MacMETH 3.2

A Fast Modula-2 Language System
For the Apple Macintosh

User Manual

Departement Informatik
ETH Zürich
September 1992

Disclaimer

The authors of the computer software described in this manual hereby disclaim any and all guarantees and warranties on the software or its documentation, both expressed or implied. No liability of any form shall be assumed by the authors. Any user of this software uses it at his or her own risk.

This product is distributed on an "as is" basis; no fitness for any purpose whatsoever nor warranty of merchantability are claimed or implied.

The authors reserve the right to make changes, additions, and improvements to the software or documentation at any time without notice to any person or organization; no guarantee is made that further versions of either will be compatible with any other version.

Authors

Niklaus Wirth
Jürg Gutknecht
Werner Heiz
Hansruedi Schär
Hermann Seiler
Christian Vetterli
Andreas Fischlin

1st Edition, Release 2.0
Edited 1986 by Werner Heiz

2nd, completely revised Edition, Release 2.5
Edited 1988 by Thomas Wolf

3rd, revised Edition, Release 2.6
Edited 1991 by Hermann Seiler

4th, completely revised Edition, Release 3.2
Edited 1992 by Hermann Seiler

Copyright

Copyright (©) 1986 Departement Informatik
Copyright (©) 1988 Departement Informatik
Copyright (©) 1991 Departement Informatik
Copyright (©) 1992 Departement Informatik
Eidgenössische Technische Hochschule (ETH) Zürich
ETH Zentrum, CH-8092 Zürich, Switzerland

All rights reserved. Modification of this work by any means is forbidden without prior written consent of the Departement Informatik.

How to get MacMETH

The MacMETH System can be obtained via anonymous internet file transfer ftp (at no charge) from the host "*neptune.inf.ethz.ch*" (Internet address 129.132.101.33) in ftp directory "*/pub/macmeth*" or as part of the modeling and simulation package RAMSES from the host "*baikal.ethz.ch*" (Internet address 129.132.80.130) in ftp directory "/pub/mac/*RAMSES*".

Please use now the following weblink to download the latest release of MacMETH:
http://www.ito.umnw.ethz.ch/SysEcol/SimSoftware/SimSoftware.html

Contents

(Library Modules continued)

1. Introduction and Startup Guide

1.1. Introduction

This manual describes the implementation of the Modula-2 development system MacMETH. It is neither a reference manual nor a course about programming in Modula-2 (refer to [1] for an introduction to the language).

MacMETH has been developed in the group "Institut für Computersysteme", ETH Zürich, in co-operation with the "Rechenzentrum", ETH Zürich, and the "Fachgruppe Systemökologie" (Systems Ecology), ETH Zürich. The primary goal was to have a tool for teaching Modula-2 on Macintosh computers, but meanwhile MacMETH also forms the basis of large software projects.

1.2. System Description

System Features

The main features of the MacMETH system can be summarized as follows:

Compiler:
- single-pass Modula-2 compiler
- full language Modula-2 supported (restriction: declarations must precede references)
- basic compiler generates native MC68000 code
- extended compiler generates native MC68020, MC68881/2 and MC68040 code
- object code relocatable, no explicit linking necessary
- Toolbox calls generate inline traps
- each Modula-2 program can be converted into a Macintosh standalone application

Editor:
- simple program editor, no restriction on file size
- supports multiple files and windows
- displays compiler detected errors
- alternate editor conforming to the Macintosh UIG

Debugger:
- symbolic debugger with multiple windows
- displays process and data state of an erroneous program
- allows "zooming" into data structures

MacMETH shell:
- compiler and editor remain loaded after activation
- very fast program switching
- hierarchical file system and paths supported
- MacMETH compiles itself: it is written entirely in Modula-2.

System Requirements

The MacMETH system requires at least a 512 kByte Macintosh and runs on all Macintosh models up to the Quadra. For serious program development, 1 MB of RAM and an external drive or preferably a hard disk is recommended.

MacMETH fully supports the hierarchical file system of the Macintosh by allowing the user to specify alternate search paths in his own configuration file (i.e. "User.Profile").

MacMETH also runs under Apple's newest System 7.0.1 in all modes except for the the Virtual Memory mode which is not fully supported, i.e. in the case of exception debugging. Hence, we recommend to turn the Virtual Memory switch off.

1.3. Getting Started

The MacMETH System can be obtained via anonymous internet file transfer ftp (at no charge) from the host "*neptune.inf.ethz.ch*" (Internet address 129.132.101.33) in ftp directory "*macmeth*" or as part of the modeling and simulation package RAMSES from the host "*baikal.ethz.ch*" (Internet address 129.132.80.130) in ftp directory "/pub/mac/*RAMSES*". The first thing to do is to make a copy of the release via ftp. Then double click the file and specify the destination of the MacMETH package on your hard disk.

The MacMETH Release 3.2 consists of the following files and folders:

Two **basic files** containing the shell and the configuration file:

MacMETH 3.2	main application shell (calls the editor, the compiler, the loader etc.)
User.Profile	configuration file

The folder "**M2Tools**" contains all system programs available with the MacMETH shell:

Compile	standard compiler (pre-linked) with MC68000 codegenerator
Compile20	standard compiler (pre-linked) with MC68040 codegenerator
Debug	debugger (pre-linked)
Decode	disassembles object files (pre-linked)
Edit	standard editor "Sara" (pre-linked)
Edit2	alternate editor (pre-linked)
ErrList.DOK	list of compiler error messages
Link	application maker/linker
Print	print utility
ReadProfile	process (the previously edited) configuration file
Unload	unload all currently loaded modules
UnmarkErrs	unmark compiler errors in source code (pre-linked)

The folder "**M2BaseLib**" contains the object and symbol files of modules, which are fundamental and essential for the proper running of the system programs. Two files with extension .OBM (object file) and .SBM (symbol file) exist for each module mentioned below, except for the module System:

Conversions	string conversions from and to numbers
CursorMouse	mouse handling and cursor tracking
EventBase	simple event scheduler
FileSystem	basic file input/output procedures
FileUtil	additional file routines
InOut	simple handling of formatted input/output
MacSystem	reference file for the Quickdraw objects
Menu	command selection from a menu
Printer	basic printer output procedures
Storage	memory allocation/deallocation
System	MacMETH runtime support including the loader
Terminal	screen oriented basic input/output procedures
TerminalIn	screen oriented basic input procedures
TerminalOut	screen oriented basic output procedures
TextWindows	text window handling
Windows	window handling

This folder also contains the file "MacSystem.RFM", the reference file for all Quickdraw objects (see "Inside MacIntosh" for further details). The object file "System.OBM" is not in the folder "M2BaseLib" as it forms part of the MacMETH shell (resource of type "CODE").

The folder "**M2Lib**" contains object and symbol files of modules, which are considered as useful for Modula-2 Programming, e.g. when learning Modula-2:

Dialog	creation and handling of dialogs
EV24	asynchronous interface to Macintosh modem port
GraphicWindows	graphic window handling
LineDrawing	see 'Programming in Modula-2' ([1], [4]), Chapter 28
LongMathLib	mathematical functions for LONGREAL
MathLib	mathematical functions for REAL
String	string handling

The folder "**Examples**" contains miscellaneous demo (source) programs:

Buggy.MOD	Edit/Compile/Execute cycle demo program
Example.MOD	debugger demo program
Compile.CMDs	sample command file

Additional files as a (default) recommendation for the alternate editor:

MEdit 1.79	application MEdit as a default for the supervisor program "Edit2"
Macros	Macro file for the application MEdit

1.4. The Configuration File "User.Profile"

The file "User.Profile" gives you the possibility to configure the MacMETH system according to your preferences or according to your hardware. This file must be at the same level (i.e. in the same folder) as the application MacMETH. The file contains the following different sections:

1. The "PATH" section
2. The "Traps" section
3. The "Menu" or "FullMenu" section
4. The "System" section
5. The "Printer" section
6. The "SANE" section
7. The "Alias" section

Note that MacMETH is case-sensitive, so please make sure that all titles and subtitles in your "User.Profile" are written *exactly as listed in this manual* (e.g. "Path" is not the same as "PATH")!

The "PATH" section

Macintosh file names have the following syntactical structure:

```
FileName    = Path Name.
Path        = [VolName] {":" SubDirName | ":"} ":".
```

VolName is the volume name (each Macintosh diskette has a name). SubDirName is the name of a subdirectory or folder. Name is the local name of the file.

Examples of correct Macintosh file names:

MacMETH:File1	disk name followed by local file name
Untitled:Library:TextWindows.DEF	disk name, folder name, local file name
:Library:RealInOut.MOD	relative path starting at current folder where the MacMETH shell resides
::User2:Temp	relative path starting one folder

above the current folder

Normally, you only want to deal with local file names and forget about paths. In the "PATH" section you give an ordered list of paths that will be prefixed to the file name one after another until the file is found (if it exists in one of the given locations or in the start up directory of the MacMETH shell which is searched first). The output files of the MacMETH applications (i.e. Edit, Compile, Link, etc.) are written to the same location where the input file was found.

Here is a sample "PATH" section (recommended default for MacMETH 3.2):

```
"PATH"
        :Examples:, :M2Tools 3.2:, :M2Lib 3.2:, :M2BaseLib 3.2:
```

All the 4 entries in the last example are relative folder names (it is advisable to use relative paths, as this keeps your file names small and allows you to move or copy your Modula-2 development directory freely without having to adapt any path specifications. Some applications limit the maximum file name length to 64 characters!). There is no limit on the path list length in the "User.Profile".

The "Traps" section

The MacMETH system "catches" all traps which occur during the execution of a MacMETH program and displays the standard MacMETH error box (see Fig. 5.2), allowing you to choose between entering the debugger, continuing or aborting program execution. In the "Traps" section, you may specify, which traps should be caught, and which not. There are six entries in this section: 'All', 'Arithmetic', 'FPU', 'F-Line', 'System' and 'Break'. Any of these traps may be either enabled (on) or disabled (off). If the whole section is omitted in the "User.Profile", traps are installed according to a machine specific default strategy, i.e. they are mostly caught.

'All'
Usually, 'All' is set to "on", meaning that **all** traps are caught according to the subsequent specifications by MacMETH. If you change the setting to "off", no traps are caught, and all other entries in this section are skipped, i.e. all other entries are valid only if 'All' is enabled!
Trap numbers: 2, 3, 4, 5, 6, 7, 11, 28, 29, 30, 31, 32, 33, 48, 49, 50, 51, 52, 53, 54.

'Arithmetic'
If a serious error occurs during the evaluation of an integer expression, an arithmetic trap of the Integer Unit (CPU) is triggered. If 'Arithmetic' is enabled, MacMETH will catch the traps and start the appropriate actions; if it's disabled, the trap request is passed to the Macintosh operating system.
Trap numbers: 5, 6, 7.

'FPU'
When a Floating-Point Hardware Unit (FPU) is available on platforms equiped by the MC68881, the MC68882 or the MC68040 *and Compile20 compiled code shall be executed*, additional Floating-Point Exception Vectors must be preset in the system. If a serious error occurs in a floating-point operation, a floating-point exception is signaled by the hardware. If 'FPU' is enabled, MacMETH will catch the floating-point hardware exception and allow to start the debugger; if it's disabled, the trap request is passed to the Macintosh operating system.
Trap numbers: 48, 49, 50, 51, 52, 53, 54 for the MC68881/MC68882.
For the MC68040, Trap numbers 48, 51 - 54 are let unchanged and MacMETH installs the so-called secondary User Exception Vectors.

'F-Line'
It's the same here, too: If 'F-Line' is set to "on", MacMETH will catch this trap, if it's disabled, the trap request is passed to the Macintosh operating system.
Trap number: 11.

For the MC68040, Trap number 11 is never changed and a secondary vector is installed.

'System'

The keyword 'System' is used here for the Access Fault, Address Error and Illegal Instruction traps, which are highly system and hardware dependent. If 'System' is set to "on", MacMETH will catch these traps, if it is disabled, the trap requests are passed to the Macintosh operating system. Note that the Access Fault Trap (#2) is installed only if the System (version) < 7.0 or if the MMU < MC68851.
Trap numbers: 2,3,4.

'Break'

If 'Break' is enabled, the MacMETH system catches all traps triggered from the (programmer's) interrupt/reset button or from break points set within the debugger. If 'Break' is disabled, MacMETH won't do anything, it'll just hand the trap over to the Macintosh system.
Trap numbers: 28, 29, 30, 31, 32, 33.

Example (recommended for execution of code compiled with Compile):

```
"Traps"
        'All'           on
        'Arithmetic'    on
        'FPU'           off
        'F-Line'        on
        'System'        on
        'Break'         on
```

Defining the File menu with the "Menu" or "FullMenu" section

Using the "**Menu**" section, MacMETH allows you to include MacMETH-compiled Modula-2 programs as **additional menu entries** in the File menu. While Edit/Compile/Execute and Quit constitute two permanent blocks in the File menu, the additional entries are located just between the two fixed parts and are separated by the dotted lines (see Fig. 1a). The character "|" acts as a delimiter between successive entries in the "Menu" string. If such an additional entry is selected,the program is loaded (including all imported modules) and executed on top of the MacMETH shell.

For example, the following "Menu" command will produce the File menu below:

```
"Menu"
        'Link/L|Decode|ReadProfile/R'
```

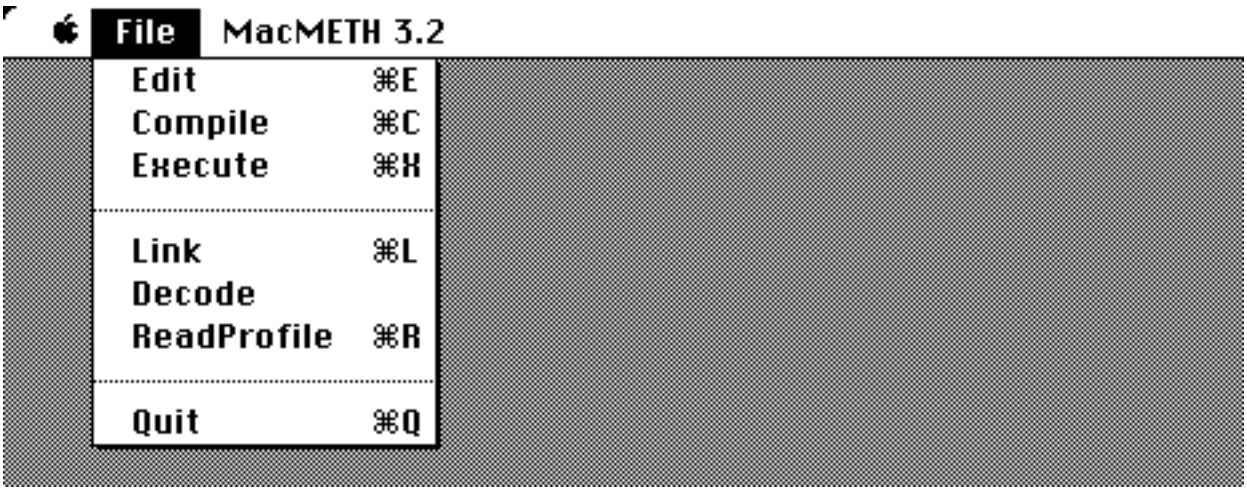

Fig. 1a: Defining **parts** of the File menu with the "**Menu**" command

In contrast to the "Menu" command, the "**FullMenu**" section allows you to configure the **entire File menu** according to your choice (see Fig. 1b). Here again, the character "|" acts as a delimiter between successive entries in the string. However, the default Edit/Compile/Execute block is not inserted and every menu entry must be specified *explicitly* within the string. These entries now start from the very beginning of the File menu. Note that the Quit command is always appended automatically as the last entry.

Example: the following "FullMenu" command will produce the File menu below:

```
"FullMenu"
    'Edit2/E|Edit|Compile/C|Compile20/2|Execute/X|(-|ReadProfile/R|Unload/U|Link/L|Debug'
```

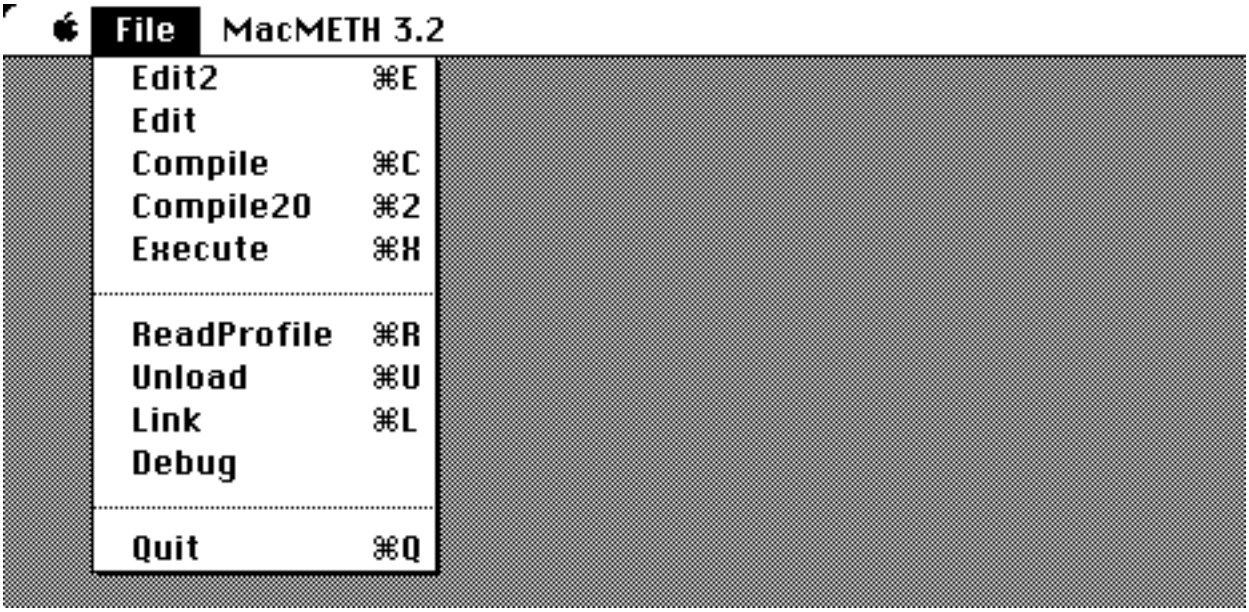

Fig. 1b: Defining the **entire** File menu with the "**FullMenu**" command

If the whole "Menu" section is omitted, the menu entries Edit/Compile/Execute and Quit are asserted in any case.

The "System" section

In this section you may determine whether or not the different compilers and editors of the MacMETH system shall be kept in memory, once they are loaded. There are two possible entries in this section ('Compiler' and 'Editor') and also two values you may set these entries to (keep and nokeep). 'Compiler' stands for the set of possible compilers and 'Editor' for the set of possible editors. Actually any module starting with module name "Compile..." or "Edit..." is affected by this mechanism. The meanings of these values are explained below:

keep
Keep the specified part of MacMETH in memory, once it's been loaded. This way a restart of that part is much quicker; it does not have to be re-read from disk again.

nokeep
As soon as the specified part isn't used anymore, free the memory it occupied.

Example:

```
"System"
    'Compiler'      keep
    'Editor'        nokeep
```

The "Printer" section

In this section you may set certain printer parameters used by the utility Print, such as page length etc. If 'Header' is specified in the "User.Profile", the text, date and page number are printed on top of each page.

Example:

```
"Printer"
    'PageLen'       60
    'LeftMargin'     8
    'Header'        ETH Zuerich
```

The "SANE" section

The MC68000 chip doesn't supply instructions for REAL arithmetic; so the basic MacMETH Compiler (Compile) has to emulate the missing operations in software. For single-precision REAL arithmetic there exist two different approaches to solve this problem. On machines without a floating-point processor the compiler will setup calls to a simple, but fast floating-point emulator developed at ETH (called Fink arithmetic for the 32-Bit REAL's). On the other hand, an **interface to Apple's SANE** package is available. Therefore the "SANE" section in the User.Profile was introduced. The essential switch is specified with the 'alwaysSANE' keyword: turning 'alwaysSANE' off will select the Fink arithmetic. Turning **'alwaysSANE' on**, however, will advise the system to use the (slower, but preciser) SANE package.

Additional keywords were introduced to control the behaviour of a program, should SANE detect a floating-point exception. When, for example, 'overflowHalt' is on and an overflow exception occurs, MacMETH will gain control over the exception and the user may start the debugger or abort the program. On the other hand, when 'overflowHalt' is turned off, the floating-point operation is not interrupted and the calculation proceeds with an Infinity as a so-called halt-disabled result. Note that only SANE's control of exceptions and halt-disabled results are in full conformance with the Floating-Point Processors (MC68881, MC68882, MC68040) of Motorola. For operations on the 64-Bit type LONGREAL and for all functions of the default version of Module MathLib, the interface to SANE is choosen anyway. The following example is self explanatory and shows all the possible switches.

Example (recommended defaults):

```
"SANE"
    'alwaysSANE'        on
    'invalidHalt'       on
    'underflowHalt'     off
    'overflowHalt'      on
    'divByZeroHalt'     on
    'inexactHalt'       off
```

The "Alias" section

When the alternate editor Edit2 is used, the "Alias" section in the User.Profile is mandatory. This section specifies the name of the application, which constitutes the alternate editor, as a parameter to the supervisor program Edit2.

Example (recommended default):

```
"Alias"
    'Edit2'         is      '  MEdit 1.79'
```

1.5. A Sample Edit/Compile/Run/Debug Session

The numbers in the text correspond to the numbers in the graphic (Fig1.1).

1. Start the system by double-clicking the MacMETH application. Select Compile from the File menu.
2. The compiler requests a file name. Enter "Buggy.MOD" and press <RETURN>. The compiler detects errors in the program.
3. Leave the compiler by typing <RETURN> and select "Edit" from the File menu. The editor suggests the last compiled file "Buggy.MOD" for editing.
4. Press <RETURN> to accept the default file name. The editor points with the caret to the erroneous position and displays the error message in the lower window. Correct the error,leave the editor *by clicking into the window's close box* and enter the compiler again. The compiler suggests "Buggy.MOD" for compiling.
5. Press <RETURN> to accept the default file name. The compiler generates 110 bytes of native MC68000 code in one pass. Leave the compiler and select "Execute" from the File menu.
6. Select "Buggy.OBM" from the file box using the mouse.
7. A runtime error occurs. Select "Debug" using the mouse.
8. The debugger displays five windows with source text (error postion marked), procedure call chain, module list and local/global variables with name, type and value.

This session takes about 60 seconds.

1.6. Keyboard and Mouse, Special Keys

The following notation is used throughout this documentation.

ML	mouse button
MM	command key
MR	option key
EOL	return key
ESC	enter key
TAB	tabulator key
SP	space bar
DEL	"<--" key

For non-ASCII-keyboard character sets see also page 14.

1.7. References

[1] Niklaus Wirth: "Programming in Modula-2", **Third corrected Edition**,
Springer-Verlag, Berlin Heidelberg New York Tokyo, 1985.

[2] "Inside Macintosh"
Apple Computer Inc., 20525 Mariani Ave, Cupertino CA 95014.

[3] Motorola, MC68000 16/32-Bit Microprocessor,
Programmer's Reference Manual, Fourth Edition, 1984,
Prentice-Hall, Inc., Englewood Cliffs.

[4] Niklaus Wirth: "Programming in Modula-2", **Fourth Edition**,
Springer-Verlag, Berlin Heidelberg New York Tokyo, **1988**.

[5] Anonymous, 1991. ResEdit Reference: For ResEdit 2.1. Reading, Mass.: Addison-Wesley Publishing, 153pp. "An Apple development document" ISBN 0-201-57091-2.

[6] Alley, P., & Strange, C., 1991. ResEdit complete. Reading, Mass.: Addison-Wesley, 3rd printing, 546pp. (includes program diskette).

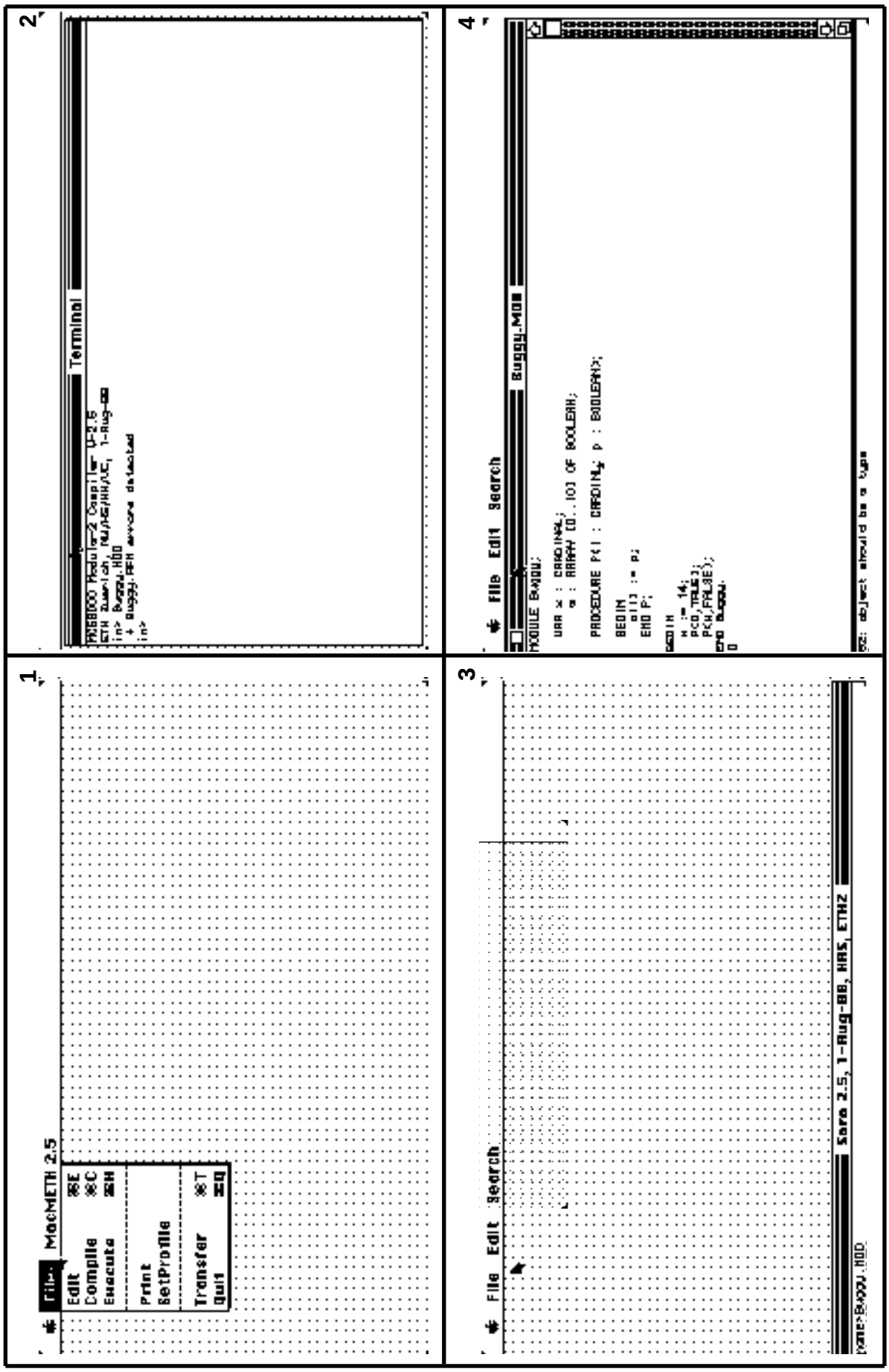

Fig. 1.1: A Sample Edit/Compile/Run/Debug Session

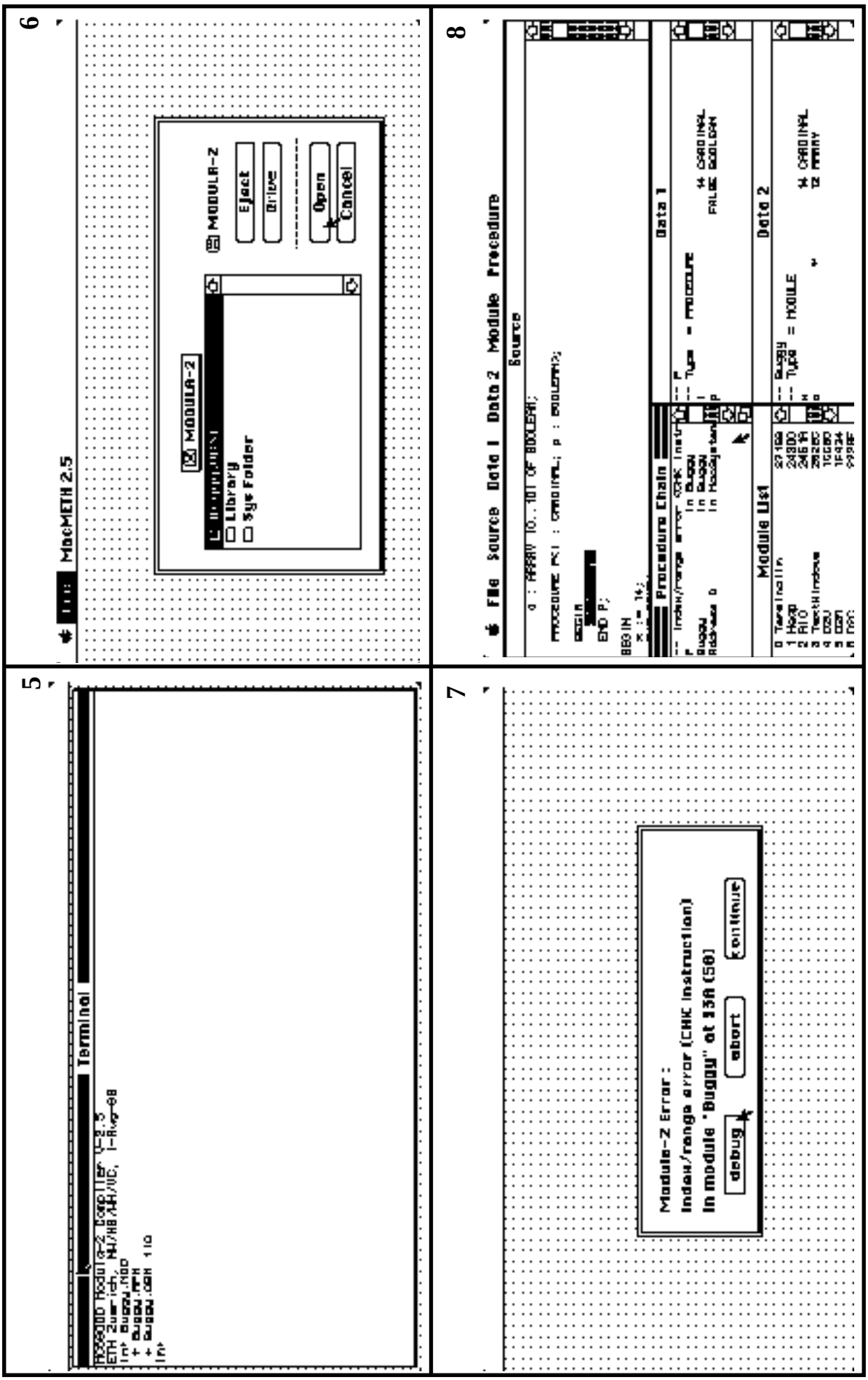

Fig. 1.1B: A Sample Edit/Compile/Run/Debug Session

2. The Program Environment

2.1. Introduction

The MacMETH system mainly consists of an editor, a compiler, and a debugger. On the installed release they are stored in form of pre-linked Modula-2 programs and are invoked by the help of a shell (the Macintosh application MacMETH). The shell contains many often needed library modules as well as a loader for Modula-2 object files. The following picture gives a model of the MacMETH system on the Macintosh.

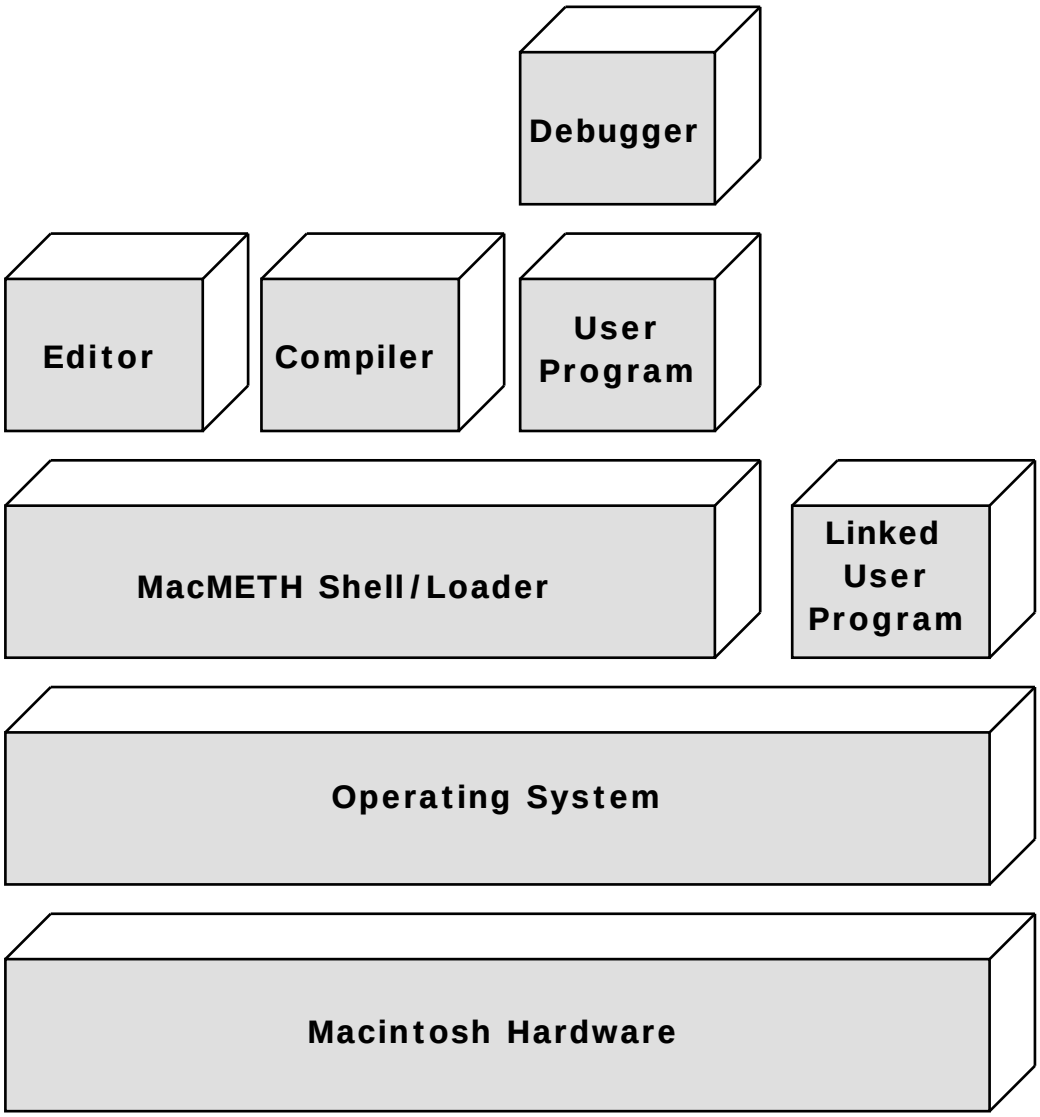

Fig. 2.1: MacMETH Layers

Each layer builds upon the resources of the next lower layer. The debugger is only called in case of a runtime error. MacMETH allows programs to dynamically call other programs (no linking necessary).

2.2. Starting MacMETH

You enter the shell by double-clicking the Macintosh application MacMETH at the Finder level. Then, the following screen image is presented:

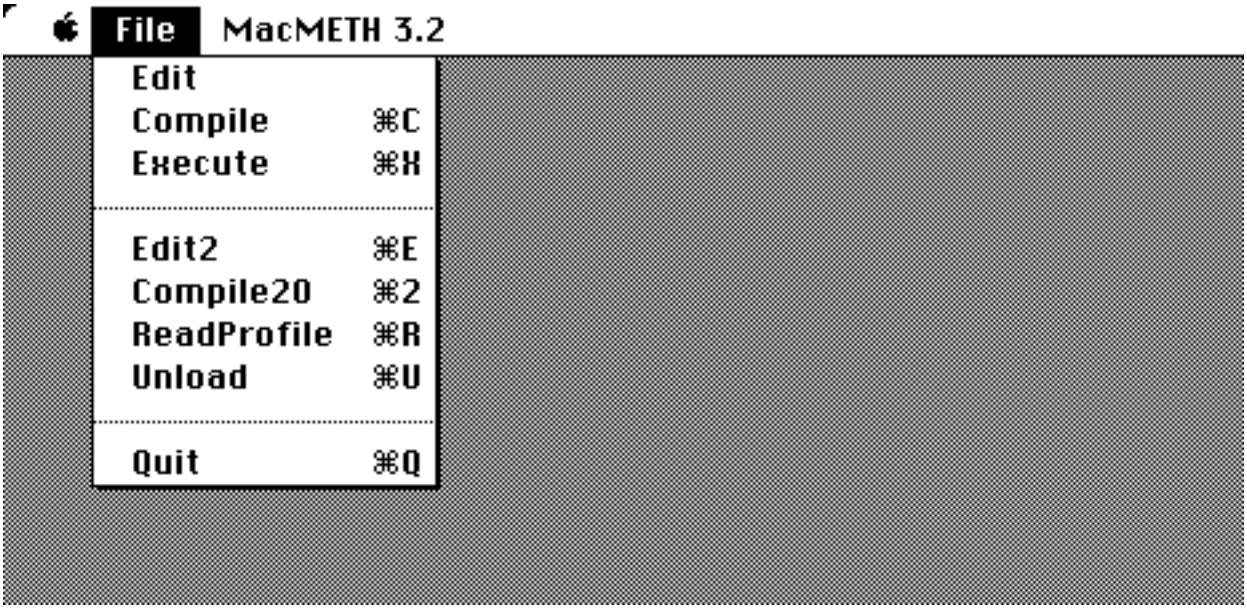

Fig. 2.2: Starting MacMETH

You have several options in the File menu: "Edit", "Compile", "Execute", "Edit2", "Compile20", "ReadProfile" et cetera, and finally "Quit " (the debugger is only called in case of a runtime error).

2.3. Several Ways to Work

There are several possible ways to work with the Modula-2 system depending on your preferences. Let's group the MacMETH users into three categories and see how they work with the system:

1. The Modula-2 program developer who likes to work on the fast MacMETH shell (most users will prefer that)
2. The Modula-2 program developer who wants to work at the Finder level (having his own favorite editor, perhaps)
3. The user of an application developed with MacMETH.

Group 3

Every Modula-2 program can be converted into a standalone Macintosh application, so the user doesn't have to know anything about MacMETH.

Group 2

Every Modula-2 program can be transformed into a standalone Macintosh application. The editor, compiler, and debugger are Modula-2 programs. So, there can be no objection to convert them (with the help of the linker, see chapter 6) into standalone applications and work at the Finder level.

Group 1

This is the fastest way to work with the Modula-2 system. You may enter the editor or the compiler through the File menu, and once they are loaded from the disk, they remain loaded as long as the shell is active. Please note, however, that the files you work with are always saved on disk.

We assume that you have successfully compiled the modules "Buggy" and "Example" in the folder "Examples". Now, if you select "Execute" from the File menu, you will get the following screens:

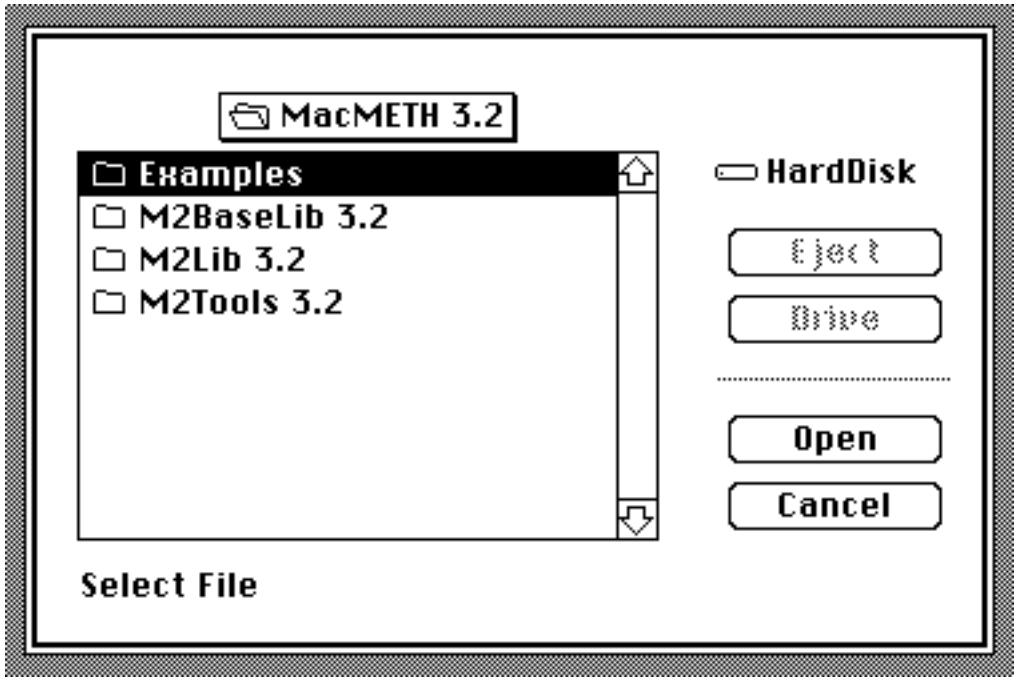

Fig. 2.3a: Selecting programs for execution

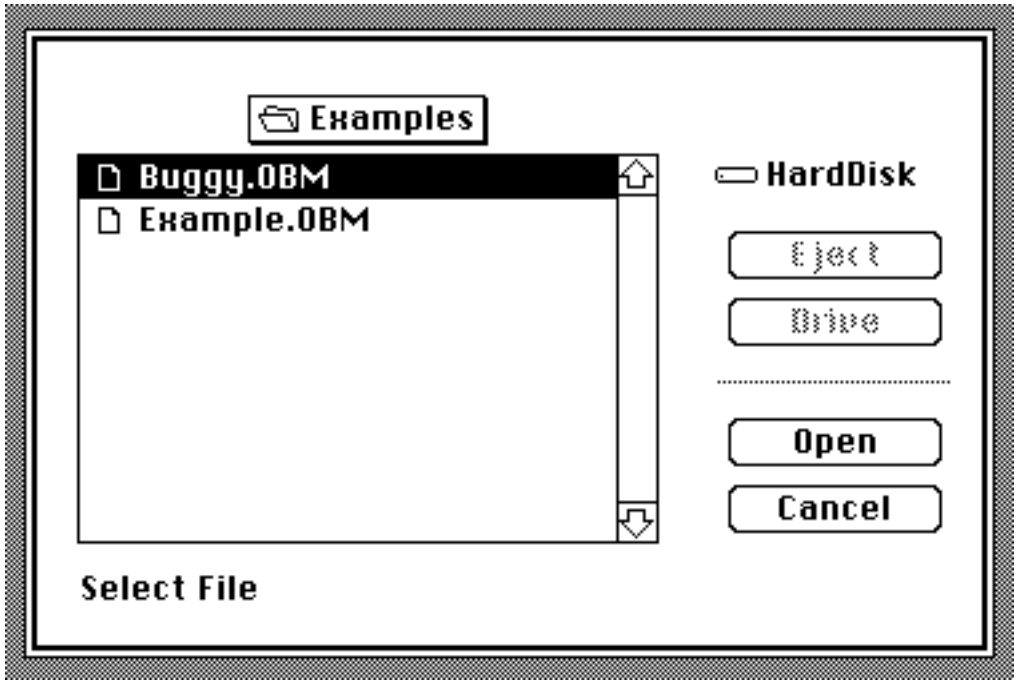

Fig. 2.3b: Executing programs

You can now select the program you want to execute and start it (all missing modules are loaded automatically). In particular, you can also start *every system program* (every tool in the folder "M2Tools") with this command: the editor, the various compilers, the linker, the decoder, the alternate editor et cetera. The difference, however, is that programs activated in this way do *not* remain loaded after termination.

3. The Editor

3.1. Introduction

This chapter describes the use of the standard MacMETH Editor "Sara". It is a full-screen editor, specially designed to create and change programs (it uses a special interface to the compiler to display possible compilation errors). This editor imposes no restriction on file size, as it uses an incremental method (piece-list) for editing.

3.2. Starting the "Sara" Editor

You start the "Sara" Editor by selecting Edit from the MacMETH-menu:

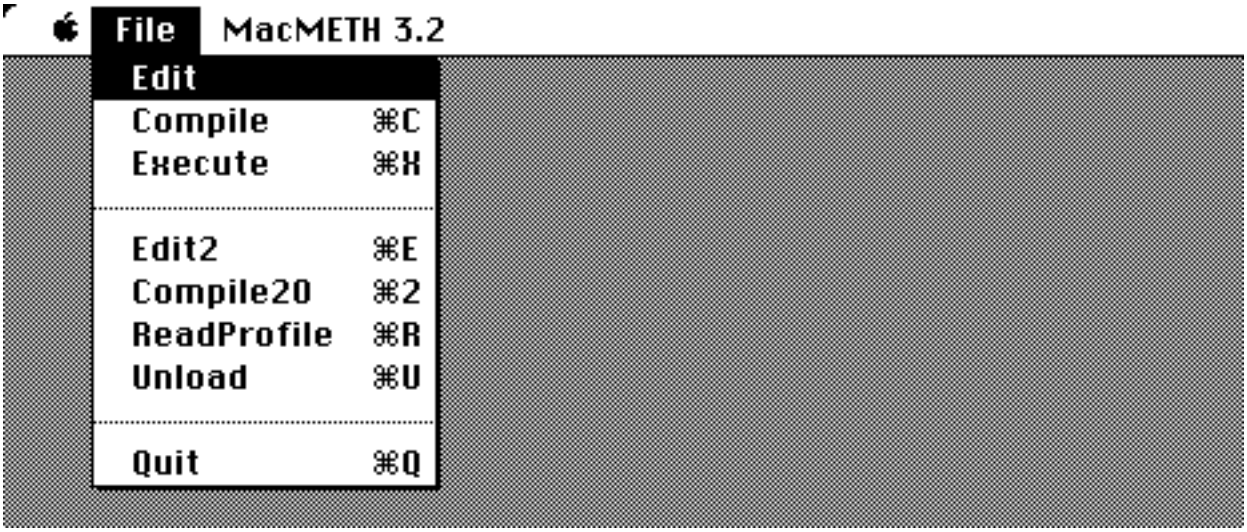

Fig. 3.1: Starting the "Sara" Editor

Now the Macintosh presents the following screen image:

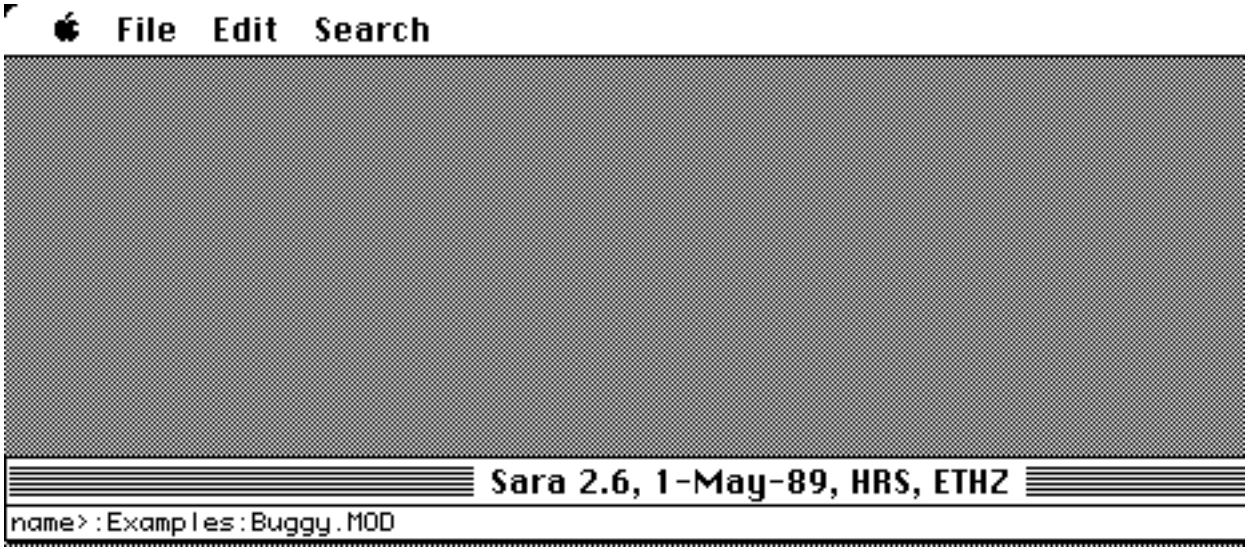

Fig. 3.2: Initial Display of the "Sara" Editor

You see a small window on the bottom of the screen, possibly giving you a default file name for editing (the last compiled file is the default). Press <return>, <space> or the mouse button to accept

the name, or enter an alternate name (to create a new text enter the empty name). If the file name ends with ".", the extension "MOD" is appended.

The editor now creates another window displaying the first page of your program. If your program contains errors (detected by the compiler in an earlier compilation) the caret (the text insertion point) is automatically placed after the first error. The dialog window displays the type of error.

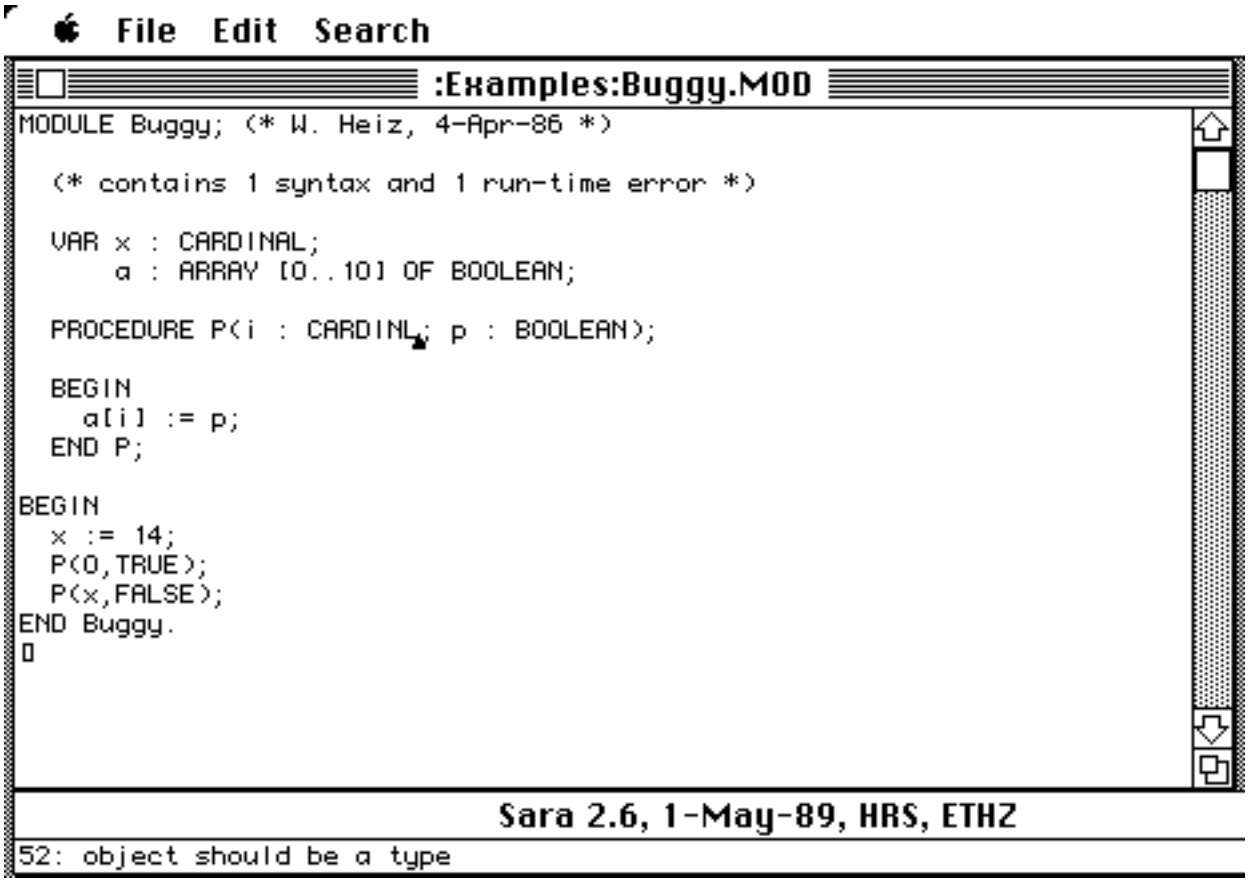

Fig. 3.3: Screen Image of the "Sara" Editor

3.3. Text Entry

The text insertion point (caret) is represented by a small triangle which can be positioned by ML (press and release at the same place). The entered text is inserted at the caret and does not overwrite a previous text. You can delete the last character by typing DEL. On the German keyboard, it is somewhat difficult to locate characters needed in Modula-2. The following table helps you finding them:

[	<option> 5
]	<option> 6
|	<option> 7
{	<option> 8
}	<option> 9
^	^^ or <option> <shift> 6 <option> <shift> 6
#	<option> <shift> 3

3.4. Text Selection

A text portion can be selected by pressing the ML button and moving the mouse over the text (dragging). If at the same time another button is active, the following commands are executed directly: Copy (MM), Delete (MR), or Move (MM and MR). The selection is cancelled by pressing ESC. An existing text selection can be extended/shortened by pointing to the last selected character and dragging.

3.5. Scrolling

The "Sara" Editor uses the standard Macintosh scroll mechanism by the help of the scroll bar on the right.

3.6. Menu Commands

The "Sara" Editor commands can be classified into 3 groups: File, Edit, and Search.

- File open, split and print documents, quit.
- Edit copy, move, delete, adjust text, autoindentation on/off.
- Search find error, find string, replace.

The commands are explained in more detail below.

File

Fig. 3.4: File Commands

Open

Open a document. The "Sara" Editor allows you to open as many as 14 files at a time. The name is read from the keyboard (the default name is the name of the last compiled file or a valid text selection in the file). An empty name creates a new document.

name>

EOL,ML	Terminate the input. A trailing "." is expanded to ".MOD".
DEL	Delete the last character.
ESC, MM, MR	Abort the command.
TAB	*The standard Macintosh file selector box is displayed. Now you can select the file using the mouse*. To confirm your selection, click the "Open" button.

Split

Split window into subwindows. Each subwindow can be scrolled independently. Selection beyond subwindow bounderies is allowed (all text between is selected). This allows you to select text portions greater than the window size.

define separation line

ML, MM, MR	Used to point to the location where separation should take place (dragging allowed).
ESC	Abort the command.

Print

Print the file denoted by the caret.

printing ...

any key	Pause the printing process.
ESC	Abort the command.

Quit

Leave the " Sara" Editor without updating the document.

exit without save?

"y", "Y", ML	Leave the "Sara" Editor.
any other key	Abort the command.

Close

Documents and subwindows are closed by activating their close boxes (upper left square in title bar). The name is read from the keyboard (the default name is the name given in the "Open" command or a valid text selection). After closing the last file the "Quit" command is called automatically.

name>

EOL, ML	Terminate the input. A trailing "." is expanded to ".MOD". The document is stored on disk and the previous version of the document gets the extension ".BAK".
DEL	Delete the last character.
TAB	Works the same way as in "Open" (see above).
ESC, MM, MR	Release the document without updating.

release?

"y", "Y", ML	Close the document without updating.
any other key	Abort the command.

Edit

Fig. 3.5: Edit Commands

Copy
Copy a piece of text. If a text sequence is selected, it is copied to the internal buffer. Then the buffer contents are inserted at the caret.

Move
Move a piece of text. This command corresponds to the command sequence Delete, Copy.

Delete
Delete a piece of text. If a text sequence is selected, it is copied to the internal buffer and deleted.

Adjust
Shift a piece of text horizontally. The selected lines are shifted to the left or to the right (useful to indent a whole procedure).

adjust(-9..9)>

"0", "1", ..., "9"	Shift 0, 1, ..., 9 position(s) to the right.
"-1", ..., "-9"	Shift 1, ..., 9 position(s) to the left.
ESC, MM	Abort the command.
any other key	Don't shift.

AutoInd
Toggle automatic indentation mode on/off. Default is on.

Search

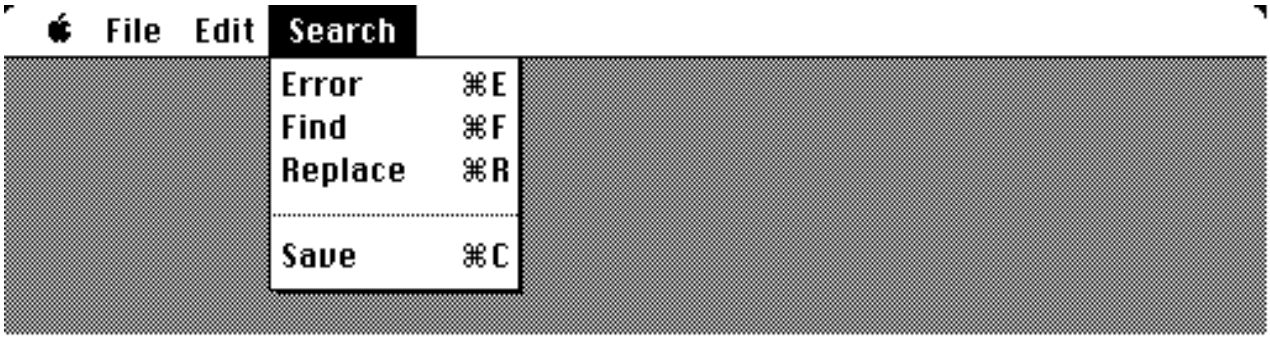

Fig. 3.6: Search Commands

Error
Search for the next (compiler detected) error starting at the caret position.

Find
Find a text portion starting at the caret position.

find>

text selected	Find the selected text.	
keyboard	Enter text by keyboard.	
	EOL	Terminate input.
	DEL	Delete last entered character.
	ESC, MR	Abort the command.
ML	Find next occurrence of a previously searched text.	
ESC, MR	Abort the command.	

Replace

Replace a text portion (to be searched for) by the internal buffer contents, starting at the caret position. The text to be found is specified as in the Find command. The new text has to be moved to the internal buffer (e.g. with Save). If the text is found the "Sara" Editor asks whether it should be replaced.

replace by buffer content?>

"y", "Y", ML	Replace and search again.
"n", "N", MR	Do not replace but search again.
"0", "1", ..., "9"	Replace 0, 1, ..., 9 times.
EOL	Replace all occurrences starting at the caret. Typing any key stops the replacing.
ESC	Abort the command.
SPC	Replace.
any other key	Do not replace.

Save

Copy a text portion to the internal buffer. If no text is selected the internal buffer is cleared (to allow the replace command to delete a found string).

3.7. Starting the Alternate Editor via Edit2

The program "Edit2" opens the door to an editor which is in conformance with the Macintosh User Interface Guidelines. MacMETH's (default) recommendation for such an alternate editor is the application "MEdit", which stands for Macro Editor (for details see Appendix E).

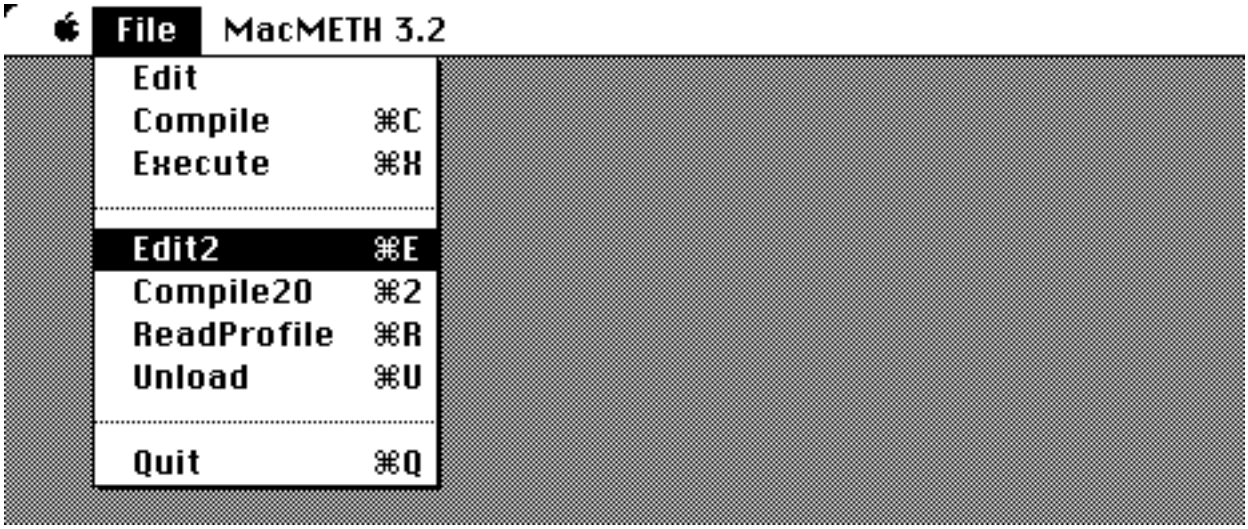

Fig. 3.7: Starting the Alternate Editor

The purpose of the system program Edit2 (see folder "M2Tools") is to facilitate the starting of the alternate editor from the MacMETH shell. Edit2 plays the role of a bridge between the MacMETH environment and the alternate editor. First, the program Edit2 inspects the User.Profile for a valid "Alias" section and saves the alias name. Second, if your program contains errors, Edit2 inserts error marks in the source file specified by the most recent compilation. Finally, Edit2 **launches the application** with the corresponding alias name (for example 'MEdit 1.79'). Now the source file of your program is automatically opened and presented in the front window. When the alternate editor application terminates, a **transfer back** to the original application takes place and the MacMETH shell is resumed. Under Multifinder or System 7 it is also possible to reenter the MacMETH shell by a single command from within MEdit ("Launch M2 shell", **1**).

The following 2 preconditions must be asserted for Edit2:

1. the presence of an "Alias" section in the configuration file is mandatory when the supervisor program Edit2 is called,

2. the application to launch (e.g. "MEdit 1.79") and its associated files (e.g. the file "Macros") have to reside in the same folder as the MacMETH shell.

WARNING:

MacMETH and MEdit are different applications: they do not really share the files. Therefore, the processed files **must** always be closed before quitting the application MEdit; otherwise MacMETH would not be able to retrieve them.

4. The Compiler

4.1. Introduction

This chapter describes the use of the standard Modula-2 Compiler. For the language definition refer to [1]. We emphasize that the standard MacMETH Compiler is based on the language report presented in the **Third corrected Edition** of 'Programming in Modula-2'. This compiler is a single-pass Modula-2 compiler generating native MC68000 code (or native MC68020 code, see Appendix A). Linking is not necessary as the code is completely relocatable and need not be fixed.

Glossary

compilation unit
: Unit accepted by compiler for compilation, i.e. definition module or program module.

definition module
: Part of a separate module specifying the exported objects.

program module
: Implementation part of a separate module (called implementation module) or main module.

source file
: Input file for the compiler, i.e. a compilation unit. Default extension is "MOD".

symbol file
: Compiler output file with symbol table information. This information is generated during compilation of a definition module. Assigned extension is "SBM".

reference file
: Compiler output file with debugger information, generated during compilation of a program module. Assigned extension is "RFM".

object file
: Compiler output file with the generated MC68000 code (or MC68020 code) in MacMETH loader format. Assigned extension is "OBM".

4.2. Starting the Compiler

You start the compiler by selecting Compile (or Compile20) from MacMETH's File menu:

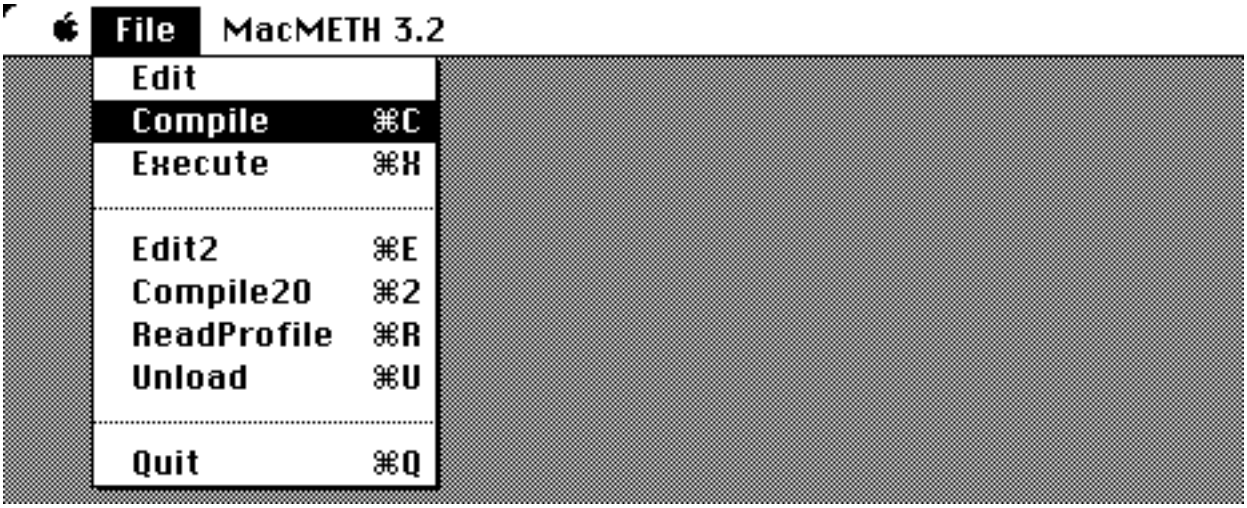

Fig. 4.1: Starting the Compiler

Now the Macintosh presents the following screen image:

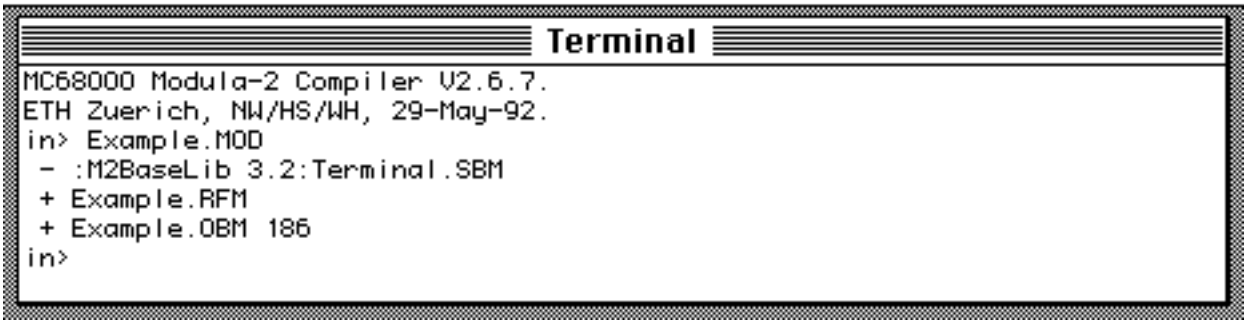

Fig. 4.2: Screen Image of the standard Compiler

You see the window "Terminal" on screen, possibly giving you a default file name for compilation (the last compiled file is the default one).

4.3. Compilation

Compilation of a Program Module

Press EOL to accept the name or enter an alternate name. If the file name ends with ".", the extension "MOD" is appended. You may also type **TAB**, which makes *the standard Macintosh file selector box appear on the screen*. Now you can select the file by using the mouse (click the "open" button or type EOL or ESC to confirm your selection).

The compiler displays all files it reads (i.e. imports) with the tag "-" and all files it produces with the tag "+". After successful compilation of a program module the number of generated code bytes is displayed. In case of an error, the message "errors detected" appears instead. After compilation, the compiler again requests a file name for the next compilation. Terminate by typing ESC, EOL or pressing the mouse button.

Compilation of a Definition Module

For definition modules the file name extension "DEF" is recommended. The definition part of a module must be compiled prior to its implementation part. Upon compilation of a definition module, a symbol file containing symbol table information is generated. This information is needed by the compiler in two cases:

1. At compilation of the implementation part of the module.
2. At compilation of another unit, importing objects from this separate module.

If a required symbol file is missing, the compilation process is stopped.

Compilation of a sequence of Modules

By typing the command key, the shift key and the numeric key '0' (or the letter 'o') simultaneously (**Shift 0** or **Shift o**), the standard Macintosh file selector box will appear on the screen. Now you can select a file. This file is interpreted by the compiler as a *command file*: each line on this file must contain the source file name (possibly followed by option characters) of a module (Definition or Program Module) to be compiled. For a sample command file see the file "Compile.CMDs" in the folder "Examples".

Compiler Output Files

Several files are generated by the compiler. Their file names are taken as **the compilation unit's module name** with the appropriate file name extension. Note that the reference file is used by the symbolic debugger.

SBM	symbol file	err.LST	error file for user (text file)
RFM	reference file	err.DAT	error file for editor (data file)
OBM	object file		

The Module Key

With each compilation unit the compiler generates a so called module key. This key is unique and is needed to distinguish different compiled versions of the same module. The module key is written on the symbol file and on the object file.

For an implementation module the key of the associated definition module is adopted. The module keys of imported modules are also recorded on the generated symbol files and the object files. Any mismatch of module keys belonging to the same module will cause an error message at compilation or loading time.

W A R N I N G

Recompilation of a definition module will produce a new symbol file with a new module key. In this case the implementation module and all units importing this module must be recompiled as well.

Recompilation of an implementation module does not affect the module key.

Compilation Options

The compiler optionally generates various redundancy checks. They can be enabled or disabled for each compilation by appending option characters to the source file name. The occurrence of an option character signals the inverse of its default value.

r	array index, subrange and pointer checks	default = on
v	integer arithmetic overflow check	default = off

Examples:

Name.MOD/v	all checks on
Name.MOD/vr	integer arithmetic overflow checks on.

You also may specify any of these two compiler options in the source of a Modula-2 program. This is done by inserting the following comments in the source file:

(*$R+*)	enables array index, subrange and pointer checks
(*$R- *)	disables array index, subrange and pointer checks
(*$R=*)	resets the option to the state before the last change

For the integer overflow checks it's just the same, you have to include (*$V+*), (*$V-*) or (*$V=*). Note that there must not be any spaces between "(*" and the $-sign!

4.4. Program Execution

Programs are normally executed on top of the resident user environment MacMETH (although you can transform any Modula-2 program into a stand-alone Macintosh application). The command interpreter accepts a program name and causes the loader to load the module on the corresponding object file into memory and to start its execution.

If a program consists of several separate modules, no explicit linking is necessary. The object files generated by the compiler are ready to be loaded. Besides the program module, which constitutes the main program for execution, all modules which are directly or indirectly imported are loaded. The loader establishes the links between the modules.

Usually some of the imported modules are part of the already loaded, resident MacMETH system (e.g. module "FileSystem"). In this case the loader sets up the links to these modules, but prohibits their reinitialization. A module cannot be loaded twice.

After termination of the program, all separate modules which have been loaded together with the main module are removed from memory. More details concerning program execution are given in chapter 7.16 (module "System").

MacMETH also supports a program stack. A program may call another program, which will be executed on the top of the calling program. After termination of the called program, control will be returned to the calling program. For more details refer to chapter 7.16 (module "System").

4.5. The Implemented Language

Standard Constants

FALSE TRUE NIL

Standard Types

BOOLEAN — a variable of this type assumes the truth values FALSE or TRUE. These are the only values of this type. The size of BOOLEAN is 1 byte.

CHAR — type CHAR is defined according to the ISO-ASCII standard with ordinal values from 0 to 255. The type is represented within 1 byte. The compiler accepts character constants in the range [0C .. 377C].

BITSET — the type BITSET is defined as SET OF [0 .. 15]. The size of this type is 2 bytes. Note that sets obey the Motorola numbering conventions for bit data: {0} corresponds to the ordinal value 1.

INTEGER — a variable of type INTEGER assumes as values the integers between -32768 and 32767. The type's size is 2. The arithmetic is signed and overflow is checked whenever the corresponding compiler option is on.

CARDINAL — a variable of type CARDINAL assumes as values cardinal numbers between 0 and 65535 (2 bytes). The arithmetic for this type is **unsigned** and overflow is not checked.

REAL — a variable of this type assumes as values the single precision real number represented in 4 bytes. The value range expands from -3.40E38 to +3.40E38.

LONGINT — the range of type LONGINT is from -2147483648D to 2147483647D. The type's size is 4. Constants of type LONGINT must have the suffix letter "D". LONGINT arithmetic is signed and overflow is checked when the corresponding option is on.

LONGCARD — the value range of type LONGCARD is from 0D to 4294967295D. The type's size is 4 bytes. LONGCARD arithmetic is **unsigned** and overflow is not checked.

LONGREAL — values of type LONGREAL are double precision real numbers represented in 8 bytes. The value range expands from -1.79D308 to +1.79D308. Note that constants of type LONGREAL use letter "D" instead of "E"!

PROC — the standard type PROC denotes a parameterless procedure (size is 4 bytes).

Standard Procedures

Standard procedures are predefined (need not to be imported). Some are generic procedures that cannot be explicitly declared, i.e. they apply to classes of operand types or have several possible parameter list forms. A type T is called a **scalar type**, if T is an enumeration type, CHAR, INTEGER, CARDINAL or a subrange type. The types INTEGER, CARDINAL and LONGINT are called the **integer types**. The following table lists all the predefined procedures (v stands for a variable, x and n for expressions, and T for a type):

ABS(x)	absolute value of x. The result type is the same as the argument type.
CAP(x)	if x is a lower case letter, the corresponding capital letter; else the same letter.
CHR(x)	the character with ordinal number x. The type of x must be scalar.
DEC(v)	v := v - 1. The type of v must be scalar. DEC(v,n) implements v := v - n.
EXCL(v,n)	v := v - {n}. v must be a set and n compatible with the base type of the set.
FLOAT(x)	x (a value of an integer type) represented as a REAL value.
FLOATD(x)	x (a value of an integer type) represented as a value of type LONGREAL.
HALT	terminate program execution and display the standard MacMETH halt box.
HIGH(v)	high index bound of array v. For an open array parameter the result type is compatible to INTEGER and CARDINAL.
INC(v)	v := v + 1. The type of v must be scalar. INC(v,n) implements v := v + n.
INCL(v,n)	v := v + {n}. v must be a set and n compatible with the base type of the set.
LONG(x)	x of scalar or REAL type extended to the long value of type LONGINT or LONGREAL, respective.
MAX(T)	the maximum value of type T, which must be either a scalar type or type REAL or LONGREAL.
MIN(T)	the minimum value of type T, which must be either a scalar type or type REAL or LONGREAL.
ODD(x)	returns x MOD 2 ≠ 0 (of type BOOLEAN). x must be an integer type.
ORD(x)	ordinal number of x in the set of values defined by the type T of x, which must be a scalar type. The ordinal value of x must be in the range of positive integers. The result type is compatible to INTEGER and CARDINAL.
SHORT(x)	x of type LONGINT or LONGREAL shortened to the corresponding short form of type INTEGER or REAL. The resulting value must be in the INTEGER or in the REAL range, respectively.
SIZE(x)	the number of bytes used in memory for x, which can be a variable or a type. The result type is compatible to INTEGER and CARDINAL.
TRUNC(x)	Real number x truncated to its integral part (of type INTEGER). The resulting value must fit in the range of the INTEGER's.
TRUNCD(x)	Long real number x truncated to its integral part (of type LONGINT). The resulting value must be in the range of type LONGINT.

The Module SYSTEM

Explicitly system-dependent features are imported from module SYSTEM. Although it lies in the nature of this module that there cannot be a standard for its contents, it typically exports the type ADDRESS, which is compatible with all pointer types. The type BYTE represents the basic unit of addressable storage.

SYSTEM Types

ADDRESS	a variable of this type holds an MC68000 address. The type´s size is 4. All integer arithmetic operators apply also to operands of this type, which is compatible with all pointer types. The arithmetic for this type is unsigned. Furthermore, type ADDRESS = POINTER TO BYTE. Hence, the type ADDRESS can be used to perform address computations and to export the results as pointers to values of type BYTE.
BYTE	1 uninterpreted byte: the smallest addressable unit of storage; assignment-compatible with all types of size 1 byte. Note that ARRAY OF BYTE is compatible with everything.
WORD	1 uninterpreted word: 2 consecutive bytes on an even address; assignment-compatible with all types of size 2.

SYSTEM Procedures

ADR(v)	the address (of type ADDRESS) of the variable v, which may be of any type.
ASH(x,n)	x * 2**n. Arithmetic shift x by n places. n ≥ 0 generates a left shift and n < 0 a right shift. The result type is the same as the type of x.
COM(x)	binary complement of x. The result type is the same as the argument type.
INLINE(n)	put n into the instruction stream. n must be a constant.
LSH(x,n)	logical shift x by n places. n ≥ 0 generates a left shift and n < 0 a right shift. The result type is the same as the type of x.
MSK(x,n)	x MOD 2**n in the proper sense of modulo arithmetic: returns the rightmost n bits of the variable x. n must be a constant in the range [1..32]. The result type is the same as the type of x.
REG(n)	the value of the MC68000 register #n. n must be a constant in the range [0 .. 15]: the D-registers are numbered from 0 to 7 (D0 .. D7) and the A-registers from 8 to 15 (A0 .. A7). The result type is LONGINT.
ROT(x,n)	rotate x by n places. n ≥ 0 rotates left, whereas n < 0 rotates right. The result type is the same as the type of x.
SETREG(n,x)	load x into the MC68000 register #n. If x is a simple type of size 4 bytes, the value is loaded; if x is a string or a procedure, the address is loaded. n must be a constant in the range [0 .. 15] specifying the register. D-registers are numbered from 0 to 7 (D0 .. D7) and A-registers from 8 to 15 (A0 .. A7).
TSIZE(T)	the number of bytes (of type INTEGER) used in memory for type T.
VAL(T,x)	x, of type T0, converted to a value of type T (without any security checks). VAL(T, x) is a replacement for the type transfer function T(x).

4.6. Differences and Restrictions

For the implementation of Modula-2 on the Macintosh some differences from the definition in [1] and restrictions must be considered.

Assignment compatibility
The following types are assignment-compatible to each other and the value is fully checked to be in range before assignment: (INTEGER, CARDINAL, LONGINT, LONGCARD), and (REAL, LONGREAL).

Type transfer functions and VAL
Type transfer functions are eliminated from the basic language repertoire! They can be obtained through the use of VAL (imported from the module SYSTEM). SYSTEM.VAL(T,x) is a replacement for the unchecked type transfer function T(x). Its value is x, interpreted as of type T. Its import is to ensure that machine-dependent type transfers are explicitly referred to in the program and are thus readily located.

Procedures
The compiler also offers a code procedure declaration. It can be used in definition and implementation modules and serves to introduce procedures implemented by supervisor calls. The code number n specifies a MC68000 opcode (typically a trap instruction) inserted as a word in the instruction stream where called. Evidently, such definitions are provided together with the operating system used. The declaration format is

```
PROCEDURE P(parameter list); CODE n;
```

Procedures referenced before declaration must be declared in advance by a FORWARD declaration. The format is

```
PROCEDURE P(parameter list); FORWARD;
```

The corresponding procedure must have the full header repeated and must lie at the same nesting level as the FORWARD declaration.

Forward references
No forward references are permitted, except in definitions of pointer types and in forward procedure declarations (see above).

Function procedures
The result type size of a function procedure must be 1, 2, 4, or 8 (that covers all simple types inclusive LONGREAL).

Data size
The maximum global (static) data size must be less than 32 kBytes per compilation unit. There's no limit on total data size over all modules or dynamic data size.

Code size
The maximum code size must be less than 32 kBytes per compilation unit. The total code size over all modules is not limited (the compiler modules have a total size of 99 kBytes).

Index types in array declarations
The index type must be a subrange type.

Subranges
The bounds of a subrange must be less than 32767 in absolute value and the difference MAX(Subrange) - MIN(Subrange) must be less than 32767.

Enumeration types
An enumeration may have at most 256 constants.

Sets
A set may have at most 16 elements.

Standard functions, procedures, types, and absolute variables
The procedures NEW, DISPOSE, TRANSFER, NEWPROCESS, and the type PROCESS are not predeclared (although defined as standard objects in earlier reports on the language). They will be implemented in separate modules.
The function SIZE is a standard procedure (globally defined), and it is identical to the function TSIZE defined in module SYSTEM. Its argument is a type or a variable; the result is the size of the type (or the variable) in number of bytes required for the type or for a variable of that type.
As NEW and DISPOSE are not predeclared, you have to write them as in the following example:

```
MODULE Example;
    IMPORT Storage;
    VAR p: POINTER TO T;                (* T denotes any type      *)
BEGIN
    Storage.ALLOCATE(p,SIZE(T));        (* this is NEW(p)          *)
    Storage.DEALLOCATE(p,SIZE(T));      (* this is DISPOSE(p)      *)
END Example.
```

Absolute variables
Absolute variables are not supported (must be accessed via pointers).

Procedures declared in definition modules
If a procedure (heading) is declared in a definition module, its body must be declared in the corresponding implementation module properly; it cannot be declared in an inner, local module.

4.7. Implementation Notes

The Module System: the MC68000 does not supply all the necessary instructions needed to implement Modula-2 [3]. A runtime support module providing the missing operations is therefore necessary. The compiler actually generates external calls to procedures of module "**System**", which is *always imported implicitly* for a program module. "System" (do not confuse it with the module SYSTEM!) is a normal Modula-2 module consisting of a definition and an implementation part. A minimal MC68000 implementation of System, for example, typically provides the Halt-mechanism, the 32-bit integer arithmetic procedures and the floating-point arithmetic procedures. If MC68040 code is executed (see Appendix A), however, only the Halt-mechanism of System is needed; the emulators for LONGINT, REAL and LONGREAL arithmetic are not called anymore.

The Implementation of REAL Arithmetic: as mentioned above, the MC68000 chip doesn't supply instructions needed to implement the REAL arithmetic of Modula-2. Therefore, the standard MacMETH Compiler, which generates native MC68000 code, will emulate the missing operations in software. There exist two different approaches to solve this problem for single-precision (32 Bit) REAL's: the so-called Fink arithmetic on one hand and an interface to the SANE package on the other hand. When you turn off the switch 'alwaysSANE' in the SANE-section of the User.Profile, the simple but fast Fink arithmetic is selected. When using Fink arithmetic, however, the control of exceptions and the results generated in exceptional cases are not consistent with the different Floating-Point Processors of Motorola. Turning 'alwaysSANE' on in the User.Profile will select the SANE emulator: SANE arithmetic is slower, but the results are more precise (in 80-bit extended precision) and the control of exceptions and the halt-disabled results generated are in full conformance with the Floating-Point Processors MC68881, MC68882 and MC68040. For double-precision (LONGREAL) arithmetic, the SANE interface is selected anyway.

The Implementation of Integer Arithmetic: for the types INTEGER and LONGINT the arithmetic is signed. Arithmetic overflow checks are fully supported if the corresponding compiler option is on. The operators DIV and MOD are implemented by means of the MC68000 instruction DIVS. For the types CARDINAL and LONGCARD the arithmetic is unsigned. DIV and MOD are implemented by the MC68000 instruction DIVU. No overflow checks are generated for these unsigned types. Note that the MC68000 hardware supplies quotient and remainder according to Euler´s arithmetic: *DIV returns the truncated quotient; and the sign of the result of a MOD-operation is the same as the sign of the dividend.*

Pointer-Checks: when a pointer is de-referenced, its value must not be NIL. This demand is asserted by a check generated by the compiler under the control of the R-option: if the R-option is on (this is the default!), a Halt-exception occurs for every dereferenced NIL-pointer.

Size and Alignment of basic types: the MC68000 uses byte addressing. However, data are transferred to and from memory in 16-bit words. Each type has an alignment factor k. Variables are aligned by the compiler to lie at an address a, such that a MOD k = 0 (otherwise the MC68000 would generate an odd address trap). Since allocation is sequential, i.e. variables are allocated in the order of their textual occurrence, the least amount of storage gets wasted through alignment, if declarations are grouped according to size. The same holds for record fields, and in this case is even more important. The following are the sizes and alignment factors of types:

Type	Size	Alignment Factor
BOOLEAN, CHAR, BYTE, enumerations	1	1
INTEGER, CARDINAL, WORD, BITSET, sets	2	2
LONGINT, LONGCARD, REAL, pointers, procedures	4	2
LONGREAL	8	2
arrays, records, string constants	multiple of 2	2

Data Size of structured types: the alignment rules also apply to the single data elements within arrays and records. The size of structured types is therefore accumulated according to the size and the alignments of its data elements. For arrays, the size is the product of the number of array elements (index range) and the aligned size (1 or a multiple of 2) of the single array elements. Note that the single array elements are allocated from lower to higher memory addresses. The final size of the array is the aligned space (a multiple of 2) needed to store all array elements. For records, the single fields are allocated according to the declaration sequence, from lower to higher memory addresses. Fields in variant parts are overlaid. The final size of the record is the aligned space (a multiple of 2) needed to store all record fields. Arrays, records and strings are always allocated on an even address (word-boundary).

Data Allocation: *allocation of data is sequential: variables are allocated in the order of their textual occurence, from higher to lower memory addresses. This principle applies to local data, parameters and global data as well.*

Global Data: in contrast to conventional compilers, global data is allocated from higher to lower memory. The first 2 bytes in the global data block are reserved for initialization. They contain a flag which indicates whether or not a separate module is already initialized. Global data are addressed relative to a dedicated address register (Address Register A4, called the *static base SB*).

Parameters: for a value parameter, the aligned size is allocated on the stack (a multiple of 2) and the supplied value is copied. This is also true for a structured value parameter. For variable parameters, 4 bytes are allocated on the stack. They contain the address of the supplied variable. For (variable and value) open array parameters, 6 bytes are allocated on the stack. They contain the address of the supplied array and its high index bound. If the array is supplied as a value parameter, it is internally copied by the procedure into its local data area.

Local Data, System Stack: the data stack of the MC68000 processor is growing by decrementing the system stack register (Address Register A7, the *stack pointer SP*). Local variables and parameters on this stack are addressed relative to a dedicated register (Address Register A6, the

mark pointer MP), which points to the so-called procedure mark. Local data are accessed with negative offsets, parameters with positive offsets relative to the mark pointer. The procedure mark needs 3 longwords on the stack: one for the return address and one for the dynamic link to the procedure mark of the calling procedure. Global procedures further have to save the current static base, while nested procedures need access to variables which are declared on static intermediate procedure levels. This is enabled by the static link, which refers to the procedure mark of the enclosing procedure on the next lower static level. The static link is set up by the calling procedure.

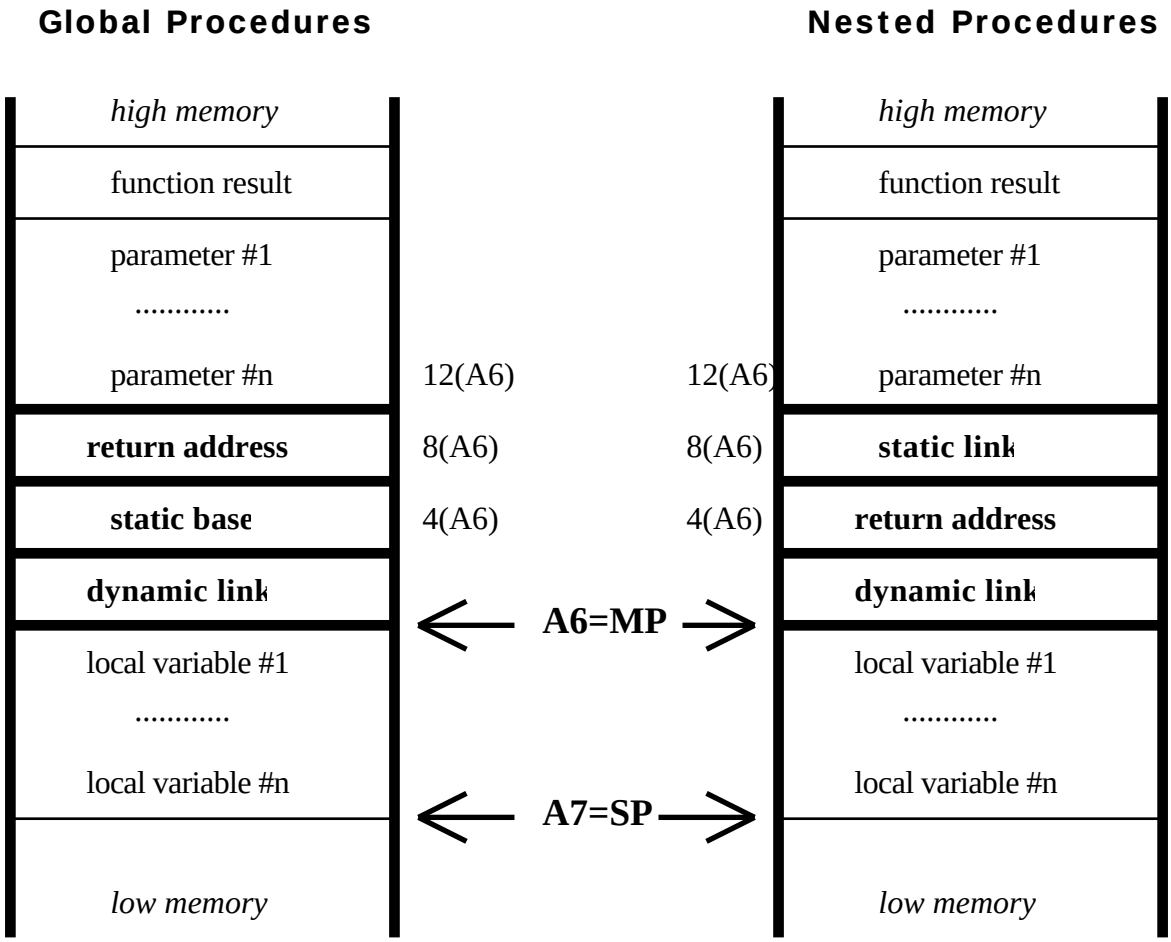

Fig. 4.7: System Stack and Procedure Mark

Register Allocation: the compiler manages the MC68000 Registers as follows: Scratch Registers are never reserved and therefore free at any time. Pool D-Registers are reserved and assigned by the compiler for the evaluation of arithmetic expressions and for index calculations. Pool A-Registers are reserved and assigned by the compiler for address computations and, in particular, for base addresses of arrays and records. The Registers A4, A5, A6 and A7 are dedicated.

Scratch D-Registers, free at any time:	D0 - D1
Pool D-Registers for expression evaluation:	D2 - D7
Pool A-Registers for address calculation:	A0 - A3
Dedicated A-Registers:	A4 = static base SB
	A5 = never used by compiler
	A6 = mark pointer MP
	A7 = stack pointer SP.

4.8. The Modula-2 Syntax of MacMETH

In this chapter, a description of the syntax of Modula-2 in EBNF is given. See also [1].

```
ident               = letter {letter | digit}.
number              = integer | real.
integer             = digit {digit} ["D"] | octalDigit {octalDigit} ("B"|"C") |
                      digit {hexDigit} "H".
real                = digit {digit} "." {digit} [ ScaleFactor | "E" | "D" ].
ScaleFactor         = ("E"|"D") ["+"|"-"] digit {digit}.
hexDigit            = digit |"A"|"B"|"C"|"D"|"E"|"F".
digit               = octalDigit | "8"|"9".
octalDigit          = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7".
string              = "'" {character} "'" | '"' {character} '"' .
qualident           = ident {"." ident}.
ConstantDeclaration = ident "=" ConstExpression.
ConstExpression     = expression.
TypeDeclaration     = ident "=" type.
type                = SimpleType | ArrayType | RecordType | SetType |PointerType |
                      ProcedureType.
SimpleType          = qualident | enumeration | SubrangeType.
enumeration         = "(" IdentList ")".
IdentList           = ident {"," ident}.
SubrangeType        = [qualident] "[" ConstExpression ".." ConstExpression "]".
ArrayType           = ARRAY SimpleType {"," SimpleType} OF type.
RecordType          = RECORD FieldListSequence END.
FieldListSequence   = FieldList {";" FieldList}.
FieldList           = [IdentList ":" type | CASE [ident] ":" qualident OF variant
                      {"|" variant}[ELSE FieldListSequence] END].
variant             = [CaseLabelList ":" FieldListSequence].
CaseLabelList       = CaseLabels {"," CaseLabels}.
CaseLabels          = ConstExpression [".." ConstExpression].
SetType             = SET OF SimpleType.
PointerType         = POINTER TO type.
ProcedureType       = PROCEDURE [FormalTypeList].
FormalTypeList      = "(" [[VAR] FormalType     {"," [VAR] FormalType}] ")"
                      [":" qualident].
VariableDeclaration = IdentList ":" type.
designator          = qualident {"." ident | "[" ExpList "]" | "^"}.
ExpList             = expression {"," expression}.
expression          = SimpleExpression [relation SimpleExpression].
relation            = "=" | "#" | "<>" | "<" | "<=" | ">" | ">=" | IN .
SimpleExpression    = ["+"|"-"] term {AddOperator term}.
AddOperator         = "+" | "-" | OR .
term                = factor {MulOperator factor}.
MulOperator         = "*" | "/" | DIV | MOD | AND | "&" .
factor              = number | string | set | designator [ActualParameters] |
                      "(" expression ")" | NOT factor | "~" factor.
set                 = [qualident] "{" [element {"," element}] "}".
element             = expression [".." expression].
ActualParameters    = "(" [ExpList] ")".
statement           = [assignment | ProcedureCall | IfStatement | CaseStatement |
                      WhileStatement | RepeatStatement | LoopStatement |ForStatement |
                      WithStatement | EXIT | RETURN[expression] ].
assignment          = designator ":=" expression.
ProcedureCall       = designator [ActualParameters].
StatementSequence   = statement {";" statement}.
```

```
IfStatement           = IF expression THEN StatementSequence {ELSIF expression
                        THEN StatementSequence}[ELSEStatementSequence] END.
CaseStatement         = CASE expression OF case {"|" case} [ELSEStatementSequence]
                        END.
case                  = [CaseLabelList ":" StatementSequence].
WhileStatement        = WHILE expression DO StatementSequenceEND.
RepeatStatement       = REPEAT StatementSequence UNTIL expression.
ForStatement          = FORident":=" expression TO expression [BY ConstExpression]
                        DO StatementSequence END.
LoopStatement         = LOOP StatementSequence END.
WithStatement         = WITH designator DO StatementSequence END .
ProcedureDeclaration  = ProcedureHeading ";"
                        (block ident  | FORWARD | CODE digit {hexDigit}"H").
ProcedureHeading      = PROCEDURE ident [FormalParameters].
block                 = {declaration} [BEGIN StatementSequence] END.
declaration           = CONST {ConstantDeclaration ";"} | TYPE {TypeDeclaration ";"} |
                        VAR {VariableDeclaration ";"} | ProcedureDeclaration ";" |
                        ModuleDeclaration ";".
FormalParameters      = "(" [FPSection {";" FPSection}] ")" [":" qualident].
FPSection             = [VAR] IdentList ":" FormalType.
FormalType            = [ARRAY OF] qualident.
ModuleDeclaration     = MODULE ident ";" {import} [export] block ident.
export                = EXPORT [QUALIFIED] IdentList ";".
import                = [FROM ident] IMPORT IdentList ";".
DefinitionModule      = DEFINITION MODULE ident ";" {import} {definition} END
                        ident ".".
definition            = CONST {ConstantDeclaration ";"} | TYPE {ident ["=" type] ";"} |
                        VAR {VariableDeclaration ";"} | ProcedureHeading ";" .
ProgramModule         = MODULE ident ";" {import} block ident "." .
CompilationUnit       = DefinitionModule | [IMPLEMENTATION] ProgramModule.
```

5. The Debugger

5.1. Introduction

The debugger is a tool with the aim of helping the programmer find the cause of execution errors in his program. The programmer is served best if he is allowed to debug on the same abstraction level at which he implemented the program. That means he should be able to view variables by name accompanied by their types and interpreted values, he should not have to struggle with addresses and their binary contents as is the case with many debuggers in current use. The debugger's main task and its relationship to the compiler is illustrated in the following diagram (Fig. 5.1):

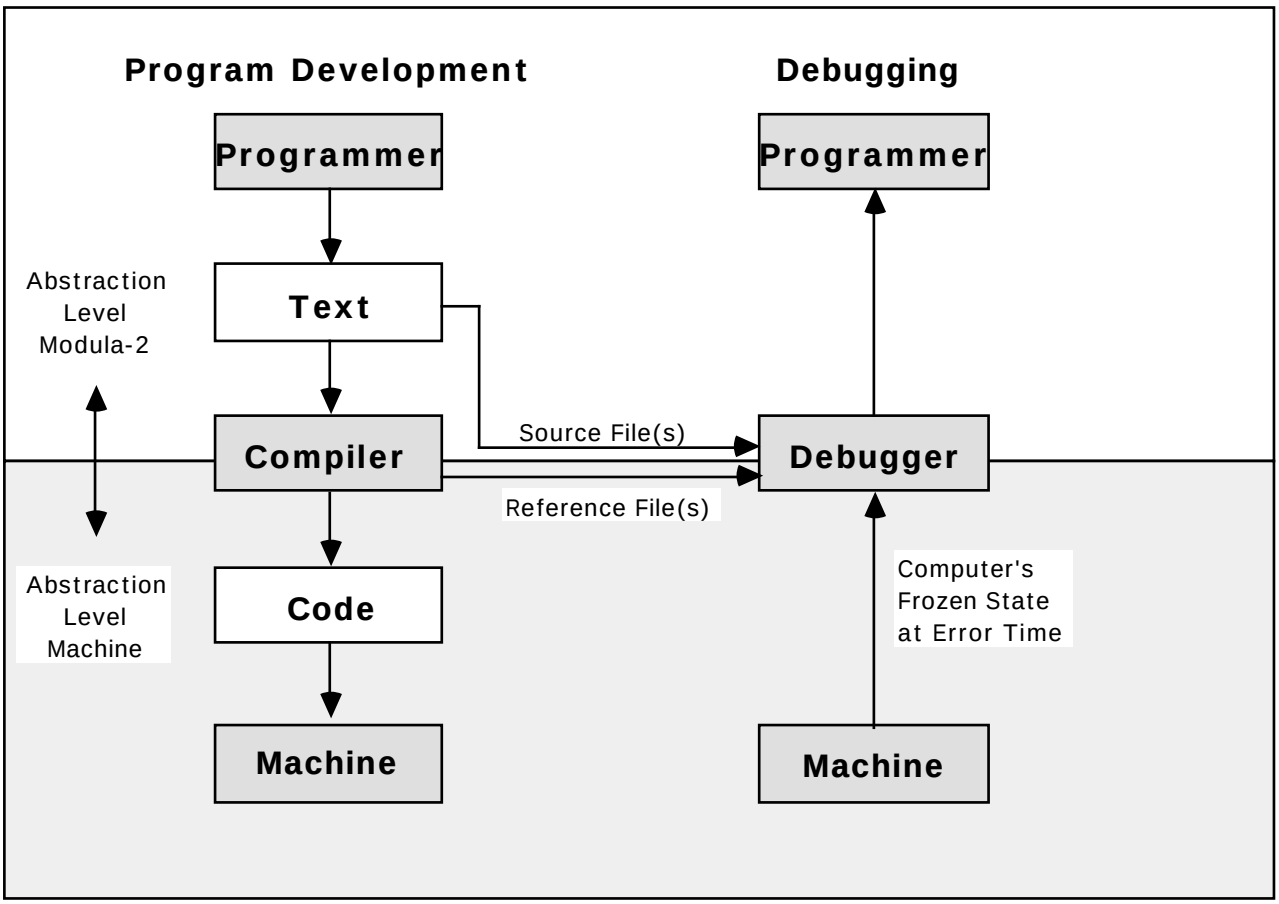

Fig. 5.1: Debugger Environment

On the left side the program development process is shown. The programmer writes his program in a high-level language, the compiler translates this text into machine code which, eventually, is loaded into the machine and executed. Thus, the compiler performs a mapping from a representative of the Modula-2 world into one of the machine world leaving the semantics invariant.

On the right side the program debugging process is shown. Suppose the program execution leads to a runtime error. The programmer wants to know the error position in the program text and find the reason for the program running into this error. The debugger maps the machine state in binary form into a form more easily understood by the programmer. Because during the mapping that the compiler performs information is lost, the debugger needs to be given program text and additional compilation information.

The compiler generates not only a code file but also a reference file containing information like variable names, types, declaration levels, addresses and such. The debugger also needs the source file to show the programmer the error position in the program text. It is important to

recognize that the debugger does not have to look at the machine code, it is static and known anyway. The debugger is only interested in the data, they are dynamic and only known at program execution time. The debugger is in contrast to the compiler not a translator but an interpreter (of the machine's state in binary form) as the mapping process is continuous and interactive.

5.2. Starting the Debugger

If your program runs into an execution error an alert box is displayed by the Macintosh as follows:

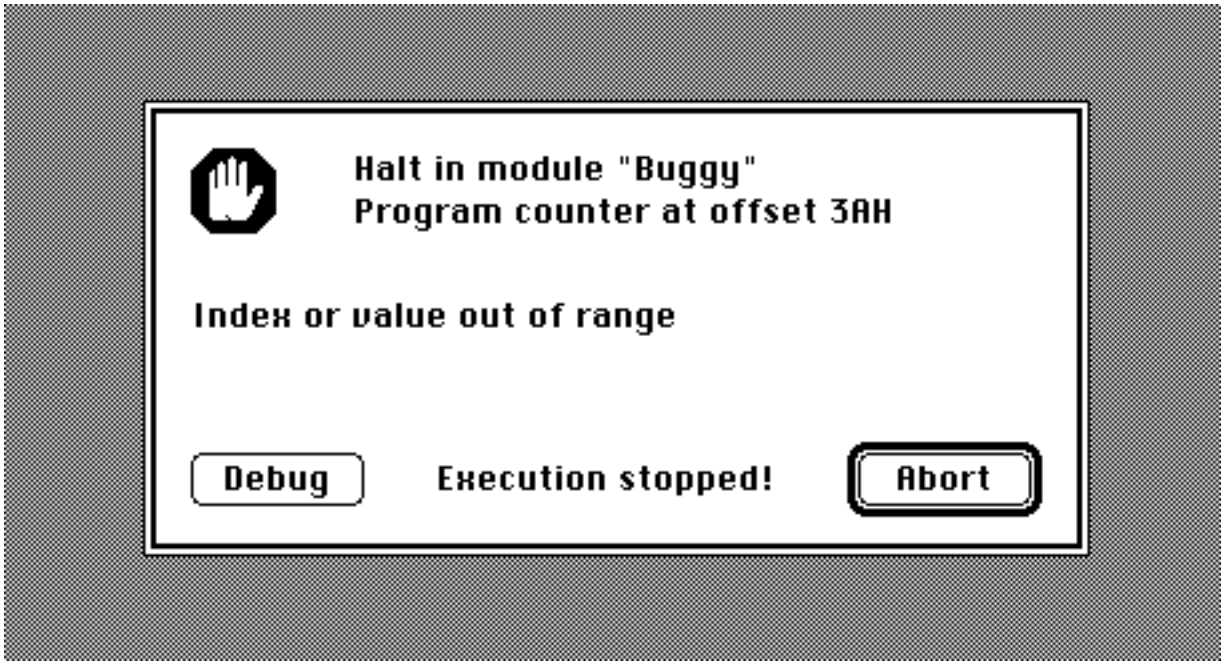

Fig. 5.2: Runtime Error

The box shows the type of error, the module the error occurred in, and the relative address in that module in form of a hexadecimal number. You have three options: enter the debugger, abort the program or continue execution (not all error types allow you to continue). Selecting Debug starts the debugger which presents the following screen image:

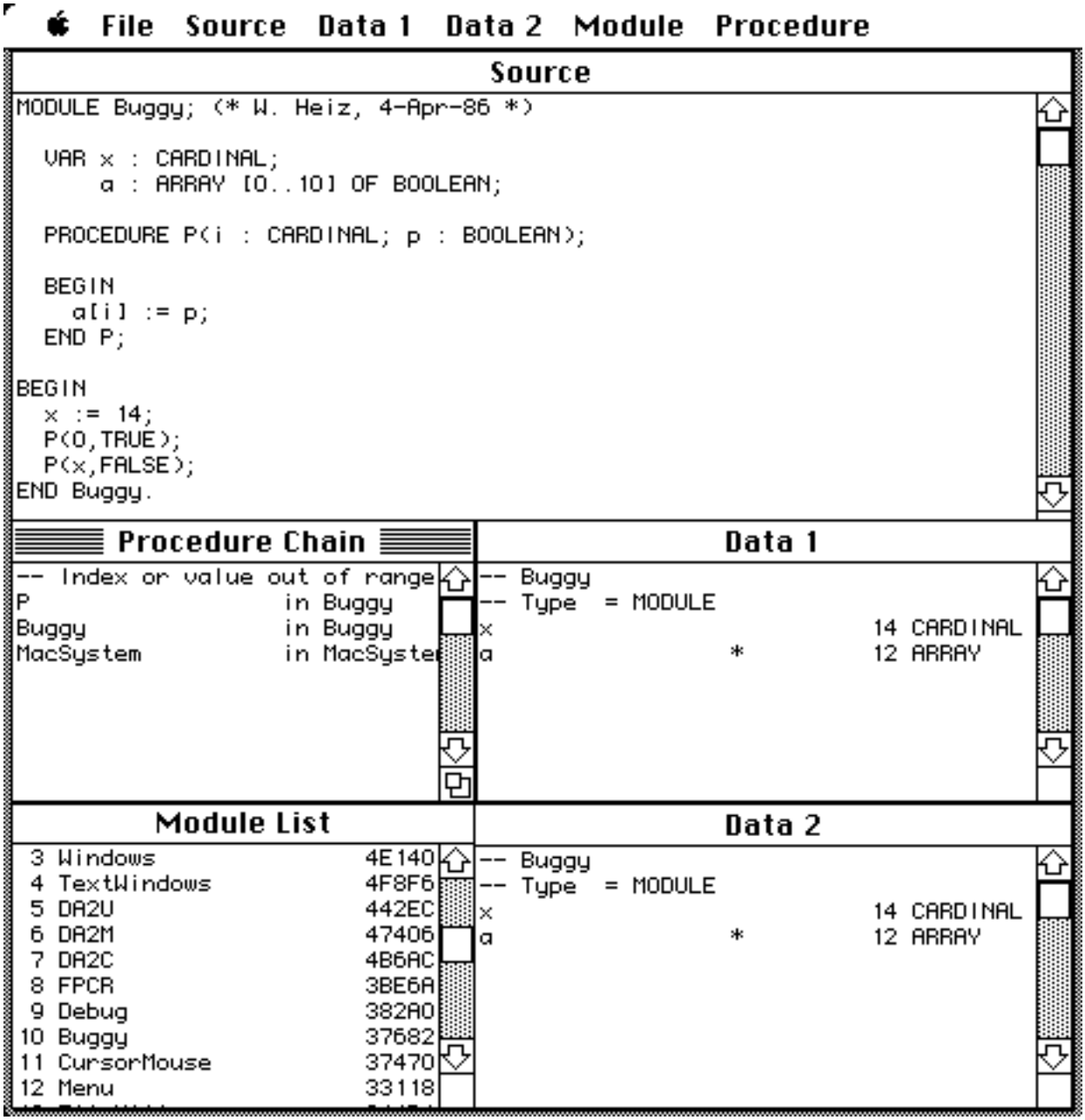

Fig. 5.3: Debugger Screen Image

5 Windows become visible:

- **Source Window:** Shows the source listing with the erroneous statement high-lighted.
- **Procedure Chain Window:** Indicates the reason for the program crash and shows the calling sequence of the active procedures.
- **Data 1 Window:** Shows all the variables of the last called procedure.
- **Module List Window:** Lists all loaded modules.
- **Data 2 Window:** Shows all the variables of the module containing the off-ending procedure.

Subsequently we shall take a closer look at each window.

5.3. Global Commands

Each Window has a scroll bar on the right. Clicking inside the scroll bar causes the contents to scroll as in the editor. Selecting Quit from the File menu leaves the debugger. The mouse buttons (ML and MR) are used as follows:

ML: Display data of selected object (selecting = pointing and clicking).
MR: Display source of selected object (if available).

5.4. Local Commands

The Source Window

The source window contains a listing which is selected through the procedure chain or the module list window. If selected through the procedure chain window the error position is highlighted.

ML: -
MR: -

Source Menu:	Ask On	Ask user for alternative file names if source file cannot be found. Normally the source file name is derived from the module name by adding ".MOD".
	Ask Off	Don't ask user for alternative file names.

The Data Windows

The two data windows display the data related with selected modules, procedures, arrays, records and pointers.

ML: Show further data of selected object if there is such. Objects with further data are marked with "*". The top line shows a path through which the present structure was reached. The path can be retraced in reverse order by selecting a previous component.
MR: -

Data Menus:	Expand	Expand window.
	Shrink	Shrink window.

The Module List Window

The module list window shows the list of loaded modules.

ML: Show source listing and global data of selected module.
MR: Show source listing of selected module.

Module Menu:	Data 1	Use data window 1 to display global data of modules.
	Data 2	Use data window 2 to display global data of modules.
	DEF	Show the definition listing of future selected modules.
	MOD	Show the implementation listing of future selected modules.

The Procedure Chain Window

The procedure chain window shows the error diagnostic and the calling sequence of all active procedures.

ML: Show source listing and local data of selected procedure.
MR: Show source listing of selected procedure.

Procedure Menu:	Data 1	Use data window 1 to display local data of procedures.
	Data 2	Use data window 2 to display local data of procedures.

6. Utility Programs

6.1. The Linker

The program "Link"collects the codes of separate modules of a program and writes them on one file. The program "Link" is called linker in this chapter.

Upon compilation of an individual module, the code generated by the Modula-2 compiler is written on an object file. An object file may be loaded by the loader of MacMETH directly. As a program usually consists of several separate modules, the loader has to read the code of the modules from several object files which are searched according to a default strategy. On one hand, this is time-consuming because several files must be searched; on the other hand, it could be useful to substitute a module from a file with a non-default name. These are some reasons for having a linker program.

Fig. 6.1: Starting the Linker

The linker simulates the loading process and collects the codes of all modules which are, directly or indirectly, imported by the so-called main module, i.e. the module which constitutes the main program. The linker applies the same default strategy as the loader to find an object file. A file name is derived from (the first 16 characters of) the module name. From the module name **AnyModule**, for example, the default file name **AnyModule.OBM** is derived.

First the linker prompts for the master object file of the main module (default extension "OBM"). Enter it by typing its name or via the file opening dialog by pressing TAB. After pressing EOL the linker starts the linking process by displaying the name of the result file preceeded by its path. A list of all the modules to be linked, starting with the name of the main module, follows:

```
MacMETH - Linker    V-2.6.2                    (input from user in boldface)
ETH Zuerich, VC/AF, 1-Sep-92

Master object file>Example.OBM

Linking ':Examples:Example.OBM':
  Example                                      main module
Terminal: Terminal.OBM                         default file name
  Terminal                                     directly imported module linked
TerminalOut: TerminalOut.OBM                   default file name
  TerminalOut                                  more modules linked from the library
TerminalIn: TerminalIn.OBM
  TerminalIn
EventBase: EventBase.OBM
  EventBase
```

Finally, if the linking process was successful, the linker displays again the resulting file and prompts for a next master object file.

```
':Examples:Example.OBM' linked
End of linkage

Master object file> --- escape
```

Terminate the linker by pressing RETURN, ESC or clicking the mouse.

Linking Options

The linker may be operated in various modes. These options are activated by adding a '/' when entering the master object file. Up to *two options in any combination* may be added.

/Q (query)
If this option is set, the linker prompts for the file names of each module to be imported. Type a file name (default extension "OBM") or simply accept the proposed name by pressing RETURN. Then the linker asks whether the found module shall actually be linked. To accept the module press "y", to ignore or remove it press "n". The linker continues to ask for any missing module until you have accepted an adequate object file by pressing "y" or until you press ESC. The latter means that this module *should not be linked*. Note that this behavior allows to replace already linked modules with new versions or to link programs only partly, so they may share some modules from a common library with other programs.

In query mode the linker accepts any file name, as long as the file actually contains the object code of the required module. Secondly it is always possible to specify file names via the standard open file dialog by pressing TAB. For example, you may wish to link into the previously linked program "Example" the LOG file variant of the TerminalOut module (see 7.17.2):

```
Master object file>Example.OBM/Q          query option (input from user in boldface)

Linking ':Examples:Example.OBM':
  Example                                 main module
  Terminal (y/n)? yes                     keep previously linked module
  TerminalOut (y/n)? no                   remove it
  TerminalIn (y/n)? yes                   keep it
  EventBase (y/n)? no                     remove it
EventBase> EventBase.OBM --- escape       final rejection => incomplete linkage
TerminalOut> TerminalOut.OBM (writes file) selecting LOG file variant via TAB
  TerminalOut (y/n)? yes                  add it
FileSystem> FileSystem.OBM                LOG file variant needs FileSystem
  FileSystem (y/n)? yes                   add it

':Examples:Example.OBM' linked
End of linkage
```

/A (application)
This option allows to generate stand-alone applications which may be started by a double click and which may be executed completely independent from the MacMETH development system. Any Modula-2 module can be linked to an application.

On the Macintosh, applications are identified by the so-called signature (or creator), which should be a four character long identification. Among other things the signature allows the Finder to associate desktop icons with a particular application. The signature of MacMETH is 'ETHM', for MacWrite it is 'MACA' etc. Hence, for each application to be linked find first a unique signature.

Secondly, within each application must exist a so-called resource fork. It contains among other things the object code of module System, a mandatory prerequisite for any MacMETH Modula-2 code execution (resource type 'CODE'), pictures (resource type 'PICT'), desktop icons (resource type 'ICN#'), or items required for dialogs (resource type 'DLOG', 'DITL'). Since the MacMETH

shell contains a resource fork with all resources required for standard program execution, a simple application linking process does not require a special resource fork. The following example links the previously linked module "Example" as an application with the signature 'MyAp' by using MacMETH's resource fork:

```
Master object file>Example.OBM/A              (input from user in boldface)
Signature>MyAp

Resource file>:Examples:Example.R --- escape   use default resources
  Copying resources from linking program ' MacMETH 3.2' .....
  Old bundle for signature 'ETHM' removed      Application decoupled from MacMETH

Linking application ':Examples:Example' (signature = 'MyAp'):
  Example
  Terminal
  TerminalIn
  TerminalOut
  FileSystem
EventBase: EventBase.OBM                       linked according to default strategy
  EventBase

Application ':Examples:Example' linked
End of linkage
```

The resulting application is named "Example" (extension "OBM" stripped) and is immediately ready for execution completely independent from MacMETH.

The linker supports more elaborate application development, for instance if you wish to develop an application with its own desktop icons etc. In this case you will have to add additional resources by using resource editing tools such as ResEdit (Anonymous, 1991 [5]; Alley & Strange, 1991 [6]). Proceed as follows: once linked, open the application by ResEdit to add the extra resources. Make sure you leave the resources copied by the linker in the application. Once your extra resources have been added, you may find it useful to store all resources in a separate file, preferably with the default extension "R", e.g. named "Example.R". This will allow you to repeat the linking process easily during further development cycles. During all subsequent application linking you may then answer the linker's request for a resource file by pressing RETURN instead of ESC. Note the linker checks the presence of the mandatory resources. If the linker detects a missing resource leading to a non-executable or fatally crashing program, it lists and marks the always mandatory resources.

Any linked application accepts the configuring specifications contained in the standard configuration file. An application looks in the same folder as it resides for a configuration file named "User.Profile". The found specifications, e.g. as made in the section 'Traps" or "SANE", will have exactly the same effect on the program behavior as if the unlinked program would be executed from within the MacMETH shell. If there is no configuration file available, the application will configure itself according to a safe strategy. This strategy depends on the current hard- and software and is described in Chapter 1.4.

Applications may also be linked in the query mode. Since *the linker accepts as a master object file also applications*, the query mode allows to replace modules already linked into an application with newer implementations. In particular this is also the case for the module System. Thus, older applications not running on newer Macintosh models, e.g. the Quadra models, may be easily updated by relinking them and using the default resource fork from the newest MacMETH shell (extra resources will have to be added in a separate step). Note that an only partly linked application will search for any missing module according to the default search strategy, i.e. it will look for the configuration file (User.Profile) and interpret the herein specified paths to search for the yet outstanding program parts.

Once linked, applications can still be executed from within the MacMETH shell (e.g. by specifying them in the "Menu" section of the configuration file). For instance you could link the compiler into a standalone application and still be able to use it from within the MacMETH shell.

6.2. The Decoder

The program "Decode" disassembles an object file. The program reads an object code file and generates a textfile with mnemonics for the machine instructions. It respects the structure of the object file as generated by the compiler. The program prompts for the name of the input file. Default extension is "OBM".

Example:
```
in>Buggy.OBM            (Input from user is in boldface)
- Buggy.OBM
```

The decoder displays all files it reads with the tag "-". Then it asks for an output file:

```
out>Buggy.DEC
```

Furthermore, it asks you whether or not it shall include the relative PC-values and the hexadecimal representation of the code; it also asks whether or not the source of the Modula-2 program shall be inserted in the output file at the appropriate locations:

```
full decode       (Y/N/ESC) ?  Y
include source    (Y/N/ESC) ?  Y
- Buggy.RFM
- Buggy.MOD
```

Typing ESC to any of these questions leaves the program without generating an output file.

The intended usage of this program is to check the compiler after modifications of the code generation; however this program may be used also to learn about code generation. In production there is no need to know the code generated by the compiler.

Example:

Input: Object file of the module "Buggy" (and possibly the source file of "Buggy", see Fig. 5.3)

Output: Decoded object file ("full decode ?" and "include source ?" both answered with "Y")

```
MC68020/68881 Modula-2 Decoder V-2.6
ETH Zuerich, CM/VC, 1-May-89
HEADER
  Key, Name = B928 03AD A7F8, Buggy
  Version   =    0
  CodeSize  =  110
  DataSize  =   42
  ConstSize =    0
  Procs     =    1
  Mods      =    2
IMPORT
  Key, Name = 0000 0000 0000, System
CODE
MODULE Buggy; (* W. Heiz, 4-Apr-86 *)
|-------------
0000: 4E71           | NOP
0002: 4E71           | NOP
0004: 2F0C           | MOVE.L    A4,-(A7)
0006: 287A FFF8      | MOVEA.L   (-8,PC),A4 /abs $00000000
000A: 4E56 0000      | LINK.W    A6,#0
000E: 08EC 0000 FFFE | BSET.B    #0,(-2,A4)
0014: 6706           | BEQ.B     *+8 /abs $0000001C
0016: 4E5E           | UNLK      A6
0018: 285F           | MOVEA.L   (A7)+,A4
001A: 4E75           | RTS
001C: 266C 0008      | MOVEA.L   (8,A4),A3
0020: 2653           | MOVEA.L   (A3),A3
```

```
0022: 4E93           | JSR       (A3)
0024: 6000 0028      | BRA.W     *+42 /abs $0000004E
|-------------

  VAR x : CARDINAL;
      a : ARRAY [0..10] OF BOOLEAN;

  PROCEDURE P(i : CARDINAL; p : BOOLEAN);
  BEGIN
|-------------
0028: 2F0C           | MOVE.L    A4,-(A7)
002A: 287A FFD4      | MOVEA.L   (-44,PC),A4 /abs $00000000
002E: 4E56 0000      | LINK.W    A6,#0
|-------------
    a[i] := p;
|-------------
0032: 342E 000E      | MOVE.W    (14,A6),D2
0036: 45BC 000A      | CHK.W     #10,D2
003A: 47EC FFD6      | LEA.L     (-42,A4),A3
003E: 17AE 000C 2000 | MOVE.B    (12,A6),(0,A3,D2.W)
|-------------
  END P;
|-------------
0044: 4E5E           | UNLK      A6
0046: 285F           | MOVEA.L   (A7)+,A4
0048: 205F           | MOVEA.L   (A7)+,A0
004A: 588F           | ADDQ.L    #4,A7
004C: 4ED0           | JMP       (A0)
|-------------

BEGIN
  x := 14;
|-------------
004E: 397C 000E FFE2 | MOVE.W    #14,(-30,A4)
|-------------
  P(0,TRUE);
|-------------
0054: 4267           | CLR.W     -(A7)
0056: 1F3C 0001      | MOVE.B    #1,-(A7)
005A: 6100 FFCC      | BSR.W     *-50 /abs $00000028
|-------------
  P(x,FALSE);
|-------------
005E: 3F2C FFE2      | MOVE.W    (-30,A4),-(A7)
0062: 4227           | CLR.B     -(A7)
0064: 6100 FFC2      | BSR.W     *-60 /abs $00000028
|-------------
END Buggy.

|-------------
0068: 4E5E           | UNLK      A6
006A: 285F           | MOVEA.L   (A7)+,A4
006C: 4E75           | RTS
|-------------

DATA
Entry Table:
0004  PROC #0

Strings:
```

6.3. The Program "ReadProfile"

This utility enables you to redefine dynamically the configuration settings of the MacMETH shell.

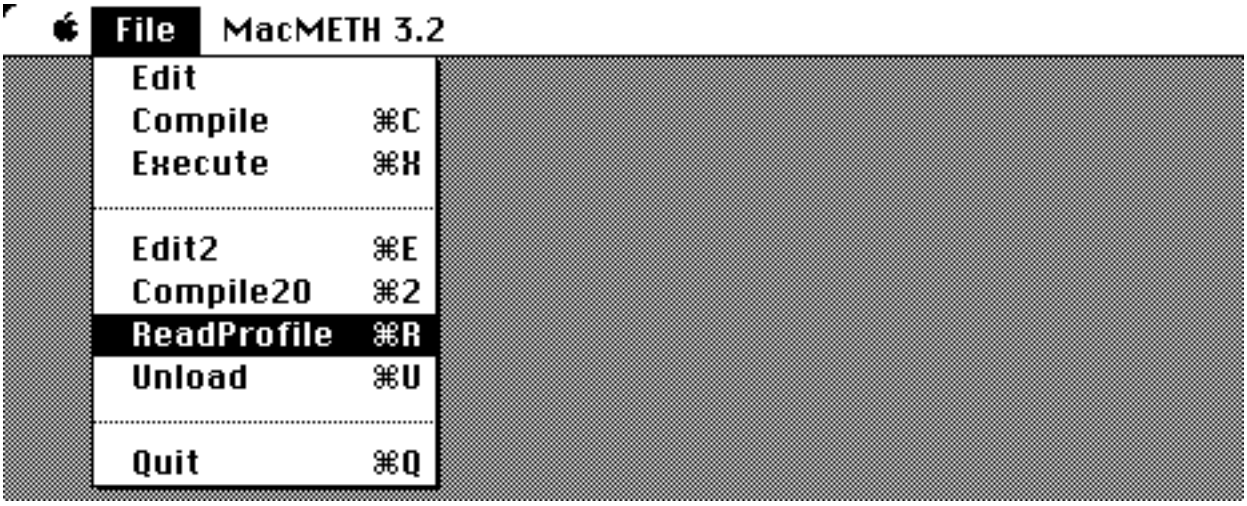

Fig. 6.3: Starting "ReadProfile"

Changes in the configuration file are best made with the editor. Save these changes onto the file "User.Profile" and quit the editor! Now select "ReadProfile" from the File menu: the ReadProfile utility immediately starts reading and processing the current "User.Profile" and updates all the specific paths, menu entries, modes and traps of the MacMETH shell accordingly. MacMETH from now on behaves according to the new values in the configuration file.

6.4. The Program "Unload"

Unload removes all permanently loaded modules, e.g. compilers, from the memory (see "System" section in Chapter 1.4).

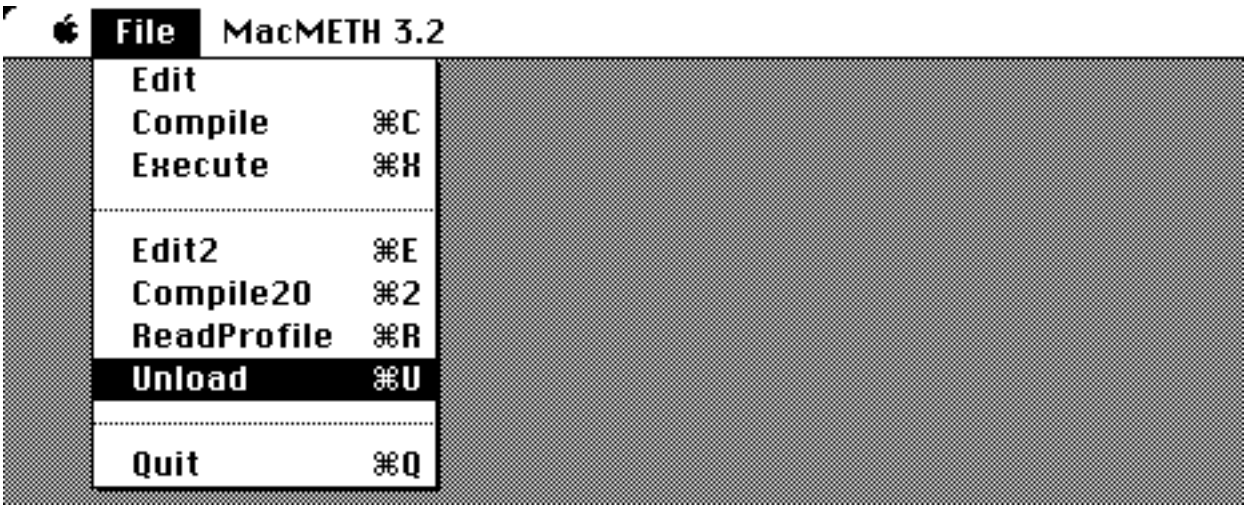

Fig. 6.4: Starting "Unload"

6.5. The Program "Print"

Print is a utility allowing you to print the text files created with the "Sara" editor. When you start the program by executing the tool "Print", the standard file selector box is displayed, allowing you to choose the file to print using the mouse. If you confirm your selection by clicking the "Open" button, printing starts. The text is printed according to the values you specified in the "Printer" section of the configuration file "User.Profile" (see "Printer" section in Chapter 1.4).

7. Library Modules

7.1. Introduction

This chapter describes a collection of some commonly used library modules on the Macintosh. For each library module a symbol file and an object file is released. The file names are derived from (the first 16 characters of) the module name ending with the extension "SBM" for symbol files and the extension "OBM" for object files. It is possible that some object files are pre-linked and therefore also contain the code of the imported modules.

Example:

Module name	FileSystem
Symbol file name	FileSystem.SBM
Object file name	FileSystem.OBM

List of the Library Modules

Conversions	string conversions from and to numbers	7.2.
CursorMouse	mouse handling and cursor tracking	7.3.
Dialog	creation and handling of dialog boxes	7.4.
Eventbase	simple event scheduler	7.5.
EV24	asynchronous interface to Macintosh modem port	7.6.
FileSystem	basic file input/output procedures	7.7.
FileUtil	additional file routines	7.8.
GraphicWindows	graphic window handling	7.9.
InOut	simple handling of formatted input/output	7.10.
LongMathLib, MathLib	mathematical functions for LONGREAL and REAL	7.11.
Menu	command selection from menus	7.12.
Printer	basic printer output procedures	7.13.
Storage	memory allocation/deallocation	7.14.
String	string handling	7.15.
System	runtime support and loader	7.16.
Terminal, TerminalIn, TerminalOut	screen oriented basic input/output procedures	7.17.
TextWindows	text window handling	7.18.
Windows	general window handling	7.19.

7.2. Conversions

This module provides a set of procedures for converting numbers and addresses to strings and vice versa.

Definition Module

```
DEFINITION MODULE Conversions;  (* Th. Wolf, 15-Jul-88 *)

(*------------------ Conversions from numbers to strings ------------------*)

PROCEDURE CardToString          (       CardNo                  : CARDINAL;
                                 Digits, Length, Base           : CARDINAL;
                                 VAR    str                     : ARRAY OF CHAR;
                                 VAR    ok                      : BOOLEAN);

PROCEDURE IntToString           (       IntNo                   : INTEGER;
                                        Digits, Length, Base    : CARDINAL;
                                 VAR    str                     : ARRAY OF CHAR;
                                 VAR    ok                      : BOOLEAN);

PROCEDURE AddrToString          (       Address                 : ADDRESS;
                                        Digits, Length, Base    : CARDINAL;
                                 VAR    str                     : ARRAY OF CHAR;
                                 VAR    ok                      : BOOLEAN);

PROCEDURE LongIntToString       (       LongNo                  : LONGINT;
                                        Digits, Length, Base    : CARDINAL;
                                 VAR    str                     : ARRAY OF CHAR;
                                 VAR    ok                      : BOOLEAN);

PROCEDURE RealToString          (       RealNo                  : REAL;
                                        Digits, Length          : CARDINAL;
                                 VAR    str                     : ARRAY OF CHAR;
                                 VAR    ok                      : BOOLEAN);

PROCEDURE LongRealToString      (       LongRealNo              : LONRREAL;
                                        Digits, Length          : CARDINAL;
                                 VAR    str                     : ARRAY OF CHAR;
                                 VAR    ok                      : BOOLEAN);

PROCEDURE RealToFixString       (       RealNo                  : REAL;
                                        Digits, Length          : CARDINAL;
                                 VAR    str                     : ARRAY OF CHAR;
                                 VAR    ok                      : BOOLEAN);

PROCEDURE LongRealToFixString   (       LongRealNo              : LONGREAL;
                                        Digits, Length          : CARDINAL;
                                 VAR    str                     : ARRAY OF CHAR;
                                 VAR    ok                      : BOOLEAN);

(*------------------ Conversions from strings to numbers ------------------*)

PROCEDURE StringToCard          (VAR    str                     : ARRAY OF CHAR;
                                        Start, Base             : CARDINAL;
                                 VAR    CardNo                  : CARDINAL;
                                 VAR    ok                      : BOOLEAN);
```

```
PROCEDURE StringToInt       (VAR str          : ARRAY OF CHAR;
                                 Start, Base  : CARDINAL;
                             VAR IntNo        : INTEGER;
                             VAR ok           : BOOLEAN);

PROCEDURE StringToAddr      (VAR str          : ARRAY OF CHAR;
                                 Start, Base  : CARDINAL;
                             VAR Address      : ADDRESS;
                             VAR ok           : BOOLEAN);

PROCEDURE StringToLongInt   (VAR str          : ARRAY OF CHAR;
                                 Start, Base  : CARDINAL;
                             VAR LongIntNo    : LONGINT;
                             VAR ok           : BOOLEAN);

PROCEDURE StringToReal      (VAR str          : ARRAY OF CHAR;
                                 Start        : CARDINAL;
                             VAR RealNo       : REAL;
                             VAR ok           : BOOLEAN);

PROCEDURE StringToLongReal  (VAR str          : ARRAY OF CHAR;
                                 Start        : CARDINAL;
                             VAR LongRealNo   : LONGREAL;
                             VAR ok           : BOOLEAN);

END Conversions.
```

Explanations

For all conversion procedures of the form 'NumberToString(...)' the following rules apply:

- The first parameter is the number that shall be converted. If you pass a negative number (if the type allows negative numbers!), the minus sign will always appear just to the left of thenumber.

- 'Digits' specifies the minimum number of digits that'll be used. If 'Digits' is greater than the actual number of digits needed and 'Base' is 2, 8 or 16, leading zeroes are added, otherwise leading blanks are added. If you specify more digits than there are characters in the result string, 'Digits' is reduced to the length of the string. If 'Digits' > 'Length', then 'Digits' is set to 'Length'. If 'Digits' is less than the number of characters needed, it's ignored.

- 'Length' is the number of characters that'll be used, i.e. the length of a field in which the number will be displayed right-adjusted. If 'Length' is less than the number ofcharacters needed, the field is filled with "#" and 'ok' := FALSE; if 'Length' specifies a field larger than the result string, the field length is reduced to the length of the string.
-'Base' gives the numbering system in which the number will be displayed. Possible values for 'Base' are 2, 8, 10 and 16; digits exceeding the decimal range 0..9 will be displayed as upper-case characters. If 'Base' has none of the values mentioned above, a default value of 10 is assumed.

- 'str' always is the result string; it's always ended by a 0C, except when the whole array is occupied by the number's character representation. If 'ok' = TRUE, the string contains thenumber's character representation, otherwise it contains 'Length' "#".

At the REAL and LONGREAL conversion routines, 'Digits' has a slightly different meaning: It specifies the maximum number of significant digits after the decimal point that will be displayed (before the decimal point, there's always just one digit).

If at the fixpoint conversion routines the number is too large to fit into the specified field, it's converted into it's exponential representation by a call either to 'RealToString' or to 'LongRealToString'.
The procedures 'CardToString', 'IntToString', 'AddrToString' and 'LongIntToString' convert the number the following way:

- If 'Base' = 16, a "H" is appended to the number's character representation.
- If 'Base' = 8, a "B" is appended to the number's character representation.
- If 'Base' = 2, a "%" is preceeding the number's character representation. If the number is negative, the "%" comes before the "-" sign.
- If 'Base' = 10, the string contains only the number's character representation itself.

REALs and LONGREALs are always converted to decimal representation ('Base' = 10).

For all procedures of the form 'StringToNumber(...)' the following rules apply:

- 'str' is the input string. It's a VAR-parameter for speedup reasons only and is not modified. If the characters in 'str' don't form a valid number or if the resulting number would exceed the type's range, 'ok' := FALSE.

- 'Start' is the position in 'str' where the number starts. All characters before this position are ignored. If 'Start' is greater than the number of characters in 'str', 'ok' := FALSE.

- The resulting number is only valid, if 'ok' is TRUE after the execution of the procedure.

- The number's representation in the string is as follows:

  - If the number is followed by a "H", conversion from base 16 (hexadecimal) is done
  - If the number is followed by a "B", conversion from base 8 (octal) is done.
  - If a "%" is preceeding the number, conversion from base 2 (binary) is done.
  - In all other cases, conversion from the given base is done.
  - REAL and LONGREAL numbers are always converted from base 10.

  If the number represented by the content of the string is negative, the '-' sign has to come after an eventually leading "%" character.

- If 'Base' has none of the values allowed, a default value of 10 is assumed.

- All numbers should be terminated either by the end of the string or a character <= " ".

This module doesn't provide special conversion procedures for LONGCARDs as the type LONGCARD is compatible with type ADDRESS.

Syntax descriptions in EBNF:

- INTEGERs and LONGINTs are converted according to the syntax:

```
IntNumber    =  BinaryInt | OctalInt | DecimalInt | HexInt.
BinaryInt    =  "%" [Sign] BinaryDigit {BinaryDigit}.
OctalInt     =  [Sign] OctalDigit {OctalDigit}"B".
DecimalInt   =  [Sign] DecimalDigit {DecimalDigit}.
HexInt       =  [Sign] HexDigit {HexDigit} "H".
BinaryDigit  =  "0" | "1".
OctalDigit   =  BinaryDigit | "2" | "3" | "4" | "5" | "6" | "7".
DecimalDigit =  OctalDigit | "8" | "9".
HexDigit     =  DecimalDigit | "A" | "B" | "C" | "D" | "E" | "F".
Sign         =  "-" | "+".
```

- Syntax for CARDINALs, ADDRESSes and LONGCARDs:

```
CardNumber    =  BinaryCard | OctalCard | DecimalCard | HexCard.
BinaryCard    =  "%" BinaryDigit {BinaryDigit}.
OctalCard     =  OctalDigit {OctalDigit}"B".
DecimalCard   =  DecimalDigit {DecimalDigit}.
HexCard       =  HexDigit {HexDigit} "H".
```

- The syntax for REALs and LONGREALs is the following:

```
RealNumber    =  [Sign] IntegerPart [FractionPart] [ScaleFactor].
IntegerPart   =  DecimalDigit {DecimalDigit}.
FractionPart  =  "." DecimalDigit {DecimalDigit}.
ScaleFactor   =  "E" [Sign] DecimalDigit {DecimalDigit}.
```

Imported Modules

Toolbox

Examples

Subsequently, we give you a series of examples giving you an idea how these conversion routines work. We assume that the string is long enough to hold the number's character representation.

```
CardToString(12457,8,10,16,str,Done);          everything ok; 'Base' = 16
    The result of this call will be
        str     =  " 000030A9H";
        Done    =  TRUE

CardToString(12457,8,10,10,str,Done);          everything ok; 'Base' = 10
    This will give you
        str     ="    12457";
        Done    =TRUE

CardToString(12457,3,5,10,str,Done);           'Digits' too small
    Result:
        str     =  "12457";
        Done    =  TRUE

CardToString(12457,5,3,10,str,Done);           'Length' too small
    Result:
        str     =  "###";
        Done    =  FALSE

CardToString(12457,2,4,10,str,Done);           'Length' and 'Digits' too small
    Result:
        str     =  "####"
        Done    =  FALSE

IntToString(-12457,5,5,10,str,Done);           'Length' too small (no space for sign!)
    Result:
        str     =  "#####";
        Done    =  FALSE
```

7.3. CursorMouse

Module to request the mouse buttons, mouse position and to display a cursor on the screen.

Definition Module

```
DEFINITION MODULE CursorMouse;  (* C. Vetterli, 26-Feb-86 *)

    CONST ML = 15; MM = 14; MR = 13;

    TYPE
        Pattern  =  RECORD
                        raster      : ARRAY[0..15] OF BITSET;   (*see Inside Macintosh*)
                        mask        : ARRAY[0..15] OF BITSET;
                        hotSpv, hotSph: INTEGER
                    END;

    PROCEDURE SetMouse      (x, y: INTEGER);
    PROCEDURE GetMouse      (VAR s: BITSET; VAR x, y: INTEGER);
    PROCEDURE MoveCursor    (x, y: INTEGER);
    PROCEDURE EraseCursor;
    PROCEDURE SetPattern    (VAR p: Pattern);
    PROCEDURE ResetPattern;

END CursorMouse.
```

Explanations

SetMouse(x,y);
This is a dummy procedure; the Macintosh handles cursor tracking itself. The procedure isprovided only for compatibility reasons.

GetMouse(b,x,y)
Returns the state of the buttons ('b') and the position of the mouse (x,y). 'b' returns the current mouse button state:

ML IN b = "mouse button pressed"
MM IN b = "command key pressed"
MR IN b = "option key pressed"

MM and MR are holdovers from the Lilith-implementation of MODULA-2 and are provided here for compatibility reasons, too. (The Lilith mouse has three buttons.)
By the way: you can get the screen size by a call to 'System.GetScreen'.

Note
The coordinates returned by 'GetMouse' differ from the ones used e.g. in module 'Graphic-Windows'. While in the latter the origin of the coordinate system is in the lower left corner, here (0,0) corresponds to the upper left corner.

MoveCursor(x,y);
Shows the cursor on the screen (parameters are dummy only).

EraseCursor;
Deletes the cursor on the screen.

SetPattern(MyPattern);
Installs a private cursor pattern. Subsequent calls of 'MoveCursor' will use this pattern. The cursor need not to be erased before calling 'SetPattern'.

ResetPattern;
Returns to the default cursor pattern (arrow) after 'SetPattern' has been called.

Imported Modules

Toolbox

Example

```
MODULE CursorExample; (* module shows you how to define an own cursor *)

    FROM CursorMouse IMPORT   Pattern, SetPattern, ResetPattern,
                              GetMouse, ML, MM, MR;

    VAR   OwnCursor           : Pattern;
          Buttons   :BITSET;
          i         : CARDINAL;
          x,y       : INTEGER;

BEGIN
    (* Define own cursor (a diagonal cross on white ground) *)
    FOR i:=0 TO 15 DO
        OwnCursor.raster[i]:={};
        INCL(OwnCursor.raster[i],i);
        INCL(OwnCursor.raster[i],15-i);
        OwnCursor.mask[i]:={0..15};
    END (* FOR *);
    OwnCursor.hotSpv := 8;
    OwnCursor.hotSph := 8;
    SetPattern(OwnCursor);
    LOOP
        GetMouse(Buttons,x,y);
        IF (MM IN Buttons) & (ML IN Buttons) THEN
            (* Command key and mouse button pressed: leave the program *)
            EXIT
        ELSIF (MR IN Buttons) & (ML IN Buttons) THEN
            (* Option key and mouse button pressed: reactivate own cursor *)
            SetPattern(OwnCursor);
        ELSIF (ML IN Buttons) THEN
            (* Only mouse button pressed: reactivate standard arrow cursor *)
            ResetPattern;
        END (* IF *);
    END (* LOOP *);
    ResetPattern;
END CursorExample.
```

7.4. Dialog

This module provides a collection of procedures for dialog management, enabling you to work with dialogs without using resources.

Definition Module

```
DEFINITION MODULE Dialog;  (* Ch. Widmer, 29-Jun-88 *)

    FROM  SYSTEM IMPORT BYTE, WORD;

    CONST  noDialog      = 0;
           lastDialog    = 8;
           windowItem    = 0;

    TYPE   Dialog        = [noDialog .. lastDialog];
           Item          = INTEGER;
           TextStyle     = (Bold,Italic,Underline,Outline,Shadow,Plain);
           EnumType      = BYTE;
           NumberType    = WORD;


    PROCEDURE New              (VAR d: Dialog; x0, y0, w, h: INTEGER);
    PROCEDURE Dispose          (VAR d: Dialog);
    PROCEDURE Open             (d: Dialog);
    PROCEDURE Close            (d: Dialog);
    PROCEDURE UserAction       (d: Dialog; VAR item: Item): BOOLEAN;

    PROCEDURE Button           (d: Dialog; VAR p: Item; default: BOOLEAN;
                                x, y: INTEGER; s: ARRAY OF CHAR);
    PROCEDURE Enumeration      (d: Dialog; VAR r: Item; VAR v: EnumType;
                                x, y, dx, dy: INTEGER; s: ARRAY OF CHAR);
    PROCEDURE Boolean          (d: Dialog; VAR c: Item; VAR v: BOOLEAN;
                                x, y: INTEGER; s: ARRAY OF CHAR);
    PROCEDURE String           (d: Dialog; VAR t: Item; VAR v: ARRAY OF CHAR;
                                x, y, w: INTEGER);
    PROCEDURE Number           (d: Dialog; VAR n: Item; VAR v: NumberType;
                                x, y, w, base: INTEGER);
    PROCEDURE LongNumber       (d: Dialog; VAR n: Item; VAR v: LONGINT;
                                x, y, w, base: INTEGER);

    PROCEDURE ActivateItem     (d: Dialog; i: Item);
    PROCEDURE DeActivateItem   (d: Dialog; i: Item);
    PROCEDURE RefreshItem      (d: Dialog; i: Item);
    PROCEDURE StaticString     (d: Dialog; t: ARRAY OF CHAR;
                                x, y: INTEGER; s: TextStyle);
    PROCEDURE StaticNumber     (d: Dialog; i: INTEGER; x, y: INTEGER; s: TextStyle);
    PROCEDURE Rectangle        (d: Dialog; x, y, w, h: INTEGER);

END Dialog.
```

Explanations

The origin of the coordinate system used in this module is in the lower left corner of the screen (or the dialog window, respectively).

New(d,x,y,w,h);
Creates a new modal dialog 'd' without displaying it on the screen. The lower left corner of the dialog window will be at coordinates (x,y), the outer rectangle of this window has width 'w' and height 'h'. The actual size of the inner rectangle (the one you actually can use) is 16 pixels less in each direction (width and height).

Dispose(d);
Deallocates the dialog in memory and deletes it from the screen. No further operations on 'd' are allowed hereafter, except another call to procedure 'New'.

Open(d);
Displays the previously defined dialog 'd' on the screen.

Close(d);
Deletes the dialog from the screen but doesn't deallocate it in memory.

UserAction(d,which);
If this procedure returns TRUE, the user either changed the value of an active item (returned in 'which') or clicked somewhere else in the dialog window. In the latter case, 'which' contains the value 'windowItem'.
This procedure should be called repeatedly in the main loop of your dialog handling procedure.
If the dialog specified by 'd' is not yet visible on the screen (i.e. if it's not opened) or if it's not the topmost one, procedure 'Open' will be called to display the dialog on the screen.

Button(d,butno,default,x,y,str);
Defines a button in dialog 'd'; the item number of this button is returned in 'butno'. The lower left corner is at coordinates (x,y) local to the dialog window, 'str' is the button's text. The first button installed with 'default' = TRUE is the default button; it's outline is drawn bold and the user not only may set it by clicking it with the mouse but also by typing the RETURN or the ENTER key.

Attention:
The following procedures 'Enumeration', 'Boolean', 'String', 'Number' and 'LongNumber' all require a VAR parameter 'v' which is used to return the value the user entered. As the dialog module tracks the user's actions and continually updates these values, the variables passed in place of these parameters must exist all the time the dialog exists (i.e. they should be declared before the 'New' statement and should exist at least until just after the call to procedure 'Dispose').

Enumeration(d,rbut,v,x,y,dx,dy,str);
Defines a whole group of so-called radio buttons in dialog 'd'. The first radio button has its lower left corner at coordinates (x,y), 'dx' and 'dy' give the offset to the next radio button (if 'dx' = 0, you'll get a column of radio buttons [vertical], if 'dy' = 0, the result will be a row of radio buttons [horizontal]).'v' should be set to the value of the enumeration corresponding to the radio button initially set. The number n of radio buttons is specified by the button string 'str'. The syntax of such a button string is the following:

```
ButtonString = RadioButton {"|" RadioButton}.
RadioButton  = {Character}.
```

The item number 'rbut' designates the whole group, the numbers 'rbut'+1 .. 'rbut'+n designate the buttons themselves.

Boolean(d,chbox,v,x,y,str);
Defines a check box in dialog 'd'; it's item number is returned in 'chbox'. The lower left corner of the check box is at coordinates (x,y), the initial value of 'v' is set to TRUE, meaning that the check box initially is checked. The string 'str' is written to the right of the check box.

String(d,textno,v,x,y,w);
Defines an editable text field in dialog 'd' with lower left corner at coordinates (x,y) and width 'w', enabling the user to modify the string 'v'. 'v' should contain the initally displayed string. The item number of the text field is returned in 'textno'.

Number(d,n,v,,x,y,w,base); LongNumber(d,n,v,x,y,,w,base);
Defines an editable field in dialog 'd' with lower left corner at (x,y) and width 'w'. The number 'v' is displayed according to the given base; if 'base' is negative, the number will be displayed signed. 'base' should be in the range [-16..-2] or [2..16].

ActivateItem(d,i);
The item 'i' in dialog 'd' is enabled, i.e. it's displayed in black and the user may change it (again).

DeActivateItem(d,i);
The item 'i' in dialog 'd' is disabled and displayed in gray.

RefreshItem(d,i);
Updates the value of the item 'i' in dialog 'd' according to the value of the variable passed in place of 'v' at the creation of the item.

StaticString(d,str,x,y,s);
Writes the static (unchangeable) string 'str' in dialog 'd' at coordinates (x,y) according to the text style specified in 's'.

StaticNumber(d,i,x,y,s);
Writes the static number 'i' in dialog 'd' at coordinates (x,y) according to text style 's'.

Rectangle(d,x,y,w,h);
Draws a rectangle in dialog 'd'. The lower left corner of the rectangle is at coordinates (x,y), its width is 'w', its height 'h'. If 'w' = 1, a vertical line is drawn; if 'h' = 1, a horizontal line is drawn.

Imported Modules

Conversions
EventBase
System
Toolbox

Example

```
MODULE DialogExample; (* demonstrates dialog handling *)

    FROM Dialog IMPORT      Dialog, Item, New, Dispose, Close, UserAction, Button,
                            Boolean, Enumeration, String, StaticString, TextStyle,
                            windowItem;

    TYPE        RadioType = (Button1,Button2);

    VAR   dialog            : Dialog;
          Quit, Cancel, Check,
          Radio, Text, TheItem   : Item;
          RadioValue        : RadioType;
          CheckValue        : BOOLEAN;
          TextValue         : ARRAY [0..63] OF CHAR;

    PROCEDURE Beep(Duration : INTEGER); CODE 0A9C8H;
   (* Code procedure, see chapter 8! *)

BEGIN
    (* Create dialog *)
    New(dialog, 50,50,400,200);
    (* Define dialog items *)
    Button(dialog,Quit,TRUE,90,10,"Quit");
    Button(dialog,Cancel,FALSE,250,10,"Cancel");
    Boolean(dialog,Check,CheckValue,10,100,"Check Box");
    RadioValue := Button2;
    Enumeration(dialog,Radio,RadioValue,200,100,0,20,"Radio Button 1|Radio Button 2");
    TextValue := "This is an example.";
    String(dialog,Text,TextValue,10,50,364);
    StaticString(dialog,"Sample Dialog",150,150,Underline);
    (* Dialog defined; display it and monitor user's actions *)
   REPEAT
      LOOP
         IF UserAction(dialog,TheItem) THEN
            IF (TheItem = Quit) OR (TheItem = Cancel) THEN
               EXIT
            ELSIF (TheItem = windowItem) THEN
               Beep(5)
            END (* IF *)
         END (* IF *)
      END (* LOOP *);
      Close(dialog)
   UNTIL (TheItem = Quit);
   Dispose(dialog)
END DialogExample.
```

7.5. EventBase

Module EventBase provides a simple scheduler intended to allow arranging a Modula-2 program on the Macintosh as a collection of event handling subroutines. This module is conceptually built on top of the Macintosh Toolbox Event Manager and its main goal is to support the decomposition of an otherwise tremenduously large CASE-statement for event handling into independent modules.

"A typical Macintosh application program is event-driven: It decides what to do from moment to moment by asking the Event Manager for events and responding to them one by one in whatever way is appropriate." (Inside Macintosh, About the Toolbox Event Manager)

Definition Module

```
DEFINITION MODULE EventBase;    (* C. Vetterli, 4-Jan-85 / 26-Feb-85 *)

   CONST
      (* events *)
      nullEvent    =   0; mouseDown =   1; mouseUp   =   2; keyDown =   3;
      keyUp        =   4; autoKey   =   5; updateEvt =   6; diskEvt  =   7;
      activateEvt =   8; networkEvt = 10; driverEvt = 11; app1Evt  = 12;
      app2Evt     = 13; app3Evt    = 14; app4Evt   = 15;

   TYPE
      Point         = RECORD
                      v, h: INTEGER
                      END;

      EventRecord    = RECORD
                        what       : INTEGER;      (* event code (see above) *)
                        message    : LONGINT;      (* event message *)
                        when       : LONGINT;      (* ticks since startup *)
                        where      : Point;        (* mouse location *)
                        modifiers  : INTEGER;      (* modifier flags *)
                       END;

      EventHandler   = PROCEDURE(VAR EventRecord): BOOLEAN;
        (* this procedure type is used by the scheduler for distributing events obtained by    *)
        (* GetNextEvent to the activated tasks                                                 *)


   PROCEDURE PushTask            (EventProc: EventHandler): INTEGER;
   PROCEDURE PopTask             (taskNum: INTEGER);

   PROCEDURE PollEventTasks;

   PROCEDURE GetBusyReadEvent   (VAR event: EventRecord): BOOLEAN;

END EventBase.
```

Explanations

EventHandler
: This procedure type is used by the scheduler for distributing events obtained by 'GetNextEvent' (see "Inside Macintosh") to the activated tasks. The EventHandler first has to analyse the event, then either handle it and return TRUE or not handle it and return FALSE.

PushTask(EventProc);
Pushes the task 'EventProc' onto the scheduler and activates it. 'PushTask' returns the task number. You may not push more than six tasks, if you try to push a seventh, 'PushTask'returns zero.

PopTask(taskNum);
Pops the task with number 'taskNum' from the scheduler and deactivates it.

PollEventTasks;
This procedure is to be called repeatedly in the main loop of a program. PollEventTasks performs a 'GetNextEvent', distributes events to the activated tasks and calls the Toolbox procedure 'SystemTask' if no events are pending.

GetBusyReadEvent(evt)
KeyDown or AutoKey events that remain unhandled by the tasks are buffered. By calls to this procedure you can get the events one after another. Module "TerminalIn" call this procedure, too, which allows the module "TerminalIn" to run as lowest priority handler.

Imported Modules

Toolbox

Example

```
MODULE EventExample;

   (* This module demonstrates event handling. This example doesn't make much        *)
   (* sense, but a sensible example would be much longer...                          *)

   FROM   EventBase        IMPORT    mouseDown, mouseUp, keyDown, Point, EventRecord,
                           PushTask, PopTask, PollEventTasks;

   FROM   InOut            IMPORT    WriteString, WriteInt, Write, WriteLn;

   VAR   MouseTask, KeyTask     : INTEGER;
         Quit                   : BOOLEAN;

   PROCEDURE TrackMouse(VAR Event : EventRecord) : BOOLEAN;

   BEGIN
      IF (Event.what # mouseDown) & (Event.what # mouseUp) THEN
         RETURN FALSE
      END (* IF *);
      IF Event.what = mouseDown THEN
         WriteString("You pressed the mouse button!  Coordinates : (");
      ELSE
         WriteString("You released the mouse button! Coordinates : (");
      END (* IF *);
      WriteInt(Event.where.h,3);
      Write(",");
      WriteInt(Event.where.v,3);
      Write(")");
      WriteLn;
      RETURN TRUE
   END TrackMouse;
```

```
   PROCEDURE KeyPressed(VAR Event : EventRecord) : BOOLEAN;
      VAR i :   RECORD
                   CASE :CARDINAL OF
                        0 : long  : LONGINT;
                       |1 : hi,lo : CARDINAL
                   END (* CASE *)
                END;

   BEGIN
      IF (Event.what # keyDown) THEN
         RETURN FALSE
      END (* IF *);
      WriteString("You pressed the key '");
      i.long := Event.message;
      Write(CHR(i.lo MOD 256));
      Quit := (CHR(i.lo MOD 256) < " ");
      Write("'");
      WriteLn;
      RETURN TRUE
   END KeyPressed;

BEGIN (* EventExample *)
   MouseTask   := PushTask(TrackMouse);
   KeyTask     := PushTask(KeyPressed);
   REPEAT
      PollEventTasks
   UNTIL Quit;
   PopTask(KeyTask);
   PopTask(MouseTask)
END EventExample.
```

7.6. EV24

This module provides procedures for accessing the Macintosh modem port.

Definition Module

```
DEFINITION MODULE EV24; (* VC, 22-Dec-85 / BH, 22-Apr-86 / VC, 8-May-87 *)

  CONST
      (* parity values *)
      none        = 0;
      odd         = 1;
      even        = 3;
      (* receiver errors *)
      SwOrErr     = 0;      (* software overrun error    *)
      PariErr     = 4;      (* parity error              *)
      HwOrErr     = 5;      (* hardware overrun error    *)
      FramErr     = 6;      (* framing error             *)

  VAR  error: INTEGER;      (* contains system errors occurred in last call (0 = no error) *)

  PROCEDURE Initialize          (port, baud, data, stop, parity, bufferSize: INTEGER);

  PROCEDURE SetFlow             (port: INTEGER);
  PROCEDURE Break               (port: INTEGER);
  PROCEDURE ReceiverErrors      (port: INTEGER; VAR err: BITSET);

 (* use the default (modem) or last initialized port *)
  PROCEDURE Write               (ch: CHAR);
  PROCEDURE Read                (VAR ch: CHAR);
  PROCEDURE BusyRead            (VAR ch: CHAR; VAR got: BOOLEAN);

 (* use designated port (should be initialized by a call to Initialize) *)
  PROCEDURE WritePort           (port: INTEGER; ch: CHAR);
  PROCEDURE ReadPort            (port: INTEGER; VAR ch: CHAR);
  PROCEDURE BusyReadPort        (port: INTEGER; VAR ch: CHAR; VAR got: BOOLEAN);

  PROCEDURE CloseEV24;

END EV24.
```

Explanations

The procedures in this module constitute an asynchronous interface to the Macintosh modem port. Transmission is full duplex, using ".Ain" and ".Aout" channels and/or ".Bin" and ".Bout" channels.

Initialize(port, baud, data, stop, parity, bufferSize);
Initializes the designated port. The meanings and possible values of the parameters are as follows :

port: 0, 1;
Selects the port to initialize(0 = Modem, 1 = Printer).
baud: 3, 6,12, 24, 48, 96, 192, 384, 576;
Selects the baud rate. The real baud rate is 'baud' * 100.
data: 5, 6, 7, 8;
Number of data bits.

stop: 1, 15, 2;
Number of stop bits (15 means 1.5).
parity: none, odd, even;
Selects parity bits.

If any of these values are zero, the following default values are set :
9600 baud, 8 data bits, 2 stop bits, no parity bit, 1000 byte buffers

SetFlow(p);
Enables Xon/Xoff control on port 'p'.

Break;
Sends a BREAK signal on the output channel of port 'p'.

ReceiveErrors(p,err);
Returns in 'err' all errors that occured since the last call to 'ReceiveErrors'.

The following three procedures operate on the default port (modem) or, if there have been initializations of other ports, on the last initialized one. This port is subsequently referred to as 'actual port'.

Write(ch);
Sends the character 'ch' to the actual port.

Read(ch);
Reads a character ('ch') from the actual port. If there's currently no character waiting to be read, the procedure waits until a character was received.

BusyRead(ch,got
As 'Read', but when there's no character waiting to be read at the actual port, the procedure immediatly returns with 'got' = FALSE and 'ch' undefined. If there's been a character waiting, 'ch' contains this character and 'got' = TRUE.

The following three procedures operate on the port specified in their parameter lists. This port should be initialized by a call to 'Initialize' before calling any of the three procedures below.

WritePort(p,ch);
Writes the character 'ch' to port 'p'.

ReadPort(p,ch);
Reads the character 'ch' from port 'p'. If there's currently no character waiting to be read at port 'p', the procedure waits until a character was received.

BusyReadPort(p,ch,got);
As 'ReadPort', but when there's no character waiting to be read at port 'p', the procedure immediatly returns with 'got' = FALSE and 'ch' undefined. If there's been a character waiting, 'ch' contains this character and 'got' = TRUE.

CloseEV24;
Closes all open serial links.

Imported Modules

DeviceMgr
System
Toolbox

7.7. FileSystem

Library module providing basic sequential file access. A description of this module is included in the Modula-2 manual [1].

Definition Module

```
DEFINITION MODULE FileSystem;  (* W. Heiz, 4-Feb-86; A. Fischlin, 1992 *)

   FROM SYSTEM IMPORT WORD;

   TYPE
      Response  =   (done, notdone);

      File      =   RECORD
                    refNum,
                    volRef      : INTEGER;
                    firstPos,
                    lastPos,
                    curPos      : LONGINT;
                    res         : Response;
                    eof,
                    dirty       : BOOLEAN;
                    nameString  : ARRAY [0..63] OF CHAR;
                    buffer      : ARRAY [0..1024-1] OF CHAR;
                    END;

   PROCEDURE Lookup         (VAR f: File; filename: ARRAY OF CHAR; new: BOOLEAN);
   PROCEDURE Close          (VAR f: File);
   PROCEDURE Delete         (VAR f: File);
   PROCEDURE Rename         (VAR f: File; filename: ARRAY OF CHAR);
   PROCEDURE SetPos         (VAR f: File; highpos, lowpos: CARDINAL);
   PROCEDURE GetPos         (VAR f: File; VAR highpos, lowpos: CARDINAL);
   PROCEDURE Length         (VAR f: File; VAR highpos, lowpos: CARDINAL);
   PROCEDURE ReadWord       (VAR f: File; VAR w: WORD);
   PROCEDURE WriteWord      (VAR f: File; w: WORD);
   PROCEDURE ReadChar       (VAR f: File; VAR ch: CHAR);
   PROCEDURE WriteChar      (VAR f: File; ch: CHAR);

   PROCEDURE LastIOErr      ():INTEGER;

   TYPE FType  =   ARRAY [0..3] OF CHAR;

   PROCEDURE SetDefItFileType           (type, creator: FType);
   PROCEDURE SetCompFileTypes           (compCreator, sbmType, obmType, rfmType : FType);
   PROCEDURE ResetDefItFileTypes;
   PROCEDURE SetBrowsingLookupMode      (activate : BOOLEAN);
   PROCEDURE GetBrowsingLookupMode      (VAR activate : BOOLEAN);

END FileSystem.
```

Explanations

Lookup(f, filename, new)
Looks for the actual file with the given file name. If the file exists, it is connected to 'f' (opened). If the requested file is not found or 'new' is TRUE, a permanent file is created with the given name. After the call

f.res	= done	if file 'f' is connected
f.res	= notdone	if the file does not exist or some error occured

Close(f)
Terminates any actual input or output operation on file 'f' and disconnects the variable 'f' from the actual file.

Delete(f)
Terminates any actual input or output operation on file 'f' and disconnects the variable 'f' from the actual file. The actual file is deleted hereafter.

Rename(f, filename)
Changes the name of file 'f' to filename. After the call

f.res = done	if file 'f' is renamed
f.res = notdone	if a file with filename already exists, or some error occured

SetPos(f, highpos, lowpos)
A call to procedure 'SetPos' sets the current position of file 'f' to 'highpos'*2^{16}+ 'lowpos'. The new position must be less or equal the length of the file.

GetPos(f, highpos, lowpos)
Returns the current file position. It is equal to 'highpos'*2^{16}+'lowpos'.

Length(f, highpos, lowpos)
Gets the position just behind the last element of the file (i.e. the number of BYTEs stored on the file). The position is equal to 'highpos'*2^{16}+'lowpos'.

WriteChar(f, ch), WriteWord(f, w)
Procedure WriteChar (WriteWord) appends character 'ch' (word 'w') to file 'f'.

ReadChar(f, ch), ReadWord(f, w)
Procedure ReadChar (ReadWord) reads the next character (word) from file 'f' and assigns it to 'ch' ('w'). If ReadChar has been called without success, 0C is assigned to 'ch'. f.eof implies ch = 0C. The opposite, however, is not true: ch = 0C does not imply f.eof. After the call

f.eof = FALSE	ch (w) has been read
f.eof = TRUE	read operation was not successful

If f.eof is TRUE:

f.res = done	end of file has been reached
f.res = notdone	some error occured

LastIOErr()
returns IO error code resulting from last IO-operation. noErr = 0, for other error codes see Inside Macintosh.

SetDefltFileType(type, creator)
Subsequent calls to Lookup with 'new' = TRUE result in files with the specified 'type' and 'creator' (defaults: type = 'TEXT', creator = 'MEDT' for MEdit).

SetCompFileTypes(compCreator, sbmType, obmType, rfmType)
Subsequent compilations result in files of the specified types and the creator 'compCreator'.

ResetDefltFileTypes
Reverses the effect of previous calls to SetCompFileTypes to the defaults: 'compCreator' = 'ETHM', 'sbmType' = 'MSYM', 'obmType' = 'MOBJ', and 'rfmType' = 'MREF'.

SetBrowsingLookupMode(activate)
If called with 'activate' = TRUE, subsequent calls to Lookup with 'new' = FALSE open the file in a read-only mode allowing other applications to have the file opened simultaneously. 'activate' = FALSE clears this mode and is the default mode.

GetBrowsingLookupMode(activate)
Allows to inquire the current mode. Typically called before SetBrowsingLookupMode to save the current mode for later restoration.

Imported Modules

System
Toolbox

Example

```
MODULE FileExample;

   (* this program copies the file "Example.TXT" into the file "Copy.BAK". *)
   (* demonstrates the access to text files.                               *)

   FROM FileSystem IMPORT File, Response, Lookup, Close, ReadChar, WriteChar;

   VAR  f,g: File;
        ch: CHAR;

BEGIN
   (* Open existing file f *)
   Lookup(f,"Example.TXT",FALSE);
   IF f.res = done THEN
      (* Create new file g *)
      Lookup(g,"Copy.BAK",TRUE);
      IF g.res = done THEN
         (* Copy f to g *)
         LOOP
            ReadChar(f,ch);
            IF f.eof THEN EXIT END;
            WriteChar(g,ch);
         END (* LOOP *);
         Close(g);
      ELSE
         (* File g could not be created (disk full?). *)
      END (* IF *);
      Close(f);
   ELSE
      (* File f could not be opened (not existent?). *)
   END (* IF *);
END FileExample.
```

7.8. FileUtil

This module provides a lot of additional useful routines operating on files.

Definition Module

```
DEFINITION MODULE FileUtil;    (* C. Vetterli, 22-May-88; A. Fischlin, 1992 *)

   FROM FileSystem IMPORT File;
   IMPORT System;

   TYPE  Path             = System.Path;
         FType            = ARRAY[0..3] OF CHAR;
         ReadProc         = PROCEDURE(VAR CHAR);
         WriteProc        = PROCEDURE(CHAR);

   VAR   termCh: CHAR; (* last entered character by given read procedure *)

   PROCEDURE ExtLookup          (VAR f: File; fname: ARRAY OF CHAR; new: BOOLEAN;
                                 VAR ok: BOOLEAN);
   PROCEDURE GetCurrentPath     (VAR curPath: Path);
   PROCEDURE AddPath            (p: Path; in: ARRAY OF CHAR; VAR out: ARRAY OF CHAR);
   PROCEDURE AppendExt          (from, ext: ARRAY OF CHAR; VAR to: ARRAY OF CHAR);

   PROCEDURE GetPos             (VAR f: File; VAR pos: LONGINT);
   PROCEDURE SetPos             (VAR f: File; pos: LONGINT);
   PROCEDURE Length             (VAR f: File; length: LONGINT);

   PROCEDURE ReadLong           (VAR f: File; VAR long: LONGINT);
   PROCEDURE WriteLong          (VAR f: File; long: LONGINT);
   PROCEDURE EOF                (VAR f: File): BOOLEAN;

   PROCEDURE GetFileName        (VAR fileName: ARRAY OF CHAR; type: ARRAY OF CHAR;
                                 VAR ok: BOOLEAN);
   PROCEDURE PutFileName        (VAR fileName: ARRAY OF CHAR; VAR ok: BOOLEAN);
   PROCEDURE ReadFileName       (VAR fileName: ARRAY OF CHAR; ext, type: ARRAY OF CHAR;
                                 Read: ReadProc; Write: WriteProc; VAR ok: BOOLEAN);
   PROCEDURE Message            (str1, str2: ARRAY OF CHAR);

   PROCEDURE SetFileType        (name: ARRAY OF CHAR; creator, type: FType);
   PROCEDURE GetDefVol          (VAR vName: ARRAY OF CHAR; VAR vRefNum: INTEGER);
   PROCEDURE SetDefVol          (vName: ARRAY OF CHAR; vRefNum: INTEGER): BOOLEAN;

   PROCEDURE Transfer           (progname: ARRAY OF CHAR);
   PROCEDURE Sublaunch          (progname: ARRAY OF CHAR);

   PROCEDURE LastIOErr          ():INTEGER;

END FileUtil.
```

Explanations

ExtLookup(f,name,new,ok);

Works the same way as 'FileSystem.Lookup' does, but if the file is not found using 'name', the procedure one by one adds all possible paths to the original file name 'name' until either the file has been found or all paths have been tried without finding the file (in this case, 'ok' is set to FALSE).

GetCurrentPath(path);
Returns the path used in the last call to 'ExtLookup'.

AddPath(path, fname, out);
This is equal to a call to 'System.AddPath'.

AppendExt(in,ext,out);
Appends the extension 'ext' to the filename 'in' and returns the resulting string in 'out'.

GetPos(f,pos); SetPos(f,pos); Length(f,l);
These procedures are the same as the corresponding procedures of module 'FileSystem', but the results are returned as LONGINTs.

ReadLong(f,long); WriteLong(f,long);
These procedures read (write) the LONGINT 'long' from (to) file 'f'.

EOF(f);
Returns 'f.eof'.

GetFileName(name,type,ok);
Displays the standard Macintosh file selector box for opening files, allowing the user to select an existing file. There are only files displayed with type 'type'. If the user quits the selector box by clicking the "Open" button, 'ok' = TRUE and 'name' contains the name of the selected file, otherwise 'ok' is set to FALSE. .

PutFileName(name,ok);
Displays the standard Macintosh file selector box for saving files. If the user quit the dialog by clicking the "Save" button, 'ok' = TRUE and 'name' contains the entered filename, else 'ok' = FALSE.

ReadFileName(name,ext,type,read,write,ok);
Reads the filename 'name' using the procedure 'read' and echoes using procedure 'write'. If the procedure reads a TAB character, it calls 'GetFileName'; if a LF is read, procedure'PutFileName' is called.

Message(str1,str2);
Displays the two strings 'str1' and 'str2' in a dialog box.

SetFileType(name,creator,type);
Sets the creator and file type of file 'name'.

GetDefVol(name,number);
Returns the name and the volume reference number of the default volume.

SetDefVol(name,number);
Sets the default volume. Use one of the following methods to define the default volume:
-'number' = 0 and 'name' = volume name with last character ":".
-'name' = empty (0C) and 'number' = volume reference number or drive number.

Transfer(progname);
Terminates the currently running program and starts the program with name 'progname'. If 'progname' is empty (i.e. progname[0] = 0C), the standard Macintosh file selector box is displayed, allowing the user to choose the program name. If the user confirms his selection by clicking the "Open" button, the transfer is done, otherwise the procedure doesn't do anything, and the program currently running is continued.

Sublaunch(progname);
Under Finder the same as Transfer. Under Multifinder or System 7 however, Sublaunch leaves the current program without terminating it and starts or resumes program 'progname'.

LastIOErr()
returns IO error code resulting from last IO-operation. noErr = 0, for other error codes see Inside Macintosh.

Imported Modules

CursorMouse
EventBase
FileSystem
System
Toolbox

Example

The example for this module is combined with the one for module 'System' (see there).

7.9. GraphicWindows

Library module for drawing graphics in a window.

Definition Module

```
DEFINITION MODULE GraphicWindows; (* E. Kohen, 26-May-86 *)

   IMPORT Windows;

   TYPE  Window        = Windows.Window;
         RestoreProc   = Windows.RestoreProc;
         Mode          = (replace, paint, invert, erase);

   VAR   Done: BOOLEAN; (* Done = "Operation was executed successfully" *)

   PROCEDURE OpenGraphicWindow     (VAR u: Window; x,y,w,h: INTEGER;
                                    name: ARRAY OF CHAR; Repaint: RestoreProc);
   PROCEDURE RedefGraphicWindow(u: Window; x,y,w,h: INTEGER);
   PROCEDURE Clear                 (u: Window);
   PROCEDURE CloseGraphicWindow(u: Window);
   PROCEDURE SetMode               (u: Window; m: Mode);

   PROCEDURE Dot                   (u: Window; x, y: INTEGER);
   PROCEDURE SetPen                (u: Window; x, y: INTEGER);
   PROCEDURE TurnTo                (u: Window; d: INTEGER);
   PROCEDURE Turn                  (u: Window; d: INTEGER);
   PROCEDURE Move                  (u: Window; n: INTEGER);
   PROCEDURE MoveTo                (u: Window; x, y: INTEGER);
   PROCEDURE Circle                (u: Window; x, y, r: INTEGER);
   PROCEDURE Area                  (u: Window; c: INTEGER; x,y,w,h: INTEGER);
   PROCEDURE CopyArea              (u: Window; sx,sy,dx,dy,dw,dh: INTEGER);

   PROCEDURE Write                 (u: Window; ch: CHAR);
   PROCEDURE WriteString           (u: Window; s: ARRAY OF CHAR);
   PROCEDURE IdentifyPos           (VAR u: Window; VAR x,y: INTEGER);

END GraphicWindows.
```

Explanations

OpenGraphicWindow(u,x,y,w,h,title,redraw);
Opens the window 'u' with lower left corner at screen coordinates (x,y), width 'w' and height 'h'. The string 'title' is drawn centered in the titlebar of the window, the window's rectangle is cleared. Then the procedure 'Repaint' is assigned to the window. This procedure is called when restoration of the window becomes necessary. If you've already opened 16 windows or if the rectangle you specified doesn't fit on the screen, the window isn't opened and 'Done' is set to FALSE.

RedefGraphicWindow(u,x,y,w,h);
Clears the window's rectangle and changes the window's size and its location on the screen. If the rectangle specified by (x,y,w,h) doesn't fit on the screen, the window isn't changed and 'Done' := FALSE.

Clear(u);
Clears the window without changing its size or location.

CloseGraphicWindow(u);
Closes the window 'u'.

SetMode(u,mode);
Sets the drawing mode to 'mode'.

Dot(u,x,y);
Displays a dot in window 'u' at the window coordinates (x,y). The dot is drawn according to the drawing mode.

SetPen(u,x,y);
Sets the pen (i.e. the current drawing position) to window coordinates (x,y).

TurnTo(u,d);
Sets the pen direction to 'd' degrees. Degrees are measured in the mathematical positive sense, i.e. anticlockwise. So 'd' = 0 means the pen is pointing to the right; 'd' = 90 makes the pen point up, etc.

Turn(u,d);
Turns the pen direction by 'd' degrees anticlockwise.

Move(u,l);
Draws a line in window 'u', starting at the pen's current location with length 'l' in the direction specified by previous calls to 'TurnTo' or 'Turn'.

MoveTo(u,x,y);
Draws a line in window 'u' from the current pen position to window coordinates (x,y).

Circle(u,x,y,r);
Draws a circle with center at window coordinates (x,y) and radius 'r' in window 'u'.

Area(u,c,x,y,w,h);
Paints a rectangular area of width 'w' and height 'h' with lower left corner at window coordinates (x,y), filled with color 'c'. The following colors may be displayed: white (c = 0), light grey (c = 1), grey (c = 2), dark grey (c = 3), black (c = 4).

CopyArea(u,sx,sy,dx,dy,w,h);
Copies the rectangular area at window coordinates (sx,sy) into the rectangle at (dx,dy). Both areas have width 'w' and height 'h' and lie in the same window 'u'.

Write(u,ch); WriteString(u,str);
Writes the character 'ch' (string 'str') in window 'u', starting at the current pen position. The pen position is changed!

IdentifyPos(u,x,y);
To given screen coordinates (x,y), the window 'u' containing these coordinates and the corresponding window coordinates are identified. The window coordinates are returned in 'x' and 'y' (these parameters are modified).

Imported Modules

MathLib
System
Toolbox
Windows

Example

```
MODULE GraphicExample; (* draws Sierpinski curves in a graphic window. *)

   FROM GraphicWindows IMPORT    Window, OpenGraphicWindow, CloseGraphicWindow, Clear,
                                 SetPen, TurnTo, Move;
   IMPORT TextWindows;

   VAR  i,x,y        : INTEGER;
        w            : Window;
        v            : TextWindows.Window;
        Length       : ARRAY [1..5] OF INTEGER;

   PROCEDURE Line (Dir, Len : INTEGER);
   BEGIN
      TurnTo(w,Dir*45);
      Move(w,Len*Length[i])
   END Line;

   PROCEDURE B (Level : INTEGER); FORWARD;
   PROCEDURE C (Level : INTEGER); FORWARD;
   PROCEDURE D (Level : INTEGER); FORWARD;

   PROCEDURE A (Level : INTEGER);
   BEGIN
      IF (Level > 0) THEN
         A(Level-1); Line(7,1); B(Level-1); Line(0,2);
         D(Level-1); Line(1,1); A(Level-1)
      END (* IF *)
   END A;

   PROCEDURE B (Level : INTEGER);
   BEGIN
      IF (Level > 0) THEN
         B(Level-1); Line(5,1); C(Level-1); Line(6,2);
         A(Level-1); Line(7,1); B(Level-1)
      END (* IF *)
   END B;

   PROCEDURE C (Level : INTEGER);
   BEGIN
      IF (Level > 0) THEN
         C(Level-1); Line(3,1); D(Level-1); Line(4,2);
         B(Level-1); Line(5,1); C(Level-1)
      END (* IF *)
   END C;

   PROCEDURE D (Level : INTEGER);
   BEGIN
      IF (Level > 0) THEN
         D(Level-1); Line(1,1); A(Level-1); Line(2,2);
         C(Level-1); Line(3,1); D(Level-1)
      END (* IF *)
   END D;
```

```
BEGIN (* GraphicExample *)
   OpenGraphicWindow(w,5,5,502,315,"Sierpinski curve",Clear);
   x := 123; y := 256;
   Length[1] := 32;
   FOR i:=2 TO 5 DO
   Length[i] := Length[i-1] DIV 2
   END (* FOR *);
   LOOP
      TextWindows.OpenTextWindow(v,312,0,200,100,"Text");
      TextWindows.WriteString(v,"Level (0 to quit) : ");
      TextWindows.ReadInt(v,i);
      TextWindows.CloseTextWindow(v);
      Clear(w);
      IF (i <= 0) OR (i > 5) THEN
          EXIT
      END (* IF *);
      SetPen(w,x,y);
      A(i); Line(7,1);
      B(i); Line(5,1);
      C(i); Line(3,1);
      D(i); Line(1,1);
   END (* LOOP *);
   CloseGraphicWindow(w)
END GraphicExample.
```

7.10. InOut

Library module for formatted input/output on terminal or files. A description of this module is also included in the Modula-2 manual [1].

Definition Module

```
DEFINITION MODULE InOut;   (* NW 20.12.82 / WH 15.07.86 / TW 23.06.88 *)

   FROM SYSTEM       IMPORT WORD;
   FROM FileSystem   IMPORT File;

   CONST EOL = 15C;

   VAR   Done       : BOOLEAN;
         termCH     : CHAR;     (* terminating character in read procedures *)
         in, out    : File;     (* for exceptional cases only *)

   PROCEDURE OpenInput          (defext: ARRAY OF CHAR);
   PROCEDURE OpenOutput         (defext: ARRAY OF CHAR);
   PROCEDURE CloseInput;
   PROCEDURE CloseOutput;

   PROCEDURE Read               (VAR ch: CHAR);
   PROCEDURE ReadString         (VAR s: ARRAY OF CHAR);
   PROCEDURE ReadInt            (VAR x: INTEGER);
   PROCEDURE ReadCard           (VAR x: CARDINAL);
   PROCEDURE ReadWrd            (VAR w: WORD);

   PROCEDURE Write              (ch: CHAR);
   PROCEDURE WriteLn;
   PROCEDURE WriteString        (s: ARRAY OF CHAR);
   PROCEDURE WriteInt           (x: INTEGER; n: CARDINAL);
   PROCEDURE WriteCard          (x,n: CARDINAL);
   PROCEDURE WriteOct           (x,n: CARDINAL);
   PROCEDURE WriteHex           (x,n: CARDINAL);
   PROCEDURE WriteWrd           (w: WORD);

   PROCEDURE ReadLongInt        (VAR x : LONGINT);
    PROCEDURE WriteLongInt      (x: LONGINT; n: CARDINAL);

   PROCEDURE ReadReal           (VAR x: REAL);
   PROCEDURE WriteReal          (x: REAL; n: CARDINAL);

   PROCEDURE ReadLongReal       (VAR x: LONGREAL);
   PROCEDURE WriteLongReal      (x: LONGREAL; n: CARDINAL);

END InOut.
```

Explanations

VAR Done : BOOLEAN;
Done indicates whether or not the least recently executed procedure terminated successfully.

Done = FALSE -->Some error occured during the execution of the last called procedure
Done = TRUE -->Don't worry, everything's fine.

OpenInput(defext);
Writes the prompt "in> " and requests a file name. If the file could be opened, subsequent input is read from this file. If the entered name ends with a period ("."), the default extension 'defext' is appended. Typing ESC quits the procedure and sets 'Done' to FALSE. 'Done' is also set to FALSE if the file could not be opened.

OpenOutput(defext);
Writes the prompt "out> " and asks for a file name. If the file could be opened, subsequent output is written to this file. Everything else works exactly as in 'OpenInput'.

CloseInput;
Closes the input file and redirects the input stream to the keyboard.

CloseOutput;
Closes the output file and redirects the output stream to the terminal window.

Read(ch);
Reads the character 'ch' from the keyboard. This procedure does not echo on the terminal window.
Done := NOT in.eof

ReadString(s);
Reads a string, i.e. a sequence of characters neither containing blanks nor control characters; leading blanks are ignored. Input is terminated by any character <= " " (blank); this character is assigned to 'termCH'.DEL is used for backspacing when the input stream currently is assigned to the keyboard.

ReadInt(i);
Reads a string, converts it to INTEGER and returns the result in 'i'. The number is converted the same way as in procedure 'Conversions.StringToInt'. If the conversion fails, 'Done' := FALSE. Leading blanks in input are ignored; input is terminated by any character ch <= " " (blank); this character is assigned to 'termCH'.

ReadCard(c);
Works exactly as 'ReadInt', but the string is converted to CARDINAL (as in procedure 'Conversions.StringToCard').

ReadWrd(w);
Reads a WORD from the file 'in'. If 'in' is open, 'Done' := NOT in.eof; if 'in' is not open, i.e. if the input stream is assigned to the keyboard, 'Done' := FALSE.

Write(ch);
Writes the character 'ch' to the current output stream. When this is assigned to the terminal window, the following characters are interpreted:

BS	10C	Sets the writing position one character backwards.
LF	12C	Sets the writing position to the same column in the next line.
FF	14C	Clears the terminal window.
CR	15C	Sets the writing position at the beginning of the current line.
DEL	177C	Sets the writing position one character backwards and erases the character.

Besides these lay-out characters, it is left undefined what happens, if non-printable ASCII characters and non ASCII characters are written out.

WriteString(s);
Writes the string 's' to the current output stream (usually, this is the terminal window). The string either fills the whole ARRAY OF CHAR or is terminated by a NUL character (0C).

WriteInt(i,n);

Converts the INTEGER 'i' to a string and then writes this string to the current output stream. The number is written right-adjusted in a field holding 'n' characters. If the field is too short to hold the number's character representation, it's filled with "#" before writing and 'Done' is set to FALSE; but there's one exception: if 'n' = 0, the number is written regardless to the field length. The conversion is done as in 'Conversions. IntToString'.

WriteCard(c,n);
Works exactly as 'WriteInt', but here a CARDINAL is converted and written.

WriteOct(c,n);
Works as 'WriteCard', but the number is converted to base 8 instead of base 10. The number's character representation is followed by "B".

WriteHex(c,n);
As 'WriteOct', but conversion to base 16 is done. The number's character representation is followed by "H".

WriteWrd(w);
Writes the WORD 'w' to file 'out'. If 'out' is not open, 'Done' is set to FALSE.

ReadLongInt(i);
Works as 'ReadInt', but the string read is converted to LONGINT (you probably imagined it : as in 'Conversions.StringToLongInt').

WriteLongInt(i,n);
Works exactly as 'WriteInt', but here a LONGINT is converted and written. Conversion is done as in 'Conversions.LongIntToString'.

ReadReal(r);
Reads a string and converts it to REAL. If the string doesn't contain the character representation of a real number, 'Done' := FALSE. Conversion is done according to the syntax given in module 'Conversions' (procedure 'StringToReal').

WriteReal(r,n);
Writes the real number 'r' right-adjusted in a field of length 'n' to the current output stream (as in 'Conversions.RealToString').

ReadLongReal(r);
Works exactly as 'ReadReal', but here the string is converted to LONRREAL. The syntax is the same, too.

WriteLongReal(r,n);
Works exactly as 'WriteReal', but here a LONGREAL is converted and written.

Imported Modules

Conversions
FileSystem
Terminal

Example

```
MODULE InOutExample;

   (* this program reads real numbers from an input file and writes these *)
   (* numbers on the screen (i.e. the "Terminal" window).                 *)

FROM InOut IMPORT  Done, OpenInput, CloseInput,
                   ReadReal, Read, WriteReal, WriteLn;

VAR  r:    REAL;
     i:    CARDINAL;
     ch:   CHAR;

BEGIN
   OpenInput("TXT");
   IF Done THEN
      i := 0;
      LOOP
         ReadReal(r);
         IF ~Done THEN
            (* end of input file reached *)
            EXIT
         ELSE
            WriteReal(r,20);
            i := (i+1) MOD 4;
            IF i=0 THEN
               WriteLn
            END (* IF *)
         END (* IF *)
      END (* LOOP *);
      CloseInput;
      Read(ch);
   ELSE
      (* Input file not opened ! *)
   END (* IF *)
END InOutExample.
```

7.11. LongMathLib, MathLib

Library modules providing some basic mathematical functions. A description of the module "MathLib" is included in the Modula-2 manual [1]. Module "LongMathLib" contains the corresponding procedures for LONGREALs and LONGINTs.

Definition Modules

```
DEFINITION MODULE MathLib (* B.Stamm, 26-May-86 *);

   PROCEDURE Sqrt          (x: REAL):     REAL;
   PROCEDURE Exp           (x: REAL):     REAL;
   PROCEDURE Ln            (x: REAL):     REAL;
   PROCEDURE Sin           (x: REAL):     REAL;
   PROCEDURE Cos           (x: REAL):     REAL;
   PROCEDURE ArcTan        (x: REAL):     REAL;
   PROCEDURE Real          (x: INTEGER):  REAL;
   PROCEDURE Entier        (x: REAL):     INTEGER;

END MathLib.


DEFINITION MODULE LongMathLib (* B.Stamm, 26-May-86 *);

   PROCEDURE LongSqrt      (x: LONGREAL):  LONGREAL;
   PROCEDURE LongExp       (x: LONGREAL):  LONGREAL;
   PROCEDURE LongLn        (x: LONGREAL):  LONGREAL;
   PROCEDURE LongSin       (x: LONGREAL):  LONGREAL;
   PROCEDURE LongCos       (x: LONGREAL):  LONGREAL;
   PROCEDURE LongArcTan    (x: LONGREAL):  LONGREAL;
   PROCEDURE LongReal      (x: LONGINT):   LONGREAL;
   PROCEDURE LongEntier    (x: LONGREAL):  LONGINT;

END LongMathLib.
```

Imported Modules

Toolbox

Release Note

With the MacMETH 3.2 Release, the file MathLib.OBM comes in 2 different versions. The name of the first version is "MathLib.OBM (**SANE**)": this is the standard implementation calling Apple's SANE package and best suited for machines **with** a Floating-Point Processor. The name of the alternate version is "MathLib.OBM (**NO SANE**)": this is an implementation especially tuned to hardware platforms **without** a Floating-Point Processor, where the (NO SANE) version is running faster than the (SANE) variant.
To activate the version of your choice, do the following procedure: open the folder M2Lib and select your version; duplicate and rename it to "MathLib.OBM": this is the relevantname requested by the system programs and the loader.

Hint: use Finder's Get Info command to identify and check the version.

7.12. Menu

Module for command selection from a menu and menu manipulation.

Definition Module

```
DEFINITION MODULE Menu;  (* C. Vetterli, 30-Jan-87/ Ch. Widmer, 5-Jun-88 *)

    TYPE  MenuRes  =  RECORD
                        menuID,
                        menuCmd: INTEGER
                      END;

    PROCEDURE SetMenu              (menuID: INTEGER; menuStr: ARRAY OF CHAR);
    PROCEDURE GetMenuCmd           (VAR cmd: MenuRes; VAR done: BOOLEAN);
    PROCEDURE InstallAbout         (aboutMsg: ARRAY OF CHAR; id: INTEGER);

    TYPE Style = (Bold, Italic, Underline, Outline, Shadow, Plain);

    PROCEDURE DeleteMenu           (menuID : INTEGER);
    PROCEDURE DisableMenu          (menuID : INTEGER);
    PROCEDURE EnableMenu           (menuID : INTEGER);
    PROCEDURE SetItem              (menuID, menuCmd : INTEGER; itemStr : ARRAY OF CHAR);
    PROCEDURE DisableItem          (menuId, menuCmd : INTEGER);
    PROCEDURE EnableItem           (menuId, menuCmd : INTEGER);
    PROCEDURE StyleItem            (menuId, menuCmd : INTEGER; style : Style);
    PROCEDURE MarkItem             (menuId, menuCmd : INTEGER; mark : BOOLEAN);
    PROCEDURE InstallAboutProc     (aboutMsg : ARRAY OF CHAR; p : PROC);

END Menu.
```

Explanations

Wherever a menu string is required as a parameter, it must have the following format:

```
menuStr  =  title {"|" item}.
title    =  {character}.
item     =  {character}.
```

The following characters are interpreted when occuring in 'item':

"/" The character 'ch' following the slash (should be a capital letter) is treated as a keyboard equivalent, i.e. the menu item may be selected by typing the option key and 'ch' at the same time.

"<" The character(s) following the "<"-sign denote the style(s) in which the menu item will be displayed. You may specify several style attributes at the same time, although in most cases this will be nonsense. The characters following the "<" have the following meaning:

"B"	-->	Bold	"O"	-->	Outlined
"I"	-->	Italic	"U"	-->	Underlined
"S"	-->	Shadowed			

"!" The character following the "!" is displayed to the left of the menu item.

"(" The menu item following the bracket will be displayed gray, i.e. it'll be entered in the menu, but it'll be disabled. A line in the menu is denoted by the item '(-' and should never be enabled.

When creating a menu, please remember the following hints about menu design:

- The appearance of a well designed menu is as simple as possible.
- A good menu should never contain more than 9 entries (5-7 is optimal).

SetMenu(menuID,str);
Installs a new menu at the end of the menu bar. 'menuID' is a user defined number. If there's already a menu installed with the same ID, it'll be substituted by the new menu specified by 'str'.

GetMenuCmd(cmd,done);
If the user made a menu selection, 'done' = TRUE and 'cmd' contains the selected menu item. 'cmd.menuID' contains the ID of the menu, 'cmd.menuCmd' the number of the selected item in the menu. The first item in a menu always has number 1.

InstallAbout(AboutItem,ResID);
Installs the string 'AboutItem' (usually of the form "About ...") in the apple menu. If this item is selected, the modal dialog with resource type "DLOG" and resource number 'ResID' is executed. (About resources see "Inside Macintosh".)

DeleteMenu(menuID);
Removes the menu with ID 'menuID' from the menu bar.

DisableMenu(menuID);
Disables the whole menu with ID 'menuID'.

EnableMenu(menuID);
Enables the menu with ID 'menuID' again.

SetItem(menuID,menuCmd,itemStr);
Changes the text of item number 'menuCmd' in menu 'menuID' to 'itemStr'. 'itemStr' shouldn't contain formatting characters (see above) as they're not interpreted but displayed, too. The string is displayed according to the style attributes set.

DisableItem(menuID,menuCmd);
Disables the item number 'menuCmd' in menu 'menuID'.

EnableItem(menuID,menuCmd);
Enables the item number 'menuCmd' in menu 'menuID' again.

StyleItem(menuId,menuCmd,style)
Sets the style attributes of item number 'menuCmd' in menu 'menuID' to 'style'. The old setting is deleted.

MarkItem(menuID,menuCmd,mark);
If 'mark' = TRUE, the item number 'menuCmd' in menu 'menuID' is checked, i.e. the check mark is displayed to the left of the item, otherwise an existing check mark is deleted.

InstallAboutProc(AboutString,p);
Installs the string 'AboutString' in the apple menu. If this item is selected, the procedure 'p' is called.

Imported Modules

Toolbox

Example

```
MODULE MenuExample; (* displays a small menu and responds to the user's actions *)

   FROM   Menu    IMPORT       MenuRes, SetMenu, GetMenuCmd, DisableMenu, EnableMenu,
                               SetItem, DisableItem, EnableItem, InstallAboutProc,
                               MarkItem;
   FROM   InOut   IMPORT       WriteString, WriteLn;

   VAR    MenuEnabled, ItemEnabled, Checked, Done   : BOOLEAN;
          Command                                   : MenuRes;

   PROCEDURE AboutThisProgram;
   BEGIN
      WriteLn;
      WriteString("*********************************");WriteLn;WriteLn;
      WriteString("  HELLO !"); WriteLn;
      WriteString('  This is an example program for module "Menu".');
      WriteString("*********************************");WriteLn;
   END AboutThisProgram;

BEGIN (* MenuExample *)
   InstallAboutProc("About MenuExample",AboutThisProgram);
   SetMenu(1,"Example|Disable menu 2|Quit/Q");
   SetMenu(2,"Menu 2|Enable  item below|(-|(Just for fun");
   MenuEnabled := TRUE; ItemEnabled := FALSE; Checked := FALSE;
   LOOP
      GetMenuCmd(Command,Done);
      IF Done THEN
         IF Command.menuID = 1 THEN
            CASE Command.menuCmd OF
              1 :  IF MenuEnabled THEN
                      DisableMenu(2);
                      SetText(1,1,"Enable  menu 2");
                   ELSE
                      EnableMenu(2);
                      SetText(1,1,"Disable menu 2");
                   END (* IF *);
                   MenuEnabled := ~MenuEnabled;
                 |2 : EXIT
            END (* CASE *);
         ELSE  (* Command.menuID = 2 *)
            CASE Command.menuCmd OF
              1 :  IF ItemEnabled THEN
                      DisableItem(2,3);
                      SetText(2,1,"Enable  item below");
                   ELSE
                      EnableItem(2,3);
                      SetText(2,1,"Disable item below");
                   END (* IF *);
                   ItemEnabled := ~ItemEnabled;
                |2 : (* Is not possible because item #2 is always disabled! *)
                |3 : Checked := ~Checked;
                   MarkItem(2,3,Checked);
            END (* CASE *);
         END (* IF Command.menuID *);
      END (* IF Done *);
   END (* LOOP *);
END MenuExample.
```

7.13. Printer

The module Printer contains procedures to write characters and screendumps to the printer.

Definition Module

```
DEFINITION MODULE Printer;   (* C .Vetterli, 18-Mar-87 *)

   FROM SYSTEM IMPORT ADDRESS;

   PROCEDURE Reset;
   PROCEDURE Write          (ch: CHAR);
   PROCEDURE WriteString    (str: ARRAY OF CHAR);
   PROCEDURE WriteLn;
   PROCEDURE OpenPrinter    (VAR ok: BOOLEAN);
   PROCEDURE ClosePrinter;
   PROCEDURE PrintScreen;
   PROCEDURE PrintWindow    (windowPtr: ADDRESS; all: BOOLEAN);

END Printer.
```

Explanations

Reset;
This procedure resets the printer.

Write(ch);
Sends the character 'ch' to the printer. The following control characters are interpreted:

CR	15C	Sets the writing position at the beginning of the next line.
LF	12C	Sets the writing position to the same column in the next line.
FF	14C	Sets the writing position to the top of the next page.
ETX	3C	'Write(ETX)' is the same as a call to 'ClosePrinter'.

WriteString(s);
Sends a whole ARRAY OF CHAR to the printer.

WriteLn;
Sets the writing position at the beginning of the next line.

OpenPrinter(ok);
Opens the connection to the printer. 'ok' is set to FALSE, if the connection could not be established; otherwise 'ok' is set to TRUE.

ClosePrinter;
Closes the connection to the printer.

PrintScreen;
Prints the whole screen, including the menu bar.

PrintWindow(ptr,all);
Prints the window designated by 'ptr'. If 'ptr' = NIL, the currently active window is printed.If 'all' is TRUE, the whole window is printed, otherwise only the window's content is printed.

Imported Modules

FileSystem
System
Toolbox

Example

```
MODULE PrintExample; (* demonstrates the access to the printer. *)

   FROM Printer IMPORT  OpenPrinter, Reset, ClosePrinter, PrintScreen,
                        PrintWindow, WriteString, WriteLn, Write;

   IMPORT InOut;

   CONST FF = 14C;  (* FF = Form feed *)

   VAR     ok : BOOLEAN;
           ch : CHAR;

BEGIN
   OpenPrinter(ok);
   IF ok THEN
      Reset;
      InOut.WriteString("This is a screendump. The whole screen is printed.");
      PrintScreen;
      InOut.WriteLn;
      InOut.WriteString("Type a key to continue : ");
      InOut.Read(ch);
      InOut.Write(FF);
      InOut.WriteString("This was a FF written to the 'Terminal' window.");
      Write(FF);
      WriteString("This was a FF written to the printer!");
      WriteLn;
      WriteString("Now the terminal window is printed.");
      WriteLn;
      InOut.WriteString("Type a key to continue : ");
      InOut.Read(ch);
      PrintWindow(NIL,TRUE);
      WriteString("End of printer demo.");
      ClosePrinter;
   ELSE
      (* Link to printer could not be established (printer off ?) *)
   END (* IF *);
END PrintExample;
```

7.14. Storage

Module Storage provides procedures for memory allocation and deallocation.

Definition Module

```
DEFINITION MODULE Storage;  (* Hansbeat Loacker, 18-Mar-87 *)

   FROM SYSTEM IMPORT ADDRESS;

   PROCEDURE ALLOCATE       (VAR adr: ADDRESS; size: INTEGER);

   PROCEDURE DEALLOCATE     (VAR adr: ADDRESS; size: INTEGER);

END Storage.
```

Explanations

ALLOCATE(adr,size);
Allocates an area of the given size and assigns its address to 'adr'. If no memory space is available, the calling program is killed.

DEALLOCATE(adr,size);
Frees the area beginning at address 'adr' with the given size.

7.15. String

This module provides routines for character string manipulations.

```
DEFINITION MODULE String;

   CONST first  =  0B;
         last                    = 77777B; (* MAX(INTEGER) *)

   PROCEDURE MakePascalString   (VAR in: ARRAY OF CHAR; VAR out: ARRAY OF CHAR);
   PROCEDURE MakeModulaString   (VAR in: ARRAY OF CHAR; VAR out: ARRAY OF CHAR);

   PROCEDURE Length             (VAR s: ARRAY OF CHAR): INTEGER;
   PROCEDURE Occurs             (VAR s: ARRAY OF CHAR; start: INTEGER;
                                 w: ARRAY OF CHAR): INTEGER;
   PROCEDURE Insert             (VAR s: ARRAY OF CHAR; at: INTEGER; w: ARRAY OF CHAR);
   PROCEDURE Append             (VAR s: ARRAY OF CHAR; w: ARRAY OF CHAR);
   PROCEDURE InsertCh           (VAR s: ARRAY OF CHAR; at: INTEGER; ch: CHAR);
   PROCEDURE AppendCh           (VAR s: ARRAY OF CHAR; ch: CHAR);
   PROCEDURE Delete             (VAR s: ARRAY OF CHAR; start, length: INTEGER);
   PROCEDURE Copy               (VAR s: ARRAY OF CHAR;  source: ARRAY OF CHAR;
                                 start, length: INTEGER);
   PROCEDURE Assign             (VAR s: ARRAY OF CHAR; source: ARRAY OF CHAR);
   PROCEDURE Same               (VAR s: ARRAY OF CHAR; start, length: INTEGER;
                                 w: ARRAY OF CHAR): BOOLEAN;
   PROCEDURE Equal              (s, w: ARRAY OF CHAR): BOOLEAN;

END String.
```

Explanations

Character strings in MacMETH are represented the following way:

```
StringType = ARRAY [0..MaxLength-1] OF CHAR;
```

Either all elements (characters) of the array are used or the string is terminated by a (single) 0C character. The first (leftmost) character has position 0, the last (rightmost) character has position length - 1.

Length(str);
Returns the number of characters in string 'str' without a trailing 0C. 'str' is a VAR-parameter for speedup reasons only and is not modified (the length of constant strings is known anyway).

Occurs(str, start, substr);
Checks if 'substr' occurs as a substring of 'str', starting at the character position 'start' or more to right. The procedure returns the index of the first character of the substring in 'str' if its found, otherwise 'last' is returned. 'str' is a VAR-parameter for speedup reasons only; it is not modified.

Insert(str, at, ins);
Inserts the string 'ins' into 'str' left to the character position 'at'. If the resulting string would be longer than the array 'str', it's clipped. If 'at' > Length(str) blanks are inserted in 'str'. If 'at' = 'last' then 'ins' is appended to 'str'.

Append(str, ins);
This equals a call 'Insert(str,last,ins)'.

InsertCh(str, at, ch)
Works exactly as 'Insert', but there's only a single character inserted. 0C is considered not to be a character.

AppendCh(str, ch);
This equals a call 'InsertCh(str,last,ch)'.

Delete(str, start, length);
Deletes a substring in 'str', starting at character position 'start' with length 'length'. If 'start' is greater than Length(str), nothing happens.

Copy(dest, source, start, length);
Copies a substring from 'source', starting at character position 'start' with length 'length' to the string 'dest'. If 'start' and 'length' denote non-existing characters (i.e. characters to the right of the last one), these non-existing characters are ignored. If the reulting string would be longer than the array 'dest', it's clipped.

Assign(dest, source);
This equals a call 'Copy(dest,source,first,last)'.

Same(str, start, length, comp);
Returns TRUE, if the substring of 'str' starting at position 'start' with length 'length' is equal to the string 'comp'. If 'start' and 'length' denote non-existing characters, these are not part of the substring to compare. All characters (including blanks) in the substring and 'comp' are compared.

Equal(str, comp);
This equals a call 'Same(str,first,last,comp)'.

The following two procedures are intended for use in connection with the Macintosh Toolbox and Operating System. The routines in the Toolbox or the Operating System all have a Lisa Pascal interface. As the string representation of Pascal slightly differs from the one used in Modula-2, it is necessary to convert strings from one to the other representation when passing strings as parameters to such routines.

MakePascalString(mstr,pstr);
Converts the Modula-2 string 'mstr' to the Pascal string 'pstr'. If the latter is too short to hold the converted string, it is truncated. 'mstr' is a VAR-parameter for speedup reasons only and is not modified.

MakeModulaString(pstr,mstr);
Converts the Pascal string 'pstr' to the Modula-2 string 'mstr'. If the latter is too short to hold the converted string, it is truncated. 'pstr' is a VAR-parameter for speedup reasons only and is not modified.

The module does not provide a procedure for lexical comparison of strings as the conventions about lexical ordering highly depend upon the language used and on the application that needs lexical comparison. So if you need it, you'll have to program it yourself.

Imported Modules

None

Example

```
MODULE StringExample; (* shows the use of the string procedures *)

   FROM  String   IMPORT last, Length, Occurs, Copy;
   FROM  InOut    IMPORT termCH, WriteString, WriteLn, ReadString;

   VAR   PathName  : ARRAY [0..63] OF CHAR;
         FileName  : ARRAY [0..31] OF CHAR;

   PROCEDURE SplitPath(VAR Path : ARRAY OF CHAR; VAR Name : ARRAY OF CHAR);
      (* splits a complete file name ('Path') into the actual path and the file name *)
      (* with the extension. The path is returned in 'Path', the file name in 'Name'.*)
      VAR i,j : INTEGER;
   BEGIN (* SplitPath *)
      (* Find last ":" in the path *)
      i := -1;
      WHILE (i < Length(Path)) DO               (* Length(s) # HIGH(s) !! *)
         j := i;
         i := Occurs(Path,i+1,":");
      END (* WHILE *);
      INC(j);
         Copy(Name,Path,j,last);
   Path[j] := 0C;                               (* Quicker than 'Delete(Path,j,last)' *)
   END SplitPath;

   BEGIN (* StringExample *)
   LOOP
      WriteLn;
      WriteString("Enter a complete file name (including the path): ");
      ReadString(PathName);
      IF termCH < " " THEN
         WriteString(" -- escape"); EXIT;
      END (* IF *);
      SplitPath(PathName,FileName);
      WriteLn;
         WriteString(" The path name is : "); WriteString(PathName); WriteLn;
   WriteString(" The file name is : "); WriteString(FileName); WriteLn;
      WriteLn;
   END (* LOOP *);
END StringExample.
```

7.16. System

The module "System" is **the heart of the Modula-2 system** on the Macintosh. It contains the loader and several utility procedures. There are also procedures for calling and terminating programs and handling the heap.

Definition Module

```
DEFINITION MODULE System;  (* H. Seiler, C. Vetterli, W. Heiz, A. Fischlin *)

   FROM SYSTEM IMPORT ADDRESS;

   TYPE    Processor = (MC68000, MC68010, MC68020, MC68030, MC68040);
   VAR     CPU : Processor;

   (* Loader *)
   CONST  MaxPrograms = 6;

   TYPE    Status =  (normal, moduleNotFound, fileNotFound, illegalKey,
                      readError, badSyntax, noMemory, alreadyLoaded, killed,
                      tooManyPrograms, continue, noApplication);

   PROCEDURE Allocate             (VAR ptr: ADDRESS; size: LONGINT);
   PROCEDURE Deallocate           (VAR ptr: ADDRESS);
   PROCEDURE Call                 (module: ARRAY OF CHAR; leaveLoaded: BOOLEAN;
                                   VAR status: Status);
   PROCEDURE Include              (module: ARRAY OF CHAR; leaveLoaded: BOOLEAN;
                                   VAR status : Status);
   PROCEDURE RemovePermModules;
   PROCEDURE Terminate            (status: Status);
   PROCEDURE DisplayStatus        (status: Status);
   PROCEDURE WarnMessage          (message: ARRAY OF CHAR);
   PROCEDURE HaltMessage          (message: ARRAY OF CHAR);
   PROCEDURE InitProcedure        (init: PROC; VAR done: BOOLEAN);
   PROCEDURE TermProcedure        (term: PROC; VAR done: BOOLEAN);

   (* HFS Support: *)
   TYPE    Path = ARRAY [0..63] OF CHAR;
   PROCEDURE FindPath             (VAR p: Path; in: ARRAY OF CHAR; VAR n: INTEGER);
   PROCEDURE AddPath              (p: Path; in: ARRAY OF CHAR; VAR out: ARRAY OF CHAR);
   PROCEDURE GetFilePath          (n: INTEGER; VAR p: Path; VAR found: BOOLEAN);
   PROCEDURE NewPath              (VAR p: Path; atEnd: BOOLEAN);
   PROCEDURE DiscardPath          (VAR path: Path);
   PROCEDURE DiscardAllPaths;
   PROCEDURE ReloadPath;

   (* Screen Support: *)
   PROCEDURE GetScreen            (VAR x, y, w, h: INTEGER);

   (* Trap Vector Table Support: *)
   PROCEDURE SuspendM2;
   PROCEDURE ResumeM2;

END System.
```

Explanations

Allocate(a, size);
Tries to allocate a memory area of the given size in the heap. If the space is not available,'a' is set to NIL, otherwise it returns the address of the reserved area.

Deallocate(a);
Releases the memory area given by address 'a'. 'a' is set to NIL.

Implementation Note:
The minimum memory area that will be reserved by 'Allocate' is size+4 bytes. Only the heap of the program that runs on the topmost level should be expanded. All other heaps must not allocate any memory through 'Allocate'.

Call(module, leaveLoaded, status);
Activates a new program that executes the body of module 'module'. The current program level is incremented by one before the execution of the program. If 'leaveLoaded' is TRUE, all modules imported directly or indirectly by the program are marked as non removable and will stay in memory for the livetime of the application, otherwise the new loaded modules are removed after program termination. After the execution of the program, 'status' indicates the termination cause.

Include(module,leaveLoaded,status);
Works as 'Call', but the current program level is not incremented. So the newly loaded and activated program in fact is a dynamical extension of the current program level. This implies that you shouldn't include a program executing the body of a main module (you'd never get back from the included module to the including one) but only programs executing the bodies of library modules.

RemovePermModules;
Remove all permanently loaded modules.

Terminate(status);
The currently running process may be terminated by a call to 'Terminate'. 'status' may signal the cause of termination.

DisplayStatus(status);
Shows the status in a dialog box.

WarnMessage(message);
Halts program execution and displays 'message' in a dialog box. The debugger may be started or the program may be continued or aborted.

HaltMessage(message);
Breaks program execution and displays 'message' in a dialog box. Similar to WarnMessage but program must be aborted.

InitProcedure(init, done);
A call to InitProcedure installs an initialization procedure 'init' which will be called whenever a program on a higher level is started. These initialization procedures are called just after the installation of the new execution level in question (i.e. after the current level is incremented) and in order of their announcements. The result parameter 'done' is set TRUE if the assignment has been successful.

TermProcedure(term, done);
A call to TermProcedure installs a termination procedure term which will be called whenever a program on a higher or same level is terminated. These termination procedures are called just before the termination of the execution level in question (i.e. before the

current level is reset) and in reverse order of their announcements. The result parameter 'done' is set TRUE if the assignment has been successful.

Implementation Note:
The total number of installed initialization and termination procedures is limited to 16 each.

FindPath(p, in, n);
The n-th path for the given file name 'in' is returned in 'p'. If the file name already contains a path, this path is returned. If there are no more paths, 'n' is set to 0.

AddPath(p, in, out);
The path 'p' and the file name 'in' are combined to the full file name 'out'.

GetFilePath(n,p,ok);
Returns in 'p' the path leading to the volume or folder or file with reference number 'n'. If 'n' is a valid number and the path could be evaluated, 'ok' = TRUE, otherwise 'ok' = FALSE.

NewPath(p, atEnd);
Add new path 'p' to the paths read from the User.Profile either at the end or if 'atEnd' = FALSE at the very beginning of the list. VAR-parameter for speedup reasons.

DiscardAllPaths;
DiscardPath(p);
Remove path 'p' from the current list, e.g. the reverse effect of NewPath. Note that any path, either read from the User.Profile or added via NewPath can be discarded. However, the actual content of the User.Profile remains unchanged. VAR-parameter for speedup.

ReloadPath;
Reloads the "PATH" section from the User.Profile.

GetScreen(x,y,w,h);
Inquires the actual size of the screen. 'x' and 'y' are always 0 (they're not interesting anyway); 'w' is the screen's width, 'h' its height.

SuspendM2;
Save the Trap Vector Table belonging to the current environment or application. This routine is typically called by the event loop responding to osEvt events, if the event is a suspend event. See Inside Macintosh, Addison Wesley, Vol. VI, pp. 5 - 15.

ResumeM2;
Resume the Trap Vector Table belonging to the calling Modula-2 program. This routine is typically called by the event loop responding to osEvt events, if the event is a resume event. For more details see Inside Macintosh, Addison Wesley, Vol. VI, pp. 5 - 15.

Imported Modules

Toolbox

Example

```
MODULE SystemExample;

   (* demonstrates the use of the procedures from module 'System' and module 'FileUtil'.      *)
   (* this program is in fact a very small shell, allowing the user to execute other programs *)
   (* or to transfer to another application.                                                   *)

   FROM System   IMPORT Call, Status;
   FROM FileUtil IMPORT GetFileName, Transfer, FType;
   FROM Menu     IMPORT MenuRes, SetMenu, GetMenuCmd;

   VAR  Command     : MenuRes;
        Done,Ok     : BOOLEAN;
        FileName    : ARRAY [0..63] OF CHAR;
        FileType    : FType;
        TermReason  : Status;

BEGIN
   SetMenu(1,"File|Execute/X|Transfer/T|Quit/Q");
   (* MacMETH sets the type of the object files it creates to 'MOBJ' *)
   FileType := "MOBJ";
   LOOP
      GetMenuCmd(Command,Done);
      IF Done & (Command.menuID = 1) THEN
         CASE Command.menuCmd OF
           1 :   GetFileName(FileName,FileType,Ok);
                 IF Ok THEN
                    Call(FileName,FALSE,TermReason);
                    IF (TermReason # normal) THEN
                       (* Some error occured *)
                    END (* IF *);
                 END (* IF *)
          |2 :   Transfer("")
          |3 :   EXIT
         END (* CASE *);
      END (* IF *)
   END (* LOOP *)
END SystemExample.
```

7.17. Terminal, TerminalIn, TerminalOut

7.17.1. TerminalIn

The module TerminalIn contains procedures to read characters from the keyboard.

Definition Module

```
DEFINITION MODULE TerminalIn;  (* W. Heiz, 12-Dec-85 / C. Vetterli 26-Feb-86 *)

   PROCEDURE BusyRead  (VAR ch: CHAR);
   PROCEDURE Read      (VAR ch: CHAR);

END TerminalIn.
```

Explanations

Read(ch)
Gets the next character from the keyboard (or from the command file) and assigns it to ch. The procedure does not "echo" the read character on the screen (i.e. on the terminal window).

BusyRead(ch)
Procedure BusyRead assigns 0C to 'ch' if no character has been typed. Otherwise BusyRead is identical to procedure 'Read'.

By typing the command key, the shift key and the key '0' (or the letter 'o') simultaneously (**Shift 0** or **Shift o**), you can reassign the input to a file (command file).

Imported Modules

EventBase
System
Toolbox

7.17.2. TerminalOut

The module TerminalOut contains procedures to write characters to a standard window called "Terminal" on the screen.

Definition Module

```
DEFINITION MODULE TerminalOut;  (* W. Heiz, 12-Dec-85 / C. Vetterli 26-Feb-86 *)

   PROCEDURE Write       (ch: CHAR);
   PROCEDURE WriteString       (string: ARRAY OF CHAR);
   PROCEDURE WriteLn;

END TerminalOut.
```

Explanations

Write(ch)
Writes the given character on the screen at the current writing position. The screen scrolls, if the writing position reaches its end. Besides the following lay-out characters, it is left undefined what happens, if non printable ASCII characters and non ASCII characters are written out.

BS	10C	Sets the writing position one character backward
LF	12C	Sets the writing position to the same column in the next line
FF	14C	Clears the terminal window
CR	15C	Sets the writing position at the beginning of the current line
DEL	177C	Sets the writing position one character backward and erases the character there

WriteString(string)
Writes the given string to the terminal window.

WriteLn
A call to procedure 'WriteLn' is equivalent to the call 'Write(CR); Write(LF)'.

Imported Modules

EventBase
System
Toolbox
(FileSystem)

Release Note

With the MacMETH 3.2 Release, the file TerminalOut.OBM comes in 2 different versions. The name of the first version is "TerminalOut.OBM (**no file outp.**)": this is the standard implementation with the simplest functionality corresponding to the description above.
The name of the alternate version is "TerminalOut.OBM (**writes file**)": when using this implementation, every character written to the window "Terminal" is also written onto the file "Terminal.OUT". Therefore, the file "Terminal.OUT" can be considered as *the real logbook of the most recent session* on the "Terminal" window.
To activate the version of your choice, do the following procedure: open the folder MBase2Lib and select your version; duplicate and rename it to "Terminal.OBM": this is the relevant name requested by the system programs and the loader.

Hint: use Finder's Get Info command to identify and check the version.

7.17.3. Terminal

The module Terminal contains all procedures of the modules TerminalIn and TerminalOut.

Definition Module

```
DEFINITION MODULE Terminal;  (* W. Heiz, 12-Dec-85 / C. Vetterli 26-Feb-86 *)

   PROCEDURE BusyRead   (VAR ch: CHAR);
   PROCEDURE Read       (VAR ch: CHAR);

   PROCEDURE Write      (ch: CHAR);
   PROCEDURE WriteString        (string: ARRAY OF CHAR);
   PROCEDURE WriteLn;

END Terminal.
```

Imported Modules

TerminalIn
TerminalOut
Toolbox

7.18. TextWindows

Library module for writing non-proportional text into a window. Text windows may be operated as scrolling displays or as forms (with positioning). A description of a similar implementation on the Lilith computer is included in report Nr. 56 of the Institut für Informatik, ETH Zürich (Author: Jürg Gutknecht, Title: "System Programming in Modula-2: Mouse and Bitmap Display").

Definition Module

```
DEFINITION MODULE TextWindows;  (* C. Vetterli, 1-Jul-86 *)

  IMPORT Windows;

  VAR  Done     : BOOLEAN;    (* Done = "previous operation was executed successfully" *)
       termCH   : CHAR;       (* termination character in all read procedures*)

     TYPE Window              = Windows.Window;
     RestoreProc              = Windows.RestoreProc;
     CloseProc                = Windows.CloseProc;
     ScrollDirection          = Windows.ScrollDirection;
     ScrollProc               = Windows.ScrollProc;

  PROCEDURE OpenTextWindow     (VAR u: Window; x,y,w,h: INTEGER; name: ARRAY OF CHAR);
  PROCEDURE RedefTextWindow    (u: Window; x,y,w,h: INTEGER);
  PROCEDURE CloseTextWindow    (u: Window);

  PROCEDURE AssignFont         (u: Window; fontNum, size: INTEGER);
  PROCEDURE TextWindowSize     (u: Window; VAR lines, col: INTEGER);
  PROCEDURE AssignRestoreProc  (u: Window; r: RestoreProc);
  PROCEDURE AssignEOWAction    (u: Window; r: RestoreProc);

  PROCEDURE ScrollUp           (u: Window);
  PROCEDURE DrawTitle          (u: Window; name: ARRAY OF CHAR);
  PROCEDURE DrawLine           (u: Window; line, col: INTEGER);
  PROCEDURE SetCaret           (u: Window; on: BOOLEAN);
  PROCEDURE Invert             (u: Window; on: BOOLEAN);
  PROCEDURE IdentifyPos        (u: Window; x, y: INTEGER; VAR line, col: INTEGER);
  PROCEDURE GetPos             (u: Window; VAR line, col: INTEGER);
  PROCEDURE SetPos             (u: Window; line, col: INTEGER);

  PROCEDURE ReadString         (u: Window; VAR a: ARRAY OF CHAR);
  PROCEDURE ReadCard           (u: Window; VAR x: CARDINAL);
  PROCEDURE ReadInt            (u: Window; VAR x: INTEGER);

  PROCEDURE Write              (u: Window; ch: CHAR);
  PROCEDURE WriteLn            (u: Window);
  PROCEDURE WriteString        (u: Window; a: ARRAY OF CHAR);
  PROCEDURE WriteCard          (u: Window; x, n: CARDINAL);
  PROCEDURE WriteInt           (u: Window; x, n: INTEGER);
  PROCEDURE WriteOct           (u: Window; x, n: CARDINAL);

  PROCEDURE AssignScrollAction (u: Window; p: ScrollProc; vert, horiz: BOOLEAN);
  PROCEDURE SetScrollBars      (u: Window; vert, horiz: INTEGER);
  PROCEDURE AssignCloseProc    (u: Window; p: CloseProc);

END TextWindows.
```

Explanations

OpenTextWindow(u,x,y,w,h,title);
Opens the window 'u' with lower left corner at screen coordinates (x,y), width 'w' and height 'h'. The string 'title' is drawn centered in the titlebar of the window, the window's rectangle is cleared. If you've already opened 16 windows or if the rectangle you specified doesn't fit on the screen, the window isn't opened and 'Done' is set to FALSE.

RedefTextWindow(u,x,y,w,h);
Changes the size and the location of window 'u'. After the call to this procedure, the lower left corner of the window will be at screen coordinates (x,y), it'll have width 'w' and height 'h'. If the rectangle specified by 'x','y','w','h' doesn't fit on the screen, the window isn't changed but the global variable 'Done' is set to FALSE.

CloseTextWindow(u);
Closes the window 'u'.

AssignFont(u,fn,size);
Assigns a non-proportional font to window 'u'. The font's number is 'fn' (see "Inside Macintosh"), it's size is 'size'.

Note
Text coordinates have their origin at the upper left corner of the window!

TextWindowSize(u,lines,cols);
Returns the number of lines and columns that can be displayed in window 'u'.

AssignRestoreProc(u,redraw)
Assigns the procedure 'redraw' to window 'u'. The procedure will be called when restauration of the window becomes necessary.

AssignEOWAction(u,eowproc);
Assigns the procedure 'eowproc' to window 'u'. This procedure will be called when the writing position reaches the end of the window (i.e. the lower right corner).

ScrollUp(u);
Scrolls the content of window 'u' one line up.

DrawTitle(u,title);
Changes the title of window 'u' to 'title'. The string is written centered in the title bar of the window.

DrawLine(u,line,col);
Draws a horizonatl or vertical line in the window 'u'.
'line' = 0 : a vertical line at position 'col' is drawn.
'col' = 0 : a horizontal line at position 'line' is drawn.
If 'line' or 'col' denote positions not within the window's boundary, or if both 'line' and 'col' are not zero, or if both are zero, 'Done' := FALSE.

SetCaret(u,on);
If 'on' = TRUE, the caret (text insertion mark, text cursor) is displayed in window 'u' at the current writing position. It'll stay visible until the next call to 'SetCaret' with 'on' = FALSE, which makes it disappear from the window.

Invert(u,on);
If 'on' = TRUE, subsequent output will be displayed in inverse mode (white on black); otherwise all output will be displayed in normal mode (black on white).

IdentifyPos(u,x,y,line,col);
To given screen coordinates (x,y), the corresponding character position within window 'u' is returned. If (x,y) denotes a location outside the window's rectangle, 'line' = 0 and 'col' = 0.

GetPos(u,line,col);
After a call to this procedure, 'line' and 'col' contain the current writing position within 'u'.

SetPos(u,line,col);
Sets the current writing position in window 'u' to line 'line' and column 'col'.

AssignScrollAction(u,scroll,v,h);
Enables the scroll bar(s) of window 'u'. If 'v' = TRUE, the vertical scroll bar will be enabled; if 'h' = TRUE, the horizontal scroll bar will be enabled; if both 'v' and 'h' are TRUE, both scroll bars will be enabled. Then, the procedure 'scroll' is assigned to window 'u'. This procedure will be called when the user changed the position of the slider (the white rectangle) in one of the installed scroll bars.

SetScrollBars(u,v,h);
Sets the position of the sliders in the installed scroll bars. The values of 'v' and 'h' have the following meaning:

v =	0	-->	The slider in the vertical scroll bar is shown at the top of the bar.
v =	999	-->	The slider in the vertical scroll bar is shown at the bottom of the bar.
h =	0	-->	The slider in the horizontal scroll bar is shown at the leftmost position of the scroll bar.
h =	999	-->	The slider in the horizontal scroll bar is shown at the rightmost position of the scroll bar.

AssignCloseProcedure(u,close);
Enables the close box of window 'u' and assign the procedure 'close' to the window. The procedure will be called when the user clicks the close box, i.e. when he wants to close the window.

ReadString(u,s); WriteCard(u,x,n); etc.
All read procedures work just the same way as those from module 'InOut', but you may not open a file to read from (input is always from keyboard). The output of the read or write procedures (e.g. the echo in a read procedure, or the output of a write procedure) always goes to the window specified by the variable 'u'.

Imported Modules

EventBase
System
Terminal
Toolbox
Windows

Example

```
MODULE TextExample; (* demonatrates the use of text windows*)

   FROM TextWindows IMPORT  Window,OpenTextWindow, CloseTextWindow, WriteLn,
                            WriteString, WriteInt, Write, IdentifyPos;
   FROM CursorMouse IMPORT  GetMouse, ML, MM;

   VAR  Buttons    : BITSET;
        x,y,col,
        line       : INTEGER;
        w          : Window;

BEGIN
   OpenTextWindow(w,5,5,502,315,"TextWindow-Demo");
   WriteString(w,"Press the mouse button!");
   WriteLn(w);
   WriteLn(w);
   LOOP
      GetMouse(Buttons,x,y);
      IF (MM IN Buttons) & (ML IN Buttons) THEN
      EXIT
         ELSIF (ML IN Buttons) THEN
         IdentifyPos(w,x,y,line,col);
         WriteString(w,"You pressed the mouse button at position (");
         WriteInt(w,x,3);
         Write(w,",");
         WriteInt(w,y,3);
         WriteString(w,") on the screen,");
         WriteLn(w);
         WriteString(w,"i.e. in line ");
         WriteInt(w,line,2);
         WriteString(w," and column ");
         WriteInt(w,col,2);
         Write(w,".");
         WriteLn(w);
         WriteLn(w);
      END (* IF *)
   END (* LOOP *);
   CloseTextWindow(w);
END TextExample.
```

7.19. Windows

Library module for handling windows on the display. A description of a similar implementation on the Lilith computer is included in report Nr. 56 of the Institut für Informatik, ETH Zürich (Author: Jürg Gutknecht, Title: "System Programming in Modula-2: Mouse and Bitmap Display").

Definition Module

```
DEFINITION MODULE Windows;  (* C. Vetterli, 1-Jul-86 *)

   CONST   Background     =   0;
           FirstWindow    =   1;
           LastWindow     =  16;

   TYPE    Window         = [Background..LastWindow];
           (* Background serves as a possible return value for the *)
           (* UpWindow procedure and is not accessible to the user *)

           RestoreProc    = PROCEDURE(Window);
           CloseProc      = PROCEDURE(Window);
           ScrollDirection = (left, right, up, down, horizontal, vertical);
           ScrollProc     = PROCEDURE(Window, ScrollDirection, INTEGER);


   PROCEDURE OpenWindow          (VAR u: Window; x, y, w, h: INTEGER;
                                  repaint: RestoreProc; VAR done: BOOLEAN);
   PROCEDURE DrawTitle           (u: Window; title: ARRAY OF CHAR);
   PROCEDURE RedefineWindow      (u: Window; x, y, w, h: INTEGER; VAR done: BOOLEAN);
   PROCEDURE CloseWindow         (u: Window);
   PROCEDURE OnTop               (u: Window): BOOLEAN;
   PROCEDURE PlaceOnTop          (u: Window);
   PROCEDURE PlaceOnBottom       (u: Window);
   PROCEDURE ScrollWindow        (u: Window; dh, dv: INTEGER);
   PROCEDURE UpWindow            (x, y: INTEGER): Window;
   PROCEDURE ActualSize          (u: Window; VAR w, h: INTEGER);
   PROCEDURE SetWindow           (u: Window);
   PROCEDURE ResetWindow;
   PROCEDURE AssignScrollAction  (u: Window; p: ScrollProc; vert, horz: BOOLEAN);
   PROCEDURE SetScrollBars       (u: Window; vert, horz: INTEGER);
   PROCEDURE AssignCloseProc     (u: Window; p: CloseProc);
   PROCEDURE GetWindowFrame      (u: Window; VAR x, y, w, h: INTEGER);

END Windows.
```

Explanations

OpenWindow(u,x,y,w,h,redraw,ok);
Opens the window 'u' with lower left corner at screen coordinates (x,y), width 'w' and height 'h'.The window is initialized by clearing its rectangle and drawing a frame with an (empty) title bar. The procedure 'redraw' will be called when restauration becomes necessary. You may have opened at most 16 windows at the same time, if you try to open a 17th, the procedure 'OpenWindow' won't open another window but will set 'ok' to FALSE.

DrawTitle(u,title);
Writes the string 'title' centered in the title bar of window 'u'.

RedefineWindow(u,x,y,w,h,ok);
Changes the size of the window and clears its rectangle. If the rectangle given by (x,y,w,h) doesn't fit on the screen, 'ok' is set to FALSE and the window is not changed.

CloseWindow(u);
Closes the window 'u'. No further operations are allowed on 'u' hereafter, except another call to 'OpenWindow'.

OnTop(u);
Returns TRUE, if the window 'u' is the uppermost, FALSE if not.

PlaceOnTop(u);
Places the window 'u' on top (in front of all other opened windows).

PlaceOnBottom(u);
Places the window 'u' on bottom (behind all other opened windows).

ScrollWindow(u,dh,dv);
Scrolls the content of window 'u' by 'dh' horizontally, and vertically by 'dv'.

UpWindow(x,y);
Returns the uppermost opened window containing (x,y), if there is any, and returns the value 'Background' if there's none.

ActualSize(u,w,h);
Returns width 'w' and height 'h' of window 'u'.

SetWindow(u);
Sets the window 'u' as the current port. See also "Inside Macintosh".

ResetWindow;
This procedure resets the current port to the window it was set to before the last call of 'SetWindow'.

AssignScrollAction(u,p,v,h);
Installs scroll bars in window 'u', enables them and also installs the scroll procedure 'p'. If v = TRUE, a vertical scroll bar is installed; if h = TRUE, a horizontal one is installed; if both 'v' and 'h' are TRUE, both scroll bars are installed. The procedure will be called after the user moved the slider (the white rectangle) in one of the scroll bars.

Imported Modules

EventBase
System
Toolbox

Example

```
MODULE WindowExample; (* general window demo (refreshing windows) *)

   FROM Windows        IMPORT Window, PlaceOnTop;
   FROM TerminalIn     IMPORT Read;
   IMPORT TextWindows;
   IMPORT GraphicWindows;

   VAR   a,b   : Window;
         Ok    : BOOLEAN;
         ch    : CHAR;

   PROCEDURE RestoreGraphic(w : Window);
   BEGIN
      GraphicWindows.Circle(w,150,150,100);
   END RestoreGraphic;

   PROCEDURE RestoreText(w : Window);
   BEGIN
      TextWindows.SetPos(w,10,2);
      TextWindows.WriteString(w,"This is an example for restoring windows.");
   END RestoreText;

BEGIN
   GraphicWindows.OpenGraphicWindow(a,0,0,300,300,"Graphic",RestoreGraphic);
   TextWindows.OpenTextWindow(b,20,20,300,300,"Text");
   TextWindows.AssignRestoreProc(b,RestoreText);
   REPEAT
      PlaceOnTop(a);
      Read(ch);
      PlaceOnTop(b);
      Read(ch)
   UNTIL (ch < " ");
   TextWindows.CloseTextWindow(b);
   GraphicWindows.CloseGraphicWindow(a);
END WindowExample.
```

8. Accessing the Macintosh Toolbox

This chapter is intended for advanced programmers only, being familiar with the 1000-page manual "Inside Macintosh" [2].

It is possible to call each Toolbox, QuickDraw, and Operating System routine. But as all these routines have a Lisa Pascal interface, there are some rules to obey:

1. Use same constant-, type-, variable-, procedure-, and function-declarations as in Lisa Pascal (but obey the Modula-2 syntax).
2. Declare parameters with size > 4 bytes as VAR parameters (the Macintosh expects an address for these parameters).
3. Use the reserved word CODE to specify the trap number. Code procedures may also be placed within definition modules.
4. Replace type CHAR by type WORD (the Macintosh uses a 2-byte representation for type CHAR, Modula-2 uses only one byte).
5. Pascal and Modula-2 strings differ in their representation. You have to do the conversion explicitly before calling a Toolbox routine (use the procedures 'MakePascalString' and 'MakeModulaString' from module 'String').
6. Pascal and Modula-2 use the same NIL representation (0D).
7. Modula-2 procedures may be passed as parameters to Toolbox routines if they save registers A2-A4/D3-D7 (see procedure 'CalledByToolbox'). The registers must be restored before each RETURN.
8. Use INLINE, SETREG, and REG to access operating system routines (see procedure 'NewPtr' and module 'InlineDemo' in the examples below). Register A5 is never touched by the compiler, registers D0 and D1 only in special cases.
9. If you access a procedure of one of the packages of the Toolbox, the Macintosh expects not only the usual parameters but also a procedure number on the stack, which means that you have to add an extra parameter to the end of the parameter list (must not be a VAR parameter!) which is used to put this procedure number on the stack.

Example

```
MODULE ToolboxDemo;  (* W. Heiz, 7-Apr-86 *)

   FROM SYSTEM IMPORT ADDRESS, BYTE, WORD, SETREG, REG, INLINE;

   TYPE    Ptr       = ADDRESS;
      Handle         = POINTER TO ADDRESS;
      Str255         = ARRAY [0..255] OF CHAR;

      (* QuickDraw types *)

      VHSelect       = (V, H);
      Point          =  RECORD
                           CASE :INTEGER OF
                                    0: v, h          : INTEGER;
                                  | 1: vh            : ARRAY [V..H] OF INTEGER;
                           END;
                        END;
      Rect           = RECORD
                        CASE :INTEGER OF
                             0: top, left, bottom, right  : INTEGER;
                           | 1: topLeft, botRight         : Point;
                        END;
                       END;
```

```
    BitMap      =   RECORD
                      baseAddr                    : Ptr;
                      rowBytes                    : INTEGER;
                      bounds   : Rect;
                    END;
    Pattern     =   ARRAY [0..7] OF BYTE;
    StyleItem   =   (bold, italic, underline, outline, shadow, condense, extend);
    Style       =   SET OF StyleItem;
    Bits16      =   ARRAY [0..15] OF CARDINAL;
    Cursor      =   RECORD
                    data, mask : Bits16;
                    hotSpot     : Point
                    END;
    GrafPtr     =   POINTER TO GrafPort;
    GrafPort    =   RECORD
                      device                      : INTEGER;
                      portBits                    : BitMap;
                      portRect                    : Rect;
                      visRgn, clipRgn             : Handle;
                      bkPat, fillPat              : Pattern;
                      pnLoc, pnSize               : Point;
                      pnMode                      : INTEGER;
                      pnPat                       : Pattern;
                      pnVis, txFont               : INTEGER;
                      txFace                      : Style;
                      txMode, txSize              : INTEGER;
                      spExtra, fgColor, bkColor   : LONGINT;
                      colrBit, patStretch         : INTEGER;
                      picSave, rgnSave, polySave  : Handle;
                      grafProcs                    : Ptr;
                    END;

(* QuickDraw routines *)

PROCEDURE GetPort       (VAR port: GrafPtr);                                  CODE 0A874H;
PROCEDURE SetPort       (newPort: GrafPtr);                                   CODE 0A873H;
PROCEDURE SetOrigin     (h, v: INTEGER);                                      CODE 0A878H;
PROCEDURE Random        (): INTEGER;                                          CODE 0A861H;
PROCEDURE SetRect       (VAR r: Rect; left, top, right, bottom: INTEGER);     CODE 0A8A7H;
PROCEDURE Pt2Rect       (pt1, pt2: Point; VAR rect: Rect);                    CODE 0A8ACH;
PROCEDURE OffsetRect    (VAR r: Rect; dh, dv: INTEGER);                       CODE 0A8A8H;
PROCEDURE InsetRect     (VAR r: Rect; dh, dv: INTEGER);                       CODE 0A8A9H;
PROCEDURE PaintRect     (VAR r: Rect);                                        CODE 0A8A2H;
PROCEDURE EraseRect     (VAR r: Rect);                                        CODE 0A8A3H;
PROCEDURE InvertRect    (VAR r: Rect);                                        CODE 0A8A4H;
PROCEDURE DrawChar      (ch: WORD);                                           CODE 0A883H;
PROCEDURE NewWindow     (wStorage: Ptr;
                         VAR boundsRect: Rect;
                         VAR title: Str255;
                         visible: BOOLEAN;
                         theProc: INTEGER;
                         behind: Ptr;
                         goAwayFlag: BOOLEAN;
                         refCon: LONGINT): Ptr;                               CODE 0A913H;
PROCEDURE DisposeWindow (theWindow: Ptr);                                     CODE 0A914H;

(* Other example procedures *)
```

```
    PROCEDURE NewPtr(byteCount: LONGINT): ADDRESS;
        CONST D0 = 0; A0 = 8;
    BEGIN
        SETREG(D0, byteCount); INLINE(0A11EH); RETURN REG(A0)
    END NewPtr;

    PROCEDURE ModulaToPascalString(m: ARRAY OF CHAR; VAR p: Str255);
        VAR i: INTEGER;
    BEGIN
        i := 0;
        WHILE (i <= HIGH(m)) & (m[i] # 0C) DO p[i+1] := m[i]; INC(i) END;
        p[0] := CHR(i);
    END ModulaToPascalString;

    PROCEDURE CalledByToolbox;
    BEGIN
        INLINE(048E7H, 01F38H);  (* MOVEM.L A2-A4/D3-D7, -(SP)  save regs *)
        (* your statements *)
        INLINE(04CDFH, 01CF8H);  (* MOVEM.L (SP)+, A2-A4/D3-D7  rest regs *)
    END CalledByToolbox;

END ToolboxDemo.



MODULE InlineDemo;  (* H. Seiler, 15-Mar-87 *)

FROM SYSTEM IMPORT ADDRESS, INLINE, REG, SETREG;

    PROCEDURE InlineDemo1(p, q: LONGINT) : LONGINT;
        (* demonstrates the access to parameters and the function result *)
        (* without explicit knowlegde of the offsets produced by the compiler. *)
    CONST D0 = 0; D1 = 1;
    BEGIN
        SETREG(D0, p);
        SETREG(D1, q);
        INLINE(4680H, 4681H, 8081H);
        RETURN REG(D0)
    END InlineDemo1;


    PROCEDURE InlineDemo2(p, q: ADDRESS) : LONGINT;
        (* copy the bytes from p^ to q^ until a zero-byte is *)
        (* found and return the number of bytes copied. *)
    CONST D0 = 0; A0 = 8; A1 = 9;
    BEGIN
        SETREG(A0, p);
        SETREG(A1, q);
        SETREG(D0, REG(A0));
        INLINE(12D8H, 66FCH);
        RETURN REG(A0) - REG(D0)
    END InlineDemo2;

END InlineDemo.
```

Appendix A

The MC68020+ Compiler

The MC68020+ Compiler

The standard MacMETH Modula-2 Compiler has been extended for the MC68020 and higher processors. Code produced by the MC68020+ Compiler will run on hardware platforms equiped with the MC68020/MC68881, the MC68030/MC68882 or the MC68040 processor. A brief description of the extensions introduced by the MC68020+ Compiler, which is called **Compile20**, is given below. You start the compiler by selecting Compile20 from the MacMETH menu. Now the Macintosh presents the following screen image:

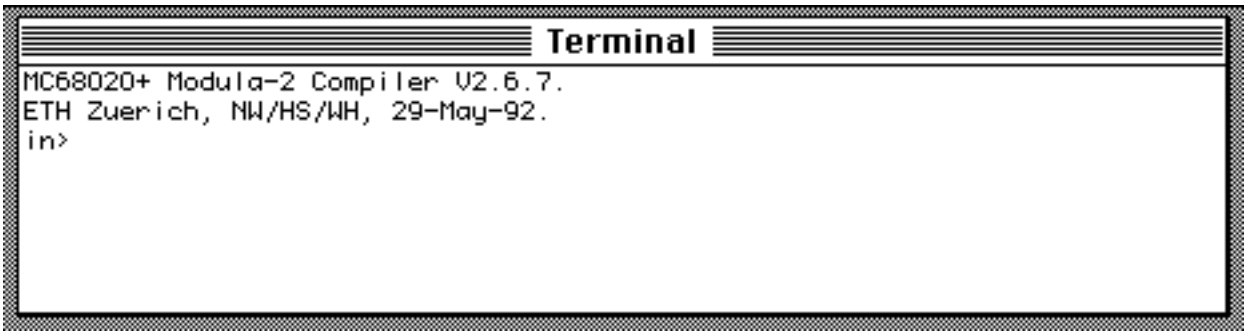

Fig. A1: Screen Image of the MC68020+ Compiler

The basic implementation

Chapter 4 also applies to the MC68020+ Compiler. The implemented language is the same as in the basic MC68000 MacMETH version: there are the same standard types, the same standard procedures and the same differences and restrictions. The sizes and alignments of data types for global data, local data, parameters and function results have **not** been changed and remain the same as in the MC68000 version of the compiler. Thus, variables intentionally are not longword-aligned. Data is still word-aligned for compatibility with already existing environments and systems, e.g. the MacMETH Library, Loader, Debugger, and the Macintosh Toolbox.

Improvements to the MC68020

The new addressing modes of the MC68020 are generated whenever a real gain in speed is achieved compared to the simple MC68000 addressing modes. The use of the scale factor in indexed addressing modes will speed up the access to array elements. For LONGINT arithmetic the long form of the multiply instruction and the long form of the divide instruction are introduced. Further improvements concern the bound checks and the exit code of procedures.

Floating-Point arithmetic with the Floating-Point Processor

The most important improvement is the cooperation with the Floating-Point Processor. The internal representation of the types REAL and LONGREAL corresponds to the IEEE standard 754 for single and double precision floating-point numbers. Thus, REAL and LONGREAL data are stored in memory as 32-bit (single precision) or 64-bit (double precision) items, respectively. These are the main floating-point formats and should satisfy the requirements of most calculations involving real numbers. Within the Floating-Point Processor, however, expressions are computed with maximum precision in the extended (80-bit) floating-point representation. The strong typing is

maintained within expressions: REAL and LONGREAL (and constants of these types) are not compatible within expressions and must not be intermixed. On the other hand it is worthwhile to notice that REAL values are assignment-compatible to the type LONGREAL.

The monadic Floating-Point Operations

Beside the standard operations of sign inversion and absolute value, supplementary monadic operators are implemented by means of pre-defined functions exported from the module SYSTEM. These mathematical functions are generic: they apply to the class of floating-point operand types. Thus, the legal arguments for these functions are either of type REAL or LONGREAL and the type of the result is the same as the type of the argument. We call the reader's attention to the fact that the procedures of the library module MathLib are perfectly replaced by generic functions with the same nomenclature.

All these monadic functions have the form

```
PROCEDURE name(x: RealType): RealType;
```

where RealType stands for either type REAL or type LONGREAL.

The following table lists the functions, their associated names exported from the module SYSTEM, and the corresponding floating-point instructions:

mathematical function	name exported from SYSTEM	floating-point instruction
sign inversion	(in standard universe)	FNEG
absolute value	(in standard universe)	FABS
arc cosine	ArcCos	FACOS
arc sine	ArcSin	FASIN
arc tangent	ArcTan	FATAN
cosine	Cos	FCOS
e**x	Exp	FETOX
ln(x)	Ln	FLOGN
log10(x)	Log	FLOG10
log2(x)	Log2	FLOG2
sine	Sin	FSIN
square root	Sqrt	FSQRT
tangent	Tan	FTAN

The dyadic Floating-Point Operations

When dyadic operators are used, both operands must be of the same type: either of type REAL or of type LONGREAL. The type of the operands defines the result type. The table below gives an overview of the dyadic operators:

mathematical function	operator	floating-point instruction
floating add	+	FADD
floating subtract	-	FSUB
floating multiply	*	FMUL
floating divide	/	FDIV
floating compare	= # > >= < <=	FCMP or FTST

Conversion procedures

While the standard procedures FLOAT and FLOATD remain unchanged, TRUNC and TRUNCD are subject to a little modification (compared to the basic MC68000 version): both TRUNC and TRUNCD now accept arguments of both types, namely REAL and LONGREAL. Two additional procedures are pre-defined and exported from the module SYSTEM: their names are Entier and Round.

Below, all possible conversion procedures are summarized:

Generics in the standard universe:

```
PROCEDURE FLOAT     (x: IntegerType)    : REAL;
PROCEDURE FLOATD    (x: IntegerType)    : LONGREAL;
PROCEDURE TRUNC     (x: RealType)       : INTEGER;
PROCEDURE TRUNCD    (x: RealType)       : LONGINT;
```

Predefined in module SYSTEM:

```
PROCEDURE Entier    (x: RealType)       : LONGINT;
PROCEDURE Round     (x: RealType)       : LONGINT;
```

An example using the monadic Floating-Point Operators

When monadic operators (e.g. the trigonometric functions) are **directly** imported from the module SYSTEM, the compiler will generate fast inline-code instructions for the Floating-Point Processor. We give an example of a user-private re-implementation of Module MathLib. The same pattern can be applied for an implementation of Module LongMathLib.

```
IMPLEMENTATION MODULE MathLib;

(* an example of an implementation of Mathlib for the Floating-Point Processor. *)

IMPORT SYSTEM;

PROCEDURE Sqrt(x: REAL) : REAL;
BEGIN RETURN SYSTEM.Sqrt(x)
END Sqrt;

PROCEDURE Exp(x: REAL) : REAL;
BEGIN RETURN SYSTEM.Exp(x)
END Exp;

PROCEDURE Ln(x: REAL) : REAL;
BEGIN RETURN SYSTEM.Ln(x)
END Ln;

PROCEDURE Sin(x: REAL) : REAL;
BEGIN RETURN SYSTEM.Sin(x)
END Sin;

PROCEDURE Cos(x: REAL) : REAL;
BEGIN RETURN SYSTEM.Cos(x)
END Cos;
```

```
PROCEDURE ArcTan(x: REAL) : REAL;
BEGIN RETURN SYSTEM.ArcTan(x)
END ArcTan;

PROCEDURE Real(x: INTEGER) : REAL;
BEGIN RETURN FLOAT(x)
END Real;

PROCEDURE Entier(x: REAL) : INTEGER;
BEGIN RETURN SHORT(SYSTEM.Entier(x))
END Entier;


END MathLib.
```

The environment for MC68020+ compiled code

The default rounding mode expected by programs compiled with the MC68020+ Compiler is Round-to-Nearest and the default rounding precision assumed is Extended. Upon initialization, MacMETH inherits the floating-point environment of the Macintosh System, which correctly presets these values in the Environment Word.

Code produced by the MC68020+ Compiler *must not* be executed on MC68000 machines or on machines without a Floating-Point Processor. Otherwise an "illegal instruction" trap or eventually an "F-line" trap will occur. During program execution, however, modules compiled with the MC68000 Compiler (Compile) may be freely mixed with modules compiled with the MC68020+ Compiler (Compile20).

An *important precondition* for the execution of code compiled with the MC68020+ Compiler is the installation of the 'FPU' traps in the configuration file (see "Traps" section in Chapter 1.4). Otherwise floating-point exceptions are likely to crash the system. If the 'FPU' traps are installed, floating-point exceptions are caught and MacMETH allows to start the debugger.

Note: be sure that 'FPU' is set to **on** in the "User.Profile" when using Compile20!

Implementation Notes

The MC68020+ Compiler is in fact a MC68040 Compiler and its output is true MC68040 code. The code meets the requirements of the Motorola 68040 chip, taking into account the RISC-design of this chip: the compiler avoids micro-coded instructions and selects only those addressing modes and instruction opcodes which are really hard-wired on the MC68040.

The benefits of this strategy are essential. As the instruction set of the MC68040 is a genuine **subset** of the instruction repertoire provided by chip-combinations like the MC68020/MC68881 and the MC68030/MC68882, the code produced by the MC68020+ Compiler is "downward" compatible and will run without problems (and even **faster**) on the MC68020/MC68881 and the MC68030/MC68882 chip-pairs.

Appendix B

The Alternate MacMETH Compiler Version 3.3

The Version 3.3 of the MacMETH Compiler is based on Modula-2 as described in N. Wirth's book 'Programming in Modula-2, **Fourth Edition**, Springer-Verlag, **1988**' [4]. The main modification concerns the data type CARDINAL: it is treated as a subrange of the type INTEGER, thereby avoiding the type incompatibilities between INTEGER and CARDINAL operands in expressions. Furthermore, the language rule about assigning a string to an array of characters is made more restrictive.

We emphasize that the Compiler Version 3.3 relies on the language presented in the **Fourth Edition** of 'Programming in Modula-2' and that the implementation is **absolutely safe** concerning the types and functions of the standard module. It is not possible for the user to generate or store illegal values, when only standard types (and their subranges) or standard procedures are involved. Finally, a few extensions and improvements were introduced, most of them leading in the direction of Oberon.

For the sake of clarity, we summarize all the predefined types, the standard functions, the module SYSTEM and the essential changes in the implementation for the **Compiler Version 3.3**:

1. Standard Types

BOOLEAN	a variable of this type assumes the truth values FALSE or TRUE. These are the only values of this type. The size of BOOLEAN is 1 byte.
CHAR	type CHAR is defined according to the ISO-ASCII standard with ordinal values from 0 to 255. The type is represented within 1 byte. The compiler accepts character constants in the range [0C .. 377C].
INTEGER	a variable of type INTEGER assumes as values the integers between -32768 and 32767. The type's size is 2.
CARDINAL	a variable of type CARDINAL assumes as values numbers in the positive integer range. **CARDINAL is pre-defined as the subrange [0..32767].**
REAL	a variable of this type assumes as values the single precision real number represented in 4 bytes. The value range expands from -3.40E38 to +3.40E38.
LONGINT	the range of type LONGINT is from -2147483648D to 2147483647D. The type's size is 4. Constants of type LONGINT must have the suffix letter 'D'.
LONGREAL	values of type LONGREAL are double precision real numbers represented in 8 bytes. The value range expands from -1.79D308 to +1.79D308. Note that constants of type LONGREAL use letter 'D' instead of 'E'!
BITSET	the type BITSET is defined as SET OF [0 .. 15]. The size of this type is 2 bytes. Note that sets obey the Motorola numbering conventions for bit data: {0} corresponds to the ordinal value 1.
PROC	the standard type PROC denotes a parameterless procedure (size is 4 bytes).

The types INTEGER, CARDINAL and LONGINT are the **integer** types. They now form a hierarchy: LONGINT includes the values of the type INTEGER, which itself includes the values of the type CARDINAL.

2. Standard Procedures

Standard procedures are predefined (need not to be imported). Some are generic procedures that cannot be explicitly declared, i.e. they apply to classes of operand types or have several possible parameter list forms. A type T is called a **scalar** type, if T is an enumeration type, CHAR, INTEGER, CARDINAL, a subrange, or type LONGINT. The following table lists all the predefined procedures (v stands for a variable, x and n for expressions, and T for a type):

ABS(x)	absolute value of x. The result type is the same as the argument type.
CAP(x)	if x is a lower case letter, the corresponding capital letter; else the same letter.
CHR(x)	the character with ordinal number x. The type of x must be an integer type.
DEC(v)	v := v - 1. The type of v must be scalar. DEC(v,n) implements v := v - n.
EXCL(v,n)	v := v - {n}. v must be a set and n compatible with the base type of the set.
FLOAT(x)	x (a value of an integer type) represented as a REAL value.
FLOATD(x)	x (a value of an integer type) represented as a value of type LONGREAL.
HALT	terminate program execution and display the standard MacMETH error box.
HIGH(v)	high index bound of array v. For an open array parameter the result type is INTEGER; otherwise the result type is equal to the index type of the array v.
INC(v)	v := v + 1. The type of v must be scalar. INC(v,n) implements v := v + n.
INCL(v,n)	v := v + {n}. v must be a set and n compatible with the base type of the set.
LONG(x)	x of INTEGER or REAL type extended to the long value of type LONGINT or LONGREAL, respective.
MAX(T)	the maximum value of type T, which must be either a scalar type or type REAL or LONGREAL.
MIN(T)	the minimum value of type T, which must be either a scalar type or type REAL or LONGREAL.
ODD(x)	returns x MOD 2 ≠ 0 (of type BOOLEAN). x must be an integer type.
ORD(x)	ordinal number of x in the set of values defined by the type T of x, which must be a scalar type. The resulting value must be a positive INTEGER and the result type is INTEGER.
SHORT(x)	x of type LONGINT or LONGREAL shortened to the corresponding short form of type INTEGER or REAL. The resulting value must be in the INTEGER or in the REAL range, respectively.
SIZE(x)	the number of bytes used in memory for x, which can be a variable or a type. The result type INTEGER.
TRUNC(x)	Real number x truncated to its integral part (of type INTEGER).
TRUNCD(x)	Long real number x truncated to its integral part (of type LONGINT).

3. The Module SYSTEM

SYSTEM Types

ADDRESS — a variable of this type holds an MC68000 address. The type´s size is 4. All operators for integer arithmetic apply to operands of this type, which is compatible with all pointer types. The arithmetic for this type is **unsigned**. Furthermore, type ADDRESS = POINTER TO BYTE. Hence, the type ADDRESS can be used to perform address computations and to export the results as pointers to values of type BYTE.

BYTE — 1 byte: the smallest addressable unit of storage; assignment-compatible with all types of size 1 byte. Note that ARRAY OF BYTE is compatible with everything.

WORD — 1 uninterpreted word: 2 consecutive bytes on an even address; assignment-compatible with all types of size 2. This type is retained for compatibility with previous versions.

LONGCARD — compatible with the type ADDRESS. This type is retained in module SYSTEM for compatibility with previous versions.

SYSTEM Procedures

(x subsequently denotes an expression of simple type with size ≤ 4 bytes)

ADR(v) — the address (of type ADDRESS) of the variable v, which may be of any type.

ASH(x,n) — x * 2**n. Arithmetic Shift x by n places. n ≥ 0 generates a left shift and n < 0 a right shift. The result type is the same as the type of x.

INLINE(n) — put n into the instruction stream. n must be a constant.

LSH(x,n) — Logical Shift of x by n places. n ≥ 0 generates a left shift and n < 0 a right shift. The result type is the same as the type of x.

MSK(x,n) — x MOD 2**n in the sense of modulo arithmetic: returns the rightmost n bits of the variable x. n must be a constant in the range [1..32]. The result type is the same as the type of x.

REG(n) — the value of the MC68000 register #n. n must be a constant in the range [0 .. 15]: the D-registers are numbered from 0 to 7 (D0 .. D7) and the A-registers from 8 to 15 (A0 .. A7). The result type is LONGINT.

ROT(x,n) — Rotate x by n places. n ≥ 0 rotates left, whereas n < 0 rotates right. The result type is the same as the type of x.

SETREG(n,x) — load x into the MC68000 register #n. If x is a simple type of size 4 bytes, the value is loaded; if x is a string or a procedure, the address is loaded. n must be a constant in the range [0 .. 15] specifying the register. D-registers are numbered from 0 to 7 (D0 .. D7) and A-registers from 8 to 15 (A0 .. A7).

TSIZE(T) — the number of bytes (of type INTEGER) used in memory for type T.

VAL(T,v) — v, of type T0, converted to a value of type T (without any security checks). VAL(T, v) is a replacement for the type transfer function T(v).

4. Implementation Notes for the Compiler Version 3.3

The class of integer types:

INTEGER, CARDINAL and LONGINT belong to the class of integer types. For these types the arithmetic is **signed**. Arithmetic overflow checks are generated whenever the corresponding compiler option is turned on. If used as arguments in standard functions, values of integer types are fully checked to be in the range of the resulting type. When integer subranges are used (CARDINAL is now an integer subrange!) the values are perfectly checked to be within the bounds of the subrange type: these checks concern the assignment as well as the standard procedures INC and DEC.

DIV and MOD for the integer types:

The implementation of DIV and MOD is changed to the subtle definition on page 142 in 'Programming in Modula-2', **Fourth Edition** [4]. For positive divisors, the result of a MOD-operation is from now always positive: 0 <= (x MOD y) < y (as in Oberon).

Assignment within the type hierarchy:

The assignment of (subtype) expressions in the direction to the supertype (LONGINT := INTEGER, LONGREAL := REAL) is fully accepted by the compiler. However, assignments in the opposite direction, where the value is possibly truncated, are no longer accepted and prohibited. A call to the standard function SHORT is now mandatory (as in Oberon).

Extensions for long types:

The standard functions CHR, DEC, INC, ODD and ORD are extended to operate on arguments of type LONGINT. For the conversion of INTEGER and REAL values into the corresponding long forms (LONGINT and LONGREAL) and vice versa, the functions LONG and SHORT are introduced as standard procedures. Furthermore, the type LONGINT is now accepted for the control-variable in a FOR-statement or the case-expression in a CASE-statement .

Strings:

The assignment a := s, where a is a character array and s a string literal, is acceptable, if the length of s is **strictly less** (rather than less or equal) to the number of elements in a. This ensures that a string is always terminated by a 0C character. The later is also true for open array parameters (ARRAY OF CHAR and ARRAY OF BYTE): the value of HIGH now points to the index-position of the 0C character (as in Oberon).

Type transfer functions and VAL:

Type transfer functions are eliminated from the basic language repertoire. They can be obtained through the use of SYSTEM.VAL. In contrast to the language report in [4], VAL is deleted from the set of standard procedures and appears with a modified semantic as member of the module SYSTEM. SYSTEM.VAL(T,x) is a replacement for the unchecked type transfer function T(x). The idea behind this change is to ensure that machine-dependent type transfers are explicitly referred to in the program and are thus readily located (as in Oberon).

The class of unsigned types:

The types BYTE, WORD and LONGCARD belong to the class of unsigned types. All these unsigned types are exported from the module SYSTEM. Note that the type LONGCARD, which is compatible with the type ADDRESS, is now a member of the module SYSTEM!

Transfer functions for the class of unsigned types:

The standard functions ORD, LONG and SHORT are extended to transfer **unsigned** typed values to (signed) integer values. Note that all these conversions are safe.

Given the declarations

```
FROM SYSTEM IMPORT BYTE, WORD, LONGCARD;
VAR b: BYTE; w: WORD; lc: LONGCARD;
```

the Compiler Version 3.3 will accept the following standard function calls:

- ORD(b) to transfer (unsigned) bytes to INTEGER's,
- LONG(b) to transfer (unsigned) bytes to values of type LONGINT,
- LONG(w) to transfer (unsigned) words to values of type LONGINT, and
- SHORT(lc) to transfer (unsigned) longwords to values of the (signed) type CARDINAL.

5. Compatibility with the standard MacMETH Compilers

The Compiler Version 3.3 is a supplement to the MacMETH System. The object code generated by the Compiler Version 3.3 is compatible with code produced by the standard Compiler Version 2.6. However, there is **one exception**: procedures having **VAR**-parameters of the 'old' CARDINAL type in their parameter list must either be adapted to the restricted range of CARDINAL or be replaced by procedures using the parameter types INTEGER or LONGINT.

Concerning the MacMETH library, the critical procedures are InOut.ReadCard, TextWindows.ReadCard, and the procedures SetPos, GetPos, Length of the module FileSystem. The basic library, however, offers a **replacement** for all these critical procedures. Instead of InOut.ReadCard, one should use InOut.**ReadInt**. Instead of TextWindows.ReadCard, use TextWindows.**ReadInt.** For the entries in module FileSystem, use the (better suited) procedures **FileUtil**.SetPos, **FileUtil**.GetPos and **FileUtil**.Length.

6. Acess to the Compilers Version 3.3

The Alter**N**ate Compilers are released on two different files suffixed by the letter '**N**' and called **CompileN** and **Compile20N**. Compile**N** (pre-linked) is the basic compiler generating native MC68000 code. Compile20**N** (also pre-linked) contains all the Version 3.3 modifications to the MC68020+ Compiler (Compile20, see Appendix A). When starting CompileN, for example, the following image is presented on the screen:

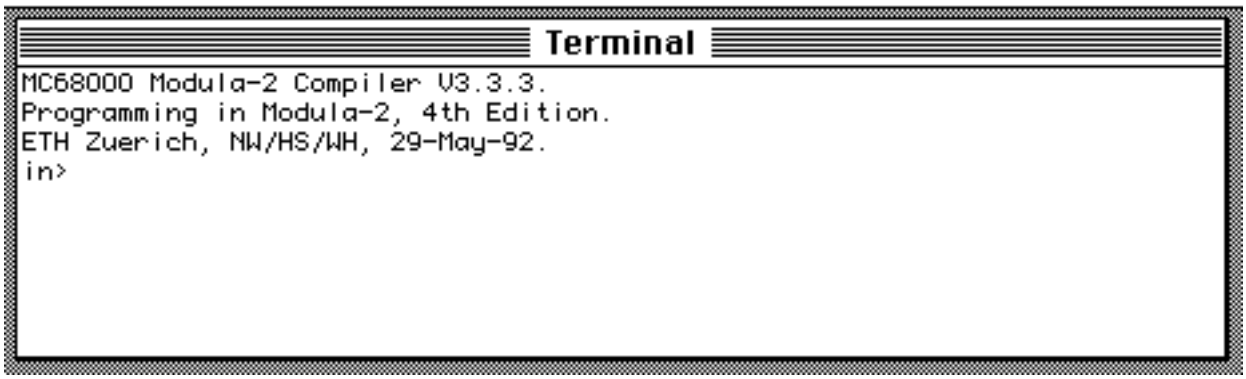

Fig. B1: Screen Image of the **Version 3.3** Compiler

Appendix C

The Syntax of the "User.Profile"

Terminal symbols are bracketted by "<" and ">". This change to standard EBNF was necessary because some terminal symbols (e.g. <"PATH">) contain the character <">.

```
User.Profile    =  PATHSection [TrapsSection] [SANESection] [MenuSection]
                   [PrinterSection] [SystemSection] [AliasSection].

PATHSection     =  <"PATH"> PathName {<,> PathName}.
PathName        =  [VolumeName] {<:> SubDirName | <:>} <:>.

TrapsSection    =  <"Traps"> [All] [Arithmetic] [FPU] [FLine] [System] [Break].
All             =  <'All'>          (<on> | <off>).
Arithmetic      =  <'Arithmetic'>   (<on> | <off>).
FPU             =  <'FPU'>          (<on> | <off>).
FLine           =  <'F-Line'>       (<on> | <off>).
System          =  <'System'>       (<on> | <off>).
Break           =  <'Break'>        (<on> | <off>).

SANESection     =  <"SANE"> [always] [invalid] [underflow] [overflow][divByZero] [inexact].
always          =  <'alwaysSANE'>          (<on> | <off>).
invalid         =  <'invalidHalt'>         (<on> | <off>).
underflow       =  <'underflowHalt'>       (<on> | <off>).
overflow        =  <'overflowHalt'>        (<on> | <off>).
divByZero       =  <'divByZeroHalt'>       (<on> | <off>).
inexact         =  <'inexactHalt'>         (<on> | <off>).

MenuSection     =  <"Menu"> MenuString | <"FullMenu"> MenuString.
MenuString      =  <'> Name {<|> Name} <'>.

PrinterSection  =  <"Printer"> [Page] [LeftMargin] [Header].
Pge             =  <'PageLen'> Number.
LeftMargin      =  <'LeftMargin'> Number.
Header          =  <'Header'> Text.

SystemSection   =  <"System"> [Compile] [Edit].
Compile         =  <'Compiler'>     (<keep> | <nokeep>).
Edit            =  <'Editor'>       (<keep> | <nokeep>).

AliasSection    =  <"Alias"> [Editor2].
Editor2         =  <'Edit2'> <is> <'> Text.<'>.

Name            =  ProgName [ </> KeybordEquiv] | Separator.
Separator       =  <(->.
KeybordEquiv    =  Letter | Digit.
Number          =  Digit {Digit}.
Digit           =  <0> | <1> | <2> | <3> | <4> | <5> | <6> | <7> | <8> | <9>.
Letter          =  <a> | <b> | <c> | ..... | <A> | <B> | <C> | .....|<Z>.
```

Any terminal symbols that aren't defined by this syntax (e.g.Text, SubDirName, ProgName) are character strings.

Appendix D

List of All Example Modules

Appendix E

Using MEdit as the Alternate Editor

What is MEdit?

MEdit 1.79 is a general purpose text editor which conforms with the Apple user interface guidelines. Its macro capability makes it especially helpful for programmers. MEdit is launched by the Edit2 command (see Chapter 1.4 "Alias" section in User.Profile and Chapter 3.7 «Starting the Alternate Editor via Edit2»).

What does MEdit?

MEdit will load by default the standard macro file "Macros" (see Fig. E1), which is in your MacMETH folder. This macro file expands the capabilities of MEdit concerning programming in Modula-2 and its integration into the MacMETH environment. The very first time you start MEdit the configuration macro will be executed. Please follow the configuration dialogue by entering your last and first name and click 'OK' for all the rest of the dialogue. If you don't complete this dialogue you are unable to switch back and forth between the MacMETH shell and MEdit.

Fig. E1: The macro file

You can execute macros via the MEdit menu Macros. Each entry in this menu corresponds to a particular macro (Fig. E2). The command "Load macro file..." in the first section remains always installed and allows to activate other sets of macros. All other sections of the menu change depending on the currently loaded macro file.

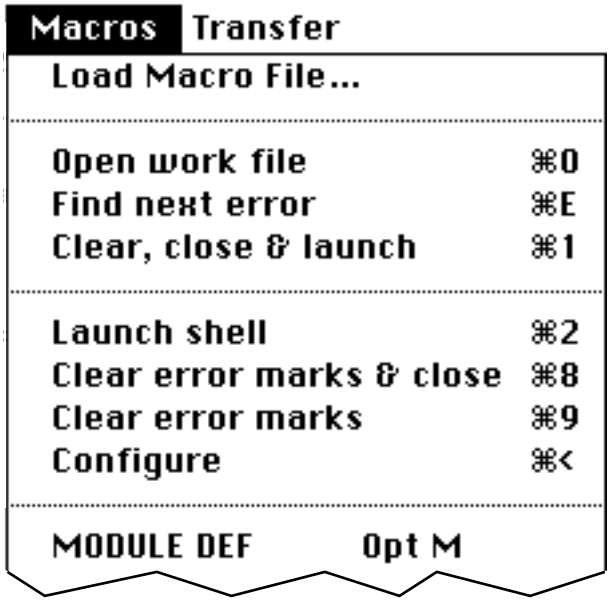

Fig E2: The macro menu of MEdit 1.79

The following text describes the functionality of the default macro file released with MacMETH 3.2 (see Fig. E2):

The most important macros for handling of work files in conjunction with MacMETH are in the second section. Use 'Open work file' (Ø Zero) to open a work file. This macro sets also the cursor to the first error (if any). 'Find next error' (E) sets the cursor to the location of the next error. If no more errors are detected, you will hear a beep and the location of the cursor will not change. 'Clear, close & launch' (1) clears all the inserted error marks, closes the work file and launches MacMETH (see Fig. E3).

In the next menu section are additional macros to handle work files and to change your initial configuration. In the fourth section there is a collection of useful macros to generate and edit Modula-2 programs. The macros in the last two sections help you to work with Modula 2 and Pascal comments, blocks and cursor positioning.

Edit/Compile Session

Typically you start an edit/compile cycle by double-clicking the MacMETH shell icon or the MEdit icon in your MacMETH folder. Use command "Compile" (C) to compile a work file (Fig. E3).

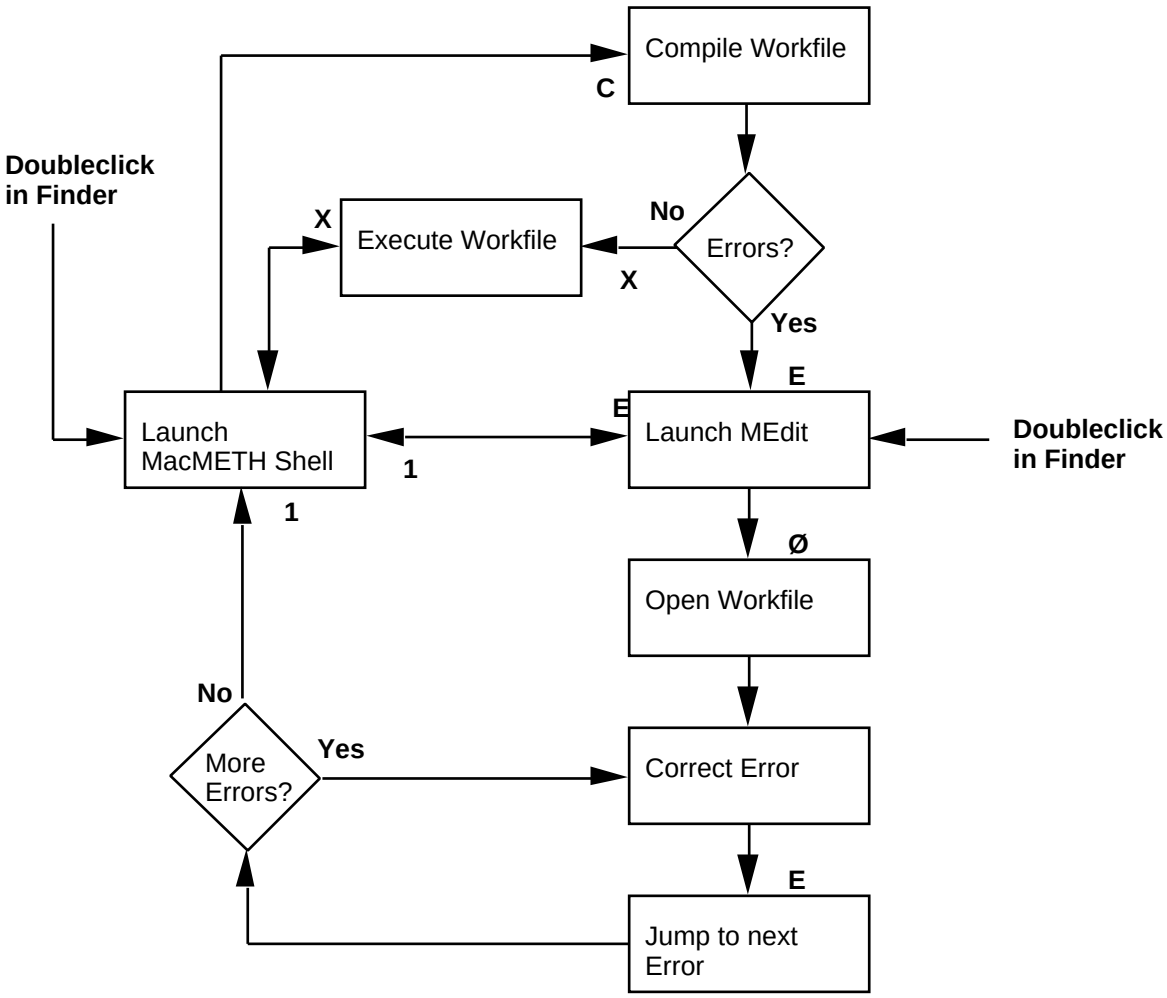

Fig. E3: Edit/Compile cycle with MEdit

If the compiler detects errors, use the command "Edit2" to switch to insert error marks into your source and switch into MEdit. If MEdit has already been opened earlier, you have first to execute the macro "Open work file" (Ø), otherwise this macro will be executed automatically. As a result

the compiled work file will be opened and the cursor is positioned to the location of the first error. Finally an explanatory error message will pop up.

After correcting the code, repeatedly select the macro "Find next error" (E) until there are no more errors. After correcting the last error, execute the macro "Clear, close & launch" (1). This macro clears any eventually remaining error marks, closes the work file and launches MacMETH. Repeat these steps summarized in Fig. E3 until the compilation of the work file is successful. For program execution select the command 'Execute' (X) in the MacMETH menu.

Hints

- A frequently encountered problem ocurrs, if you leave the work file open when leaving MEdit, e.g. by switching back to MacMETH on behalf of the Finder, and attempt to compile it. The compiler will display the following message:

  ```
  MC68000 Modula-2 Compiler V2.6.7.
  ETH Zuerich, NW/HS/WH, 29-May-92.
  in> workfile.MOD -- not found
  ```

  Since MEdit is in contrast to the "Sara" Editor (see Chapter 3.1) a completely different application, the MacMETH compiler is unable to compile a file while it remains opened by another application. Hence, as a rule, close your work file always before leaving MEdit, for instance by using only the macro command 'Clear, close & launch'.

- The compiler writes after each compilation the name of the work file into the file err.DAT. If the file err.DAT is missing or if its content does not match your current working environment, the macro "Open work file" (Ø) will display an alert. Since this macro is automatically executed, you will see this message also at every start-up of MEdit. If this bevahior disturbs you, get rid of it by compiling any existing file, which will produce a new err.DAT.

- If the compiler detects errors, it never writes the corresponding error information (which error ocurred where) into your work file, it writes it only into the file err.DAT. It is only Edit2 which reads this information from err.DAT, then it inserts the error marks into your work file before launching MEdit or whichever editor has been specified in the alias section of the User.Profile.

- The macro "Placeholder" (G) sets the cursor to the next Modula-2 comment (Form "(*. Comment .*)"). The macro "Comment" (K) and the various macros in the fourth section generate this kind of comments. If you comment statements in your work file for debugging purpose, you can easily jump to these locations, by using the "Comment" macro. It is also very useful to fill out templates generated by the macros in section four.

- You can save your current position with the macro "Save position" (B). After moving around in the document you can jump back to the saved position with the macro "Back to position" (⌥B).

- The default macro file contains macros which create whole module. E.g. execute the macro "MODULE MOD" to create a new program module, or "MODULE DEF" (⌥M) to create a new definition. The latter creates an empty definition module with a header including your name (can be set via macro "Configuration"). Once a definition module is completed, the macro "MODULE DEF->MOD" can be used to generate automatically the corresponding implementation module. For every procedure heading found in the definition module will be produced a procedure declarations with an empty body.

Index

Index of key words

Impressum of the pdf-edition:

This Portable Document File has been generated after the original publication of the MacMETH manual. Note, that in addition to this electronically distributed verision, there may be still printed manuals available. To order a printed manual, please mail to: mailto:macmeth@ito.umnw.ethz.ch.

Recommended citation:

Wirth, N., Gutknecht, J., Heiz, W., Schär, H., Seiler, H., Vetterli, C. & Fischlin, A., 1992. *MacMETH. A fast Modula-2 language system for the Apple Macintosh. User Manual. 4th, completely revised ed.* Department of Computer Sciences (ETH), Zürich, Switzerland, 116 pp.

MacMETH has now a home page:

http://www.ito.umnw.ethz.ch/SysEcol/SimSoftware/RAMSES/MacMETH.html.

From there you can also download the latest version of MacMETH and the "Manual Examples" as described in this document.

To report on bugs or for technical inquiries please write to: mailto:macmeth@ito.umnw.ethz.ch

Andreas Fischlin, ETHZ/January 5th 2003
mailto:andreas.fischlin@ito.umnw.ethz.ch