User's Guide to the MacLogimo Plus Modula-2 Development System

Version 1.2

June 1990

Project Modula
P.O. Box 2137 Provo, UT 84603- 2137

Table of Contents

Overview

MacLogimo Plus is a Modula-2 development system including a text editor, compiler, linker, debugger, library modules, application maker, disassembler, and workshop. Most of the tools are Modula-2 programs and were developed with MacLogimo Plus.

While MacLogimo Plus cannot be compared with expensive commercial development systems, substantial programs have been created with MacLogimo.

The smallest system configuration you need is a 512K Mac with an external drive. MacLogimo Plus works much better on a Macintosh with a hard disk, more RAM, and a faster CPU.

For the benefit of the academic community, MacLogimo Plus has been placed in the PUBLIC DOMAIN. It is distributed by Project Modula for a nominal materials and handling fee. In addition, MacLogimo Plus may also be distributed via "the Internet" using anonymous ftp and/or email, commercial information services, and public bulletin board systems. It may not be commercially exploited. It may not be sold, bundled with hardware or disks, nor may it be given away as an inducement to purchase any other product or services.

If you make copies of MacLogimo Plus for others, please make verbatim copies of the original disks and follow standard procedures to avoid transmission of viruses. This way, the integrity of the development system can be maintained.

This release has several features:

- It is now distributed on three 800K disks instead of five 400K disks. This reduces disk swapping and increases the space available for user programs.
- A new text editor that can display Modula-2 keywords in a bold typeface.
- A MacLogimo.LOD workshop that allows the compiler, linker, and debugger to be loaded from the menu bar. Your programs are linked to this workshop by default to speed the edit-compile-link-debug development cycle.
- The Library provides access to most functions of the Macintosh Toolbox (from *Inside Macintosh* volumes I to III) and it is in a coherent and very usable state.
- The compiler and the linker are highly reliable. Several large Modula-2 programs were successfully compiled with MacLogimo and are in extensive use on the Macintosh (e.g. a Prolog Interpreter, LAN-access programs, and MacLogimo Plus itself).
- The library supports most functions of the Toolbox and is in a coherent and very usable state. The Toolbox modules are a translation to Modula-2 of the Pascal procedures and data structures (see Inside Macintosh volumes I-IV). The module names are the same or similar as the corresponding "Managers". Volume V and VI functions are not implemented yet.
- REAL numbers are implemented, and the transcendental functions exist also. The library also contains procedures for formatted input/output from the keyboard, to the screen and to files.

The compiler, linker, and debugger themselves do not yet use HFS, however, the programs created with them certainly may. It should be noted that these tools were ported from another 68000 environment and are not meant to be good examples of Macintosh application programming.

Disk Contents

This release consists of three 800K disks: M2, M2Lib, and MacLogimo Plus Documentation.

M2

The M2 disk has the MacLogimo.LOD workshop, a text editor, the compiler, linker, debugger, and a couple of sample Modula-2 source files. If you plan to use MacLogimo Plus from floppy disks, follow the installation instructions and put a minimal System Folder on this disk. You should also put your sources and your link files on this disk. In principle, for every program or group of programs, a separate M2 disk is needed. You are recommended to plan your secondary storage in terms of boxes of disks rather than in single disks.

The M2 disk must have some free space because old versions of files are deleted only at the end of a compilation or linker run. Therefore it may be necessary to drag old LOD files manually to the Trash. Do not remove MacLogimo.LOD by accident.

M2Lib

The M2Lib diskette has four folders: SYM, LNK, REF, and Utilities. The SYM folder holds all the library symbol files needed for the compiler. The LNK folder has all the library object code files needed for the linker. The REF folder has all the library reference files needed for the debugger. The Utilities folder has an application maker, an M2App loader, and a ".LOD" file decoder (disassembler).

The ApplMaker application is used to create double-clickable applications from ".LOD" files that were linked with the "/S" stand alone option. ApplMaker uses the M2App loader file to make it happen. Source code is included.

Note: ApplMaker looks for FileType = MLOD and Creator = MLDR. MacLogimo looks for FileType = MLOD and Creator = MLPL.

MacLogimo Plus Documentation

This is a documentation diskette with the following contents:

User's Guide: this document.
DEF1 Folder: this folder contains the DEFINITION modules of a set of comprehensive library modules.
DEF2 Folder: this folder contains the DEFINITION modules of special modules, which require deep or even exotic knowledge of special Modula-2 modules and Modula/Macintosh language interface modules.
DEF3 Folder: this folder contains the DEFINITION modules of the Toolbox routines of the Macintosh ROM (see "Inside Macintosh" for documentation).
DEF4 Folder: this folder contains 3 modules, which make the toolbox programming easier, by providing standard methods for dealing with windows, desk accessories and events.
Some sample programs are also included on this disk.

Installation

The first thing you should do is lock the original MacLogimo Plus disks, make working copies of the disks, and put the original disks in a safe place as your backup.

Floppy Installation

You should then create a minimal System Folder on the M2 disk if you will be using MacLogimo Plus from only the floppy drives. Discard desk accessories, fonts, and inits you don't need for Modula-2 development. We don't supply you with a System Folder because we are not licenced to do so, and besides, you know your hardware configuration and the appropriate software better than we do.

Hard Disk Installation

If you have a hard disk, just copy everything to the hard disk, making sure the MacLogimo.LOD file, the editor, the compiler (and its support files), the linker, and the debugger files are located at the "root" or highest level of the disk. The reasons for putting all the tools at the root will be obvious if you don't.

Next, put all the SYM files in a folder called SYM, LNK files in the LNK folder, and REF files in the REF folder. Give your hard disk a short name like "M2" instead of "My big Modula-2 hard disk". You will be typing the volume name frequently, so make it easy on yourself.

Tutorial

For all volumes that use the Hierarchical File System (HFS) you must use the /L option to specify a path to the SYM, LNK, and REF files needed by the compiler, linker, and debugger. Requiring the user to enter a path is a bad user interface indeed, but remember that the tools were ported from a different operating system. A library manager is still under development to provide a base for HFS-aware development tools.

The following examples show how to use the /L option for the recommended installation configurations described above.

Floppy disk demo

Start up with the M2 disk in the internal drive and insert the M2Lib disk in the external drive. Double-click the document Hi.MOD. It appears in the editor. With the editor several documents can be opened at once, e.g. MOD, DEