

Chapter 4
The M2DECODE Utility

The ***M2DECODE*** utility decodes an object file (.OBJ) generated by the ***LOGITECH Modula-2 Compiler***. A text file is generated from the given OBJ file. This text file contains information such as imported modules, symbolic disassembled code and data areas.

Code is disassembled into standard Assembler language format. Exported procedure and exported data are identified by their name. If the ***LOGITECH Compiler*** was invoked using the switch **/SYMBOL**, all procedures and global data are identified by name.

If the generated output is large, the decoder may generate a multiple of decoded files. The filenames of the output are automatically generated, starting with the extensions .DC1, .DC2 and so on.

To decode a .OBJ file, type:

M2DECODE <example> ↵

M2DECODE will then ask you if you want the source lines to be put in the decoding. It may also ask you for the name of the source file (if it is not in the same directory as the object file). Source lines will be inserted in the disassembled code at the corresponding place.

NOTE

Source line can be inserted in the disassembled code only if the *LOGITECH Compiler* has generated the appropriate information (using the switch **/SYMBOL**)

Chapter 5
The M2VERS Source Manager Utility

M2VERS is a program that manages different versions of one program.

All modifications to a program are made in only one source code listing and the various versions can be derived automatically from that "master copy" using M2VERS.

NOTE

The "master copy" is itself the source code of one of the target versions and can be compiled without being processed by M2VERS. When another version must be produced, M2VERS is applied to the "master copy" and generates the converted source text.

We recommend that you always modify the same version of the program and generate the other versions from this original version. This 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.

5.1 Marking the M2VERS 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. M2VERS does not distinguish between upper and lower case.

Function	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 number 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 M2VERS.

5.2 Invoking the M2VERS Utility

Invoke M2VERS from the *DOS* command line by typing:

M2VERS ↵

The program asks you 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. A backup file is generated with the extension .VBK). Then M2VERS 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 commas. The program asks if the unselected parts of the master copy shall be deleted. Type Esc to exit the program.

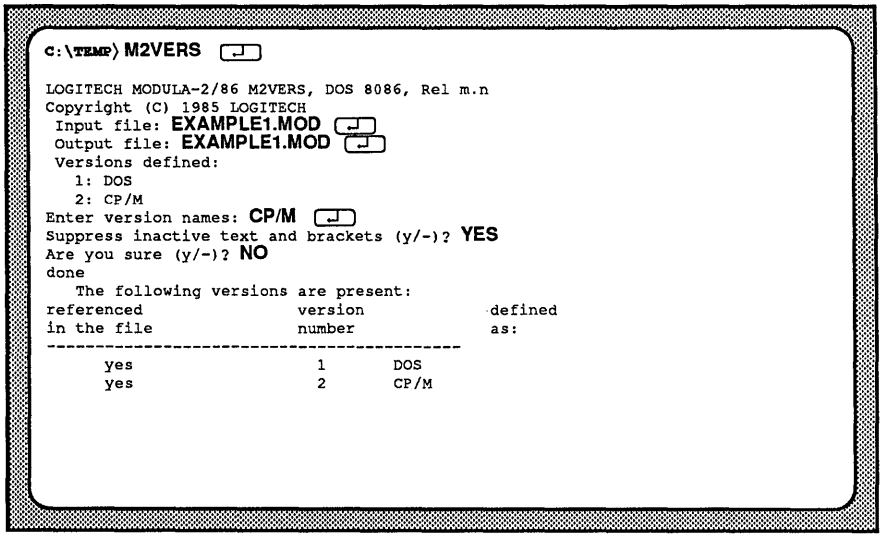
At the end of the processing, M2VERS indicates which defined versions were encountered in the file.

5.3 Example Dialog

To run M2VERS, for example, against a file named EXAMPLE1.MOD in a directory named **TEMP**, type:

M2VERS ↵

You will see the following:



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 the 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 .VBK extension.

/B- (Default)
/B+ This option indicates that M2VERS is used in a batch file. This option disables keyboard polling while the error listing is written to the screen.

Type: **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;
```

Figure 5.1 Updated Listing for EXAMPLE1.MOD

Type: **TYPE EXAMPLE1.VBK** ↵

```
(*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;
```

Figure 5.2 Backup Listing for EXAMPLE1.VBK

5.4 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 of source text 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

Warning messages only indicate that something is wrong. The program is not stopped.

`warning: can't activate version because of deactivated environment`

A portion of text which should be activated is nested in a deactivated one.

Error messages

Error messages do not stop the program execution, but no output file can be produced.

`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 M2VERS. The execution is stopped and the output file is deleted.

fatal error: too long symbol

An identifier that is too long is encountered. The maximum length of an identifier is **80** characters.

fatal error: buffer full

The input buffer is full. 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.

Notes:

Chapter 6
The Cross-Reference Utility M2XREF

M2XREF generates cross reference information tables of ***Modula-2*** source or listing files.

The program reads a text file and generates a table with line number references to all identifiers occurring in the text. All standard 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 as greater than lower case letters.

If the lines on the input file start with a number for example, and it is a listing file generated by the Compiler, ***M2XREF*** takes these numbers as referencing line numbers. Otherwise, ***M2XREF*** 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.

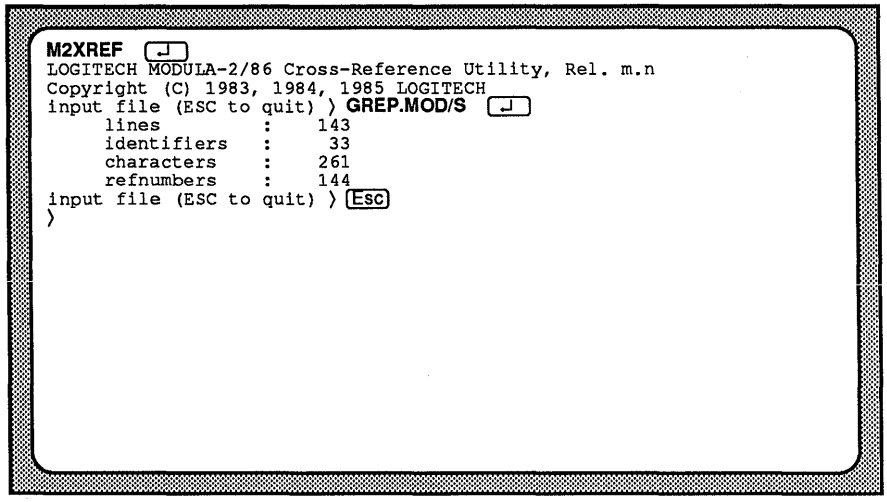


Figure 6.1 M2XREF Screen with /S Option

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.

Chapter 7
The M2MAKE Utility

The *M2MAKE* utility program is a valuable tool in the management of *LOGITECH Modula-2* programs containing large numbers of modules. It is used during source code maintenance or development. *M2MAKE* creates a batch file containing the minimal steps for recompilation required by an application to produce a new executable version when one or more changes have been made.

The *M2MAKE* utility is most easily run from a batch file under *DOS*. Considerations for building batch file are discussed in the next section.

LOGITECH Modula-2 source language programs contain all necessary dependency information, and are read directly by the *M2MAKE* utility. Therefore changes in module structure introduced by source program editing are recognized as soon as *M2MAKE* is used to rebuild a system.

One useful by-product of the *M2MAKE* utility is a cross-reference listing of inter-module relationships.

7.1 Features

M2MAKE has the following features and options:

- Command file generation can be user-tailored through the specification of command patterns for six different sections of the command file.
- The powerful command pattern facility allows cascaded make runs which can compile and link overlay systems in minimum time.
- The command pattern facility allows generation of other useful command files, such as to extract (backup) or print all source modules in a given application.
- Cross-reference listings of inter-module relationships in two different levels of detail are an optional by-product.
- Syntax errors in the import lists of all source files are reported. In this case no command file or cross-reference listing is generated.
- The same search strategy is used as for the compiler. Thus, modules in a hierarchy of directories and libraries can be processed.
- Recursive imports in definition modules are detected and reported. (Recursive imports in implementation modules are of course allowed).
- An autoquery option allows the specification of file names different from module names.
- A log option gives a filename and date comparison listing of the files under consideration.
- The name of the command file generated can be specified.
- During parameter entry, an interactive help facility details parameter options.
- Statistics relating to the number of source lines, statements, modules and procedures can be reported.

7.2 Usage

M2MAKE is typically useful in an environment where, in building or maintaining a collection of modules, .DEF and .MOD modules are periodically updated, and then it is necessary to partially recompile the whole system and relink it.

M2MAKE uses information from the directory timestamps to determine whether source files need recompiling. It is therefore essential that the files are edited on a system where the real-time clock is always running and set to the correct date and time.

7.3 How to Run M2MAKE

The *M2MAKE* program has to produce a command file which compiles a system of modules including and imported by the module MYPROG. For example:

Step 1: Type:

M2MAKE MYPROG [↵]

This will produce a *DOS* batch file CMDFILE.BAT in the current directory, which for example might contain:

```
m2c  import1.DEF/NOA/B import2.DEF/NOA/B
m2c  import1/NOA/B import2/NOA/B myprog/NOA/B
m2l  myprog
```

Step 2: Run batch file, type:

CMDFILE [↵]

This compiles and links all necessary modules. If these modules compile and link correctly, then running *M2MAKE* again produces an empty CMDFILE.BAT. If some modules fail to compile successfully, then run *M2MAKE* again to generate a command file for those modules.

Alternatively, *M2MAKE* can be invoked without the program name on the command line. With this method, it prompts for the program name.

After entering the program name, options may be entered. At any time where an option may be entered, a help facility is available, by typing [?]. Options may also be entered on the command line, but in this case the interactive help facility is not available or useful. The module name and option input is terminated by pressing [Spacebar] or [↵].

Options are preceded by a /(forward slash). Details of the options available, and their meanings are described in a section **7.5 M2MAKE Options**.

7.4 Invoking M2MAKE From a Batch File

This is the most automatic and convenient way to use the *M2MAKE* utility. It is possible to use more or less complex batch files depending on your own requirements, or the need to compile overlay systems. (To understand batch files, see the section on batch files in the *DOS* manual).

Step 1: Make a batch file called MYMAKE.BAT which contain:

```
m2make %1
cmdfile
```

Step 2: Store this in a directory specified in the *DOS* **PATH**, so that it can be used from any directory.

Step 3: Invoke *M2MAKE* using this batch file, by typing either

MYMAKE MYPROG <options> [↵]
or
MYMAKE [↵]

(and answer the prompt for Module Name and Options).

M2MAKE will construct the batch file necessary to do the compilation (CMDFILE.BAT), and the batch file MYMAKE.BAT will invoke CMDFILE.BAT.

Step 4: To review the batch file generated before allowing it to run, use a batch file containing:

```
m2make  %1
type    cmdfile.bat|more
pause
cmdfile
```

Example batch files for invoking M2MAKE:

Use a standard pattern file:

```
m2make %1/PF=\make.pat
cmdfile
```

Use the non-overlay compiler, multiple compilations per *DOS* command, and no run-time tests except stack overflow:

```
m2make %1/PD="m2c #.DEF/NOA/B "/PM="m2c/r-/t-/f-/NOA/B # "
cmdfile
```

Use standard .MOD and .DEF compile patterns, but bypass all implementation module compilations if any definition module compilations fail:

```
m2make %1/PB="if exist *.lst goto end"/PT=:end
cmdfile
```

7.5 M2MAKE Options

M2MAKE can accept a number of options, entered immediately following the filename. Each option begins with a / (forward slash), followed by the option letter (in upper or lower case), followed by a value. Enter main module name (without extension) followed by options:

<Module Name> <option>

Some options represent switches, in which case the value field is +(on) or, -(off).

Some options specify a filename, in which case the value is any valid *DOS* filename. If only a directory path is entered ending in \(backslash), a default filename is created on that subdirectory. If the filename extension is omitted, a default extension is supplied. Some options specify command file patterns, in which case the value is either a character string terminated by a / (forward slash), or [↵] , or a character string enclosed in single or double quotes.

NOTE

M2MAKE does not support spaces before option indicator " / ". Any space will be interpreted as a terminator for the command line.

The *M2MAKE* options are:

Switch options

/A Autoquery Option (Default **/A+**) Request Filename if not found.

This option specifies the action required when DEF/SYM or MOD/OBJ (non-library source/object pair) files are not found that use the module-name for the filename.

/A+ prompts you for a filename until one of the pairs is found with that filename or until you press [Esc], which indicates M2MAKE should continue without that file. A pathname may be entered with or without a filename. A pathname without a filename ends in " \ ", and the specified directory will be searched for the module. The new specified path becomes the master path for the remaining files of that module (i.e. if "**path**" is the answer to a prompt for a DEF/SYM pair, then "**path**" is used as the second directory (i.e. after the current directory) to search for the MOD/OBJ pair.) The **/I** option is disabled temporarily for remaining files of modules for which a prompt is issued.

With the **/A-** option, you are not prompted for missing files.

The action taken when files are missing is controlled by **/N** option.

/F+ **/F-** make full system; ignore dates

The default **/F-** causes a pattern generation for all modules which need compiling according to the relative file dates and module dependencies.

The **/F+** option generates the patterns for all files, regardless of the relative file dates, and in the order determined by the module dependencies.

/G Log option
(Default **/G-**) lo**G** the files scanned and checked.

Causes ***M2MAKE*** to write at the terminal the name of each file it is referencing. If the named file is found, its full name including directory path is written, together with its directory timestamp. If a file is not found, the timestamp (Last Modified) field is blank, and the filename without a directory path is logged. If both files of a DEF/SYM or MOD/OBJ pair are found, an indication of the relative ages of the files is also logged.

"**<**" indicates MOD or DEF older than OBJ or SYM;

"**>**" indicates younger.

The relative age is one of the criteria ***M2MAKE*** uses to decide which files need to be compiled.

/I+ **/I-** Ignore object in path deeper than source

This option is useful when the **M2xxx** environments are used to specify a hierarchy of directories with a private-library/master-library usage. With **/I+**, an object file is ignored by M2MAKE if it is further down the object search paths than the source is down the corresponding source paths.

For example, if we work in **\user** and most files are in **\master**, and we set all **M2xxx** to "**\master**", then all search paths become "**\master**" (ie current directory, followed by **\master**). If we start with a fully made system of modules in **\master**, and no files in the current directory (**\user**), and copy and update the source files D1.DEF and M1.MOD from **\master** to **\user** then when M2MAKE is run with the **/I** option, the files D1.SYM and M1.OBJ in **\master** are not seen by M2MAKE, so that it generates the patterns to compile M1.MOD and D1.DEF and all its dependencies. When the ***Compiler*** puts its outputs into the current directory, this leaves the made set of modules in **\master** intact. Further, modules can be independently updated and the system remade in **\master**, at which a further make run in the user directory will still generate all necessary compiles for the **\user** version with no impact on the **\master** version.

/M=filename[.map] /M= read .MAP file of made overlay base.

The specified .MAP file is read. Modules mentioned are not fully processed, as thay are assumed to be in the **Base**. Only their .SYM file dates are checked so that modules which import them, but are not in the **Base**, will be correctly compiled.

/L Library file - assume all "made"

/L = <filename>

This option forces M2MAKE not to check for some files that will be assumed to be already "made". The set of these files is discribed in the file specified by **<filename>**. The syntax used in this description file is as follows:

MODULE = <Modula-2 module names>

for each file that must not be checked. A typical use is for library files (like the standard library) which do not need to be checked for each run of M2MAKE.

If a file with the name "LIBRARY.MAK" exists on the default directory, *M2MAKE* will automatically use it as a description file. A description file is already provided with *M2MAKE*, which has the name LIBRARY.MAK. It contains the list of all the files of the library provided with the **LOGITECH** *Modula-2* system.

Multiple library description files may be specified and separated by commas.

/N geNerate pattern even if source is missing.
(Default **/N-**)

When a source/object pair is missing, or when a source is missing and its object date and import relationship to other modules indicates it requires compilation, the **/N** option indicates what action *M2MAKE* should take. With the default **/N-**, the compile pattern is not generated, but a message is issued instead. With **/N+**, the compilation pattern is generated, even though the source file is missing.

/T- Print some module sTatistics

Some statistics of number of modules, number of imports etc. are printed to the log. If the **/E** option is used, causing a full scan of all source modules, additional statistics including number of statements, procedures, internal modules, lines etc. are printed.

/X Cross-reference option
(Default **/X-**)

This option causes the *M2MAKE* program to write an inter-module cross-reference listing into a .XRM file. This listing is in three sections:

- An alphabetical listing of modules with modules they import.
- An alphabetical listing of modules with those modules which import them.
- An alphabetical listing by Identifier.Module of exported identifiers and the modules which import them. In this listing, "******" represents a qualified import of a module, e.g.:

```
IMPORT MOD1
```

as opposed to

```
FROM MOD1 IMPORT Ident.
```

The alphabetical listings are case-sensitive (upper case letters will precede lower case letters in the lexical order).

/E Extended Cross-Reference Option
(Default **/E-**)

This option causes the ***M2MAKE*** program to write a cross-reference listing similar to the **/X** option. This listing is extended in that under each identifier the source text of the identifier definition is printed.

NOTE

To achieve this, the ***M2MAKE*** program scans each definition and implementation module in its entirety (not just to the end of the IMPORT lists). This takes longer, and gives the possibility of detecting more syntax errors (which will prevent the command-file generation).

Identifiers which are exported but undeclared, and declared but unexported are highlighted in the extended cross-reference listing. **Note:** ***M2MAKE*** allows you to make a cross-reference of a suite of main modules which may share a common subset of modules. A list of main modules, separated by " , " should be given together with the **/X+** or **/E+** options. In this case no command file is generated.

Filename options

/C Command File Name option
(Default **/C**=CMDFILE.BAT. Default extension is .BAT)

Generates a command file with a user-selectable name.

/PF Pattern File Name option
(Default **/PF**=PROG.PAT, where PROG is the main module name; default extension is .PAT)

Enables a specified pattern file to be used for command file generation.

/S Extension of link output file option
(Default **/S**=EXE)

Lets you specify a linker output file extension other than .EXE. This is useful if using the absolute linker rather than the standard linker. In this case, use **/S**=H86.

The link output file is examined by ***M2MAKE*** to check its existence and timestamp in order to decide whether a link step should be generated.

Pattern Specification Options

The command file is generated in six sections. One way to specify a pattern for each section is to enter it as an option. (See the section on Pattern specification for more details). The six options for pattern specifiers are:

/PH Head of command file
/PD DEF module compilations
/PB Between DEF and MOD
/PM MOD module compilations
/PL Link step
/PT Tail of command file

Each option specifier may be followed by =, and then by a character string terminated either by /, [↵] , or enclosed in single or double quotes.

7.6 M2MAKE Pattern Specification

Command file generation is done in six separate sections. Individual specification of how to generate these sections is done with character string templates - one for each section. Each of the six character string templates may be specified in one of three ways:

- By specifying it in the options.
- By specifying it in a pattern file.
- By allowing it to take a default value.

For each of the command file sections, that template is used which is specified by the earliest of the above methods. For instance, if the MOD compilation pattern is specified as an option, any MOD compilation pattern specified in a pattern file is ignored, as is the default MOD compilation pattern.

The templates may contain directives to specify the filenames to be generated in the command file.

Is replaced by the directory path and filename prefix. The directory path is that determined for the file after finding it either via the search strategy or by autoquery.

^ Is replaced by the filename prefix only.

##

^^ Generates **#** or **^** characters respectively.

Letters associated with each of the six templates specify the pattern in the option list or the pattern file.

The six templates, and their associated letters are:

/PH Head of command file. Always copied to the beginning of the command file.

/PD DEF file compilation. Used each time a definition module is to be compiled.

/PB Between DEF and MOD section. *M2MAKE* generates all required definition module compilations before any implementation or program module compilations. The **PB** pattern is copied to the command file between the DEF and MOD compile sections of the command file.

/PM MOD file compilation. Used each time a MOD file is compiled.

/PL Link step of command file. Used to generate a link if *M2MAKE* deems this to be necessary.

/PT Tail section of command file. Always copied to the end of the command file.

In the DEF and MOD patterns, it is possible to generate multiple filenames on one command line up to the *Dos* limit of **127** characters. This allows use of the compiler feature which allows multiple compiles without reloading the compiler (speed improvement). To specify this, the section to be repeated for each file should be enclosed in braces " { ", " } ". For example, the pattern:

```
m2c {#.mode/s-}"
```

might generate:

```
m2c \path1\m1.MOD/s- \path2\m2.MOD/s-
```

Default values for all the pattern sections are:

```
PH=
PD="m2c #.DEF/NOA/B "
PB=
PM="m2c #/NOA/B "
PL="m2l #"
PT=
```

Each of the command file sections may be specified in a pattern file. A pattern file is used by *M2MAKE* if:

- It is specified by name in the **/PF** parameter, and is found.
- **/PF** is not specified, and a file with a name constructed from the main module name plus the extension .PAT is present in the current directory or on the M2PAT search path.

If a pattern file is used, each of the six command file section patterns begins in the pattern file after a line beginning with

.head	or **.h**	(i.e. only the first two characters are checked).
.def	or **.d**	
.between	or **.b**	
.mod	or **.m**	
.link	or **.l**	
.tail	or **.t**	

The pattern definition ends before the next pattern definition or end of the file.

A pattern file might contain:

```
.head - this is generated first
del *.lst
.def  - this is generated for each def compilation
m2c {#.def/NOA/B}
if exist #.lst goto error
.mod  - this is generated for each mod compilation
m2c {#/NOA/B/r-/t-}
.link
m2l #
.tail
if not exist *.lst goto end
:error
dir *.lst
:end
```

This example will generate a command file which will stop when a compilation error is detected in a definition module, but if all definition modules compile successfully, it will compile all implementation modules.

7.7 Search Strategy

The compiler uses a search strategy for finding .SYM files. The linker uses this same strategy for finding .OBJ files. To be compatible, the *M2MAKE* program uses this same search strategy for looking up all files (the files which are subject to the search strategy are DEF, SYM, MOD, OBJ, EXE, MAK, PAT). The strategy searches the following paths in order until it finds a required file:

1: The current directory.

2: The master path.

3: The **M2xxx** paths, set up by a prior *DOS* **SET** command, where XXX is the relevant file extension.

7.8 Compiling Overlay Systems

To build an overlay system and pattern, batch files can be set up so that one *M2MAKE* run per link is done. The base layer is built in the first invocation of *M2MAKE*, and the other layers are built in turn by subsequent invocations.

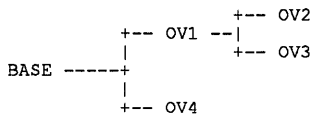
A scheme can be used where the generated command file is appended to the batch file which invokes the *M2MAKE* program. In this way, a loop in the command file can be established, and an arbitrary number of overlay layers can be built with a single command.

A batch file MAKEOVLY.BAT, and a copy of it MAKEOVLY.STD are provided for this purpose. To use it as provided, each of these files must be placed in the current directory, together with the file MAKEOVLY.PAT. The batch file may then be invoked with the following parameters:

parm1 - main base layer module name
parm2 - first overlay layer name
parm3 - base of next overlay layer
parm4 - next overlay layer module name

The pair parm3, parm4 may be repeated for as many layers as exist (except for the command line length limitation of *DOS*).

For example, if a system has the following overlay structure:



The whole system can be built with the command:

```
MAKEOVLY BASE OV1  OV1 OV2  OV1 OV3  BASE OV4
```

MAKEOVLY.BAT contains:

```
m2make %1/pl="m2l #"/pf=makeovly.pat
goto continue
:loop
if x%2 == x goto end
del %2.ovl
m2make %2/pl="m2l # (%1)"/pf=makeovly.pat/s=OVL
shift
shift
:continue
copy makeovly.std+cmdfile.bat makeovly.bat
```

MAKEOVLY.PAT contains:

```
.tail
goto loop
:end
```

MAKEOVLY.STD contains:

```
m2make %1/pl="m2l #"/pf=makeovly.pat
goto continue
:loop
if x%2 == x goto end
del %2.ovl
m2make %2/pl="m2l # (%1)"/pf=makeovly.pat/s=OVL
shift
shift
:continue
copy makeovly.std+cmdfile.bat makeovly.bat
```

Many variations on this theme of command and pattern files are possible. For an overlay system in which some specific layers require the link step to be fed with module names in answer to autoquery questions it may be more suitable to write a specific batch file to build each layer. The tail of each command file except the last could then invoke the batch file to build the next layer.

7.9 Program Operation

M2MAKE runs in five passes. At the start of each pass, an indicative message is written to the screen. These messages are:

- Reading File: **<filename>**.MAK, **<filename>**.PAT
- Reading Directories: **path*.xxx**
- Reading Source Modules and Comparing Timestamps
- Generating Command File
- Generating Cross Reference Listing

During the first pass, the library and pattern files are read.

During the second pass, derectories specified in the **M2xxx** environements are read for later use.

During the third pass, the source programs are read, starting with the main module, in order to determine all imported modules. Imported module source programs are read repeatedly, until all mentioned modules have been processed. Normally, (with **/E-**) the programs are read only as far as the first source taken beyond the import list for each module, as this is sufficient to completely define the module structure. In addition, presence of object files, and relative dates of object and source files are noted in this pass. These findings are written to the log if the **/G+** option is used.

During the second pass, the need for compilations to be performed is calculated, based on the module dependencies determined from reading the import lists.

A source (.DEF or .MOD) must be compiled if any of the following are true:

- its corresponding .OBJ does not exist.
- its .OBJ is older than its source.
- the .DEF file of an import is to be compiled.
- its .OBJ is older than the .SYM file of an import.
- (for a .MOD) its .OBJ is older than its SYM.
- (for a .MOD) its .DEF is to be compiled.

NOTE

When copying files from other systems, make sure that the dates of the files that overwrite the existing files reflect the desired version. Example: You buy an update of a special library. All files delivered are older than your application, but of course newer than the old version. *M2MAKE* will not recognize that the files are no good. Use option **/F** to update your application.

Source Language Syntax

The syntax of the source language accepted by *M2MAKE* is an extension of that detailed in the *EBNF* description of *Appendix I of Wirth's Programming In Modula-2, Third Edition.* The extensions allow use of **8086**-format address constants (segment:offset), both in a **ConstantDeclaration**, and optionally enclosed in brackets after each **ident** in the **IdentList** of a **VariableDeclaration.** These extensions are implemented as a superset of the syntax allowed by the compiler. *EBNF* for these additions is as follows:

```
 AddressOrConstExpr = ConstExpression  [ ":" ConstExpression ].
ConstantDeclaration = ident  "=" AddressOrConstExpr.
           VarIdent = Ident  [ "[" AddressOrConstExpr "]" ].
       VarIdentList = VarIdent  { "," VarIdent }.
VariableDeclaration = VarIdentList  ":"  type.
```

Without using either cross-reference option, *M2MAKE* parses **ProgramModule** and **DefinitionModule** and **ImplementationModule** as far as the end of the import declarations. With the **/X+** option, it parses **DefinitionModule** as far as the end of the export list, and with the **/E+** option, it parses all three modules completely.

7.10 Error Messages

The following error messages may occur. The environment in which each occurs is explained below.

`Identifier space exceeded - terminated`

There is an absolute limit of **63 K**bytes on the number of characters in a symbol table internal to the *M2MAKE* program. This message indicates that the table has overflowed. The message is generated during the Reading Import Lists pass, and causes immediate termination of the *M2MAKE* program.

The symbol table contains character strings. Character strings occupy **n+1** bytes in the symbol table, where **n** is the length of the character strings. Items entered in this symbol table are:

- each pattern specification
- each unique module name
- for every module, each filename
- for every module, each directory path
- if **/X+** or **/E+**, each exported or imported identifier.

`Syntax Error in xxx MODULE at line nnn`

Where **xxx** is IMPLEMENTATION, DEFINITION or PROGRAM, and **nnn** is a line number. Syntax errors detected while reading a source module are reported with this message to the log. In addition, a descriptive message for the error is logged. Such syntax errors cause further parsing of the current module to be abandoned. However, other modules continue to be processed.

The command file generation and cross reference listing passes are not executed.

no source file : name.ext

Where **name** is a module name, and **.ext** is .MOD or .DEF. *M2MAKE* has determined that it should compile the named module; however, the source for that module cannot be found through the search strategy.

A missing .DEF file may appear to require compilation only when its corresponding .SYM file cannot be located.

A missing .MOD file may appear to require compilation under the following conditions:

- its .OBJ file is missing
- its .OBJ file is older than its .SYM file
- its .DEF file is to be compiled

A missing .MOD file will not give rise to this error if its .OBJ file exists, is younger than its .SYM file, and the .DEF file is not to be compiled

This message is issued to the log. It is also placed in the generated command file at the point where the compilation pattern for the named module would normally be written. The compilation pattern is written to the command file in this error situation only if the **/R+** option is used. The error does not cause termination of command file generation or M2XREF listing generation.

imports missing from file : name.ext

M2MAKE has determined that compilation of the given module should fail because required .SYM files will not be available after previously generated compiles have run without error. It is always preceded by at least one message indicating a .DEF file missing.

This message is issued to the log. It is also placed in the generated command file at the point where the compilation pattern for the named module would normally be written. The compilation pattern is written to the command file in this error situation only if the **/N+** option is used. The error does not cause termination of command file generation or cross reference listing generation.

link step not generated - missing files

M2MAKE knows that a link step should fail because modules it expects to find are missing. It is written to the log, and also placed in the generated command file. It will always be preceded by at least one "file missing" message.

recursive import of DEF modules

A .DEF module is importing itself, either directly or indirectly. The path of modules through which the first-detected recursive import chain exists is then logged, one module per line, starting from the innermost. The batch file generation is terminated.

Notes:

Chapter 8
The M2CHECK Utility

M2CHECK helps you find errors in *Modula-2* programs. It reads and analyzes *Modula-2* source code and produces warning messages which indicate possible errors or "dangerous" codes. It also indicates unused variables, types, constants, and procedures.

A module to be processed by *M2CHECK* has to be a correct *Modula-2* module; *M2CHECK* produces error messages when it encounters syntax errors. Although execution continues, it *M2CHECK* may not find all problems. For this reason, use *M2CHECK* only to modules that have already been successfully compiled.

M2CHECK sends output both to your display screen and to a .LST file. This .LST file can then be called up through **Load Listing** in the **M2ASSIST** menu when the corresponding .MOD file is in the active window in the *POINT Editor*.

NOTE

Syntax errors are referenced by number. See **Section 8.3 Warning Messages** for the messages and their meanings.

Since none of the error-types detected by *M2CHECK* can appear in a definition module, *M2CHECK* can be used on implementation or "main" modules only. To completely check the usage of variables and other identifiers, *M2CHECK* needs to read the symbol files of imported modules as well as the symbol file of the implementation module. *M2CHECK* uses the same search strategy as the *LOGITECH Compiler*.

8.1 Running M2CHECK

To run *M2CHECK* type:

M2CHECK EXAMPLE ↵

You will get a screen that looks like the following:

```
m2check example ↵
LOGITECH Modula-2 M2CHECK, PC-DOS, Rel 3.0, (C), Jun 87
Copyright (C) 1987 LOGITECH.

  source file> Example.MOD
- pass 1
Example : C:Example.SYM
Terminal: C:\m2lib\sym\Terminal.SYM
- pass 2
- listing generation

167             exp := exp DIV 2;
*****             ^505
* 505: assignment to a value parameter

end analysis
```

After the filename, options may be added to adjust the exact operation of *M2CHECK* to your specific needs (see **Section 8.5 Options**). If the filename is omitted on the command line, *M2CHECK* will prompt for it. In this case, options may again be appended to the filename, separated by " / ".

8.2 Operational Errors

If *M2CHECK* cannot complete the checking of a module for any reason described above, it will issue one of the following messages, indicating operational errors:

```
- DEFINITION MODULE not checked

- ---- fatal error:  illegal symbol file
       (symbol file has a bad structure)
```

8.3 Warning Messages

Generation of the warning messages described in this section is the very purpose of the semantic checker M2CHECK. Most of the checks can be individually enabled or disabled through options, thus allowing to adapt M2CHECK to produce only those warnings which seem relevant to the user.

500: Identifier ambiguity

Occurs when several variables may be accessed, for example a global and a local variable with the same name, or two fields of two different records are visible at the same time because of the **WITH** statements.

501 : Identifier not referenced

Signals that an identifier (variable, type, procedure, etc) declared or imported in the implementation is not used.

502: Assignment to a FOR or WITH variabl

Occurs when an assignment is made to the variable of the **FOR** loop or to the variable of the **WITH** statement. Note that if assignment of the **FOR** variable is discouraged because of the possible side-effects, assignment to a **WITH** variable has unpredictable effect and is therefore very dangerous.

503: Variable referenced above

Occurs when a variable is used before its declaration. This may be right, but it may also signal that the programmer has forgotten to declare a local variable.

NOTE

Forward references are not allowed by some *Modula-2* compilers. Using them may reduce code portability.

504: Variable from another scope

Occurs when an access is made to a variable belonging to an embedding procedure or module, for example in nested procedures. This is again perfectly legal but the programmer could also have forgotten to declare a local variable.

505: Assignment to a value parameter

Occurs when an assignment is made to a formal procedure parameter passed by value.

506: PROCEDURE or hidden TYPE not implemented

The compiler does not signal which identifier is missing in the implementation while declared in definition, M2CHECK will list all missing identifiers in an implementation

507: Identifier declared in DEFINITION not referenced

Signals that an identifier (variable, type, procedure, etc) declared in the definition is not used inside the implementation.

8.4 DOS Error-level Variable

Upon termination, ***M2CHECK*** sets the ***MS-DOS*** error-level variable to one of the following values. This error-level variable can be checked in a batch file.

0: no error detected, no warning issued

1: execution completed, but warning(s) issued

2: symbol file missing

3: other operational error detected

8.5 Options

The following options are available:

Query:

Forces *M2CHECK* to ask for SYM files

/Q-(default)
/Q+

Listing:

Generates a complete listing with line numbers and error messages at the appropriate positions; if disabled, it will only produce the summary of the warning (on the display and on the listing file).

/L- (default)
/L+

Autoquery:

Enable *M2CHECK* to ask for .SYM file if it does not found it. This is useful while using batch files.

/A+ (default)
/A-

Scope checking:

Checks for accessing variables in other scopes. (warning **504**)

/S- (default)
/S+

Variable checking:

Checks for ambiguity during variable accessing. (warning **500**)

/V+ (default)
/V-

Reference checking:

Checks whether or not identifiers are referenced. (warning **501**)

/R+(default)
/R-

Definition module checking:

Checks for coherency between **DEFINITION** and **IMPLEMENTATION**

/D+(default)
/D-

Illegal assignment checking:

Checks for assignment to **FOR** or **WITH** variables. (warning **502**)

/I+(default)
/I-

Paramater assignment checking:

Checks for assignment to procedure parameter passed by value. (warning **505**)

/P+(default)
/P-

Notes:

INDEX

H

I

L

M

O

P

R

S

T

U

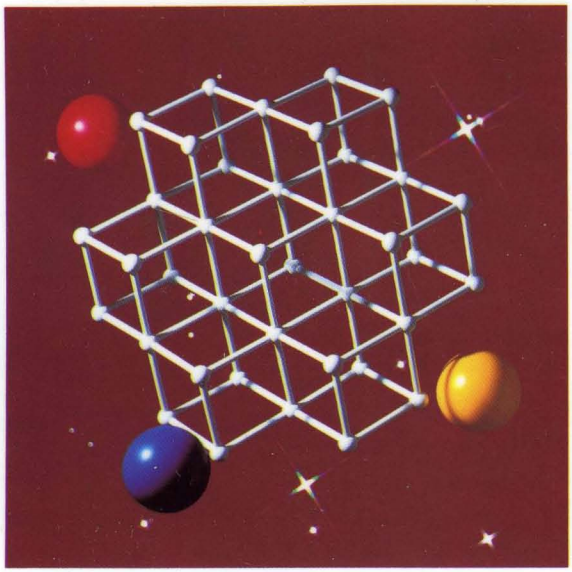
V

W

X

Notes:

LOGITECH MODULA-2™
VERSION 3.0



TOOLKIT

Linker	Decoder
Make Utility	Formatter
Library Sources	Disassembler
Run Time Debugger	Cross Reference Utility
Version	

LOGITECH

Logitech U.S.A.
Corporate Headquarters
6505 Kaiser Drive
Fremont, CA 94555
Tel: 415-795-8500

Logitech Switzerland
European Headquarters
CH-1111 Romanel/Morges
Switzerland
Tel: 41-21-869-9656

Logitech Taiwan
Far East Headquarters
15 R&D Road 2
Science Based Industrial Park
Hsinchu, Taiwan, ROC
Tel: 886-35-77-8241

Algol-Logitech Italy
Via Durazzo 2
20134 Milano MI
Italy
Tel: 39-2-215-5622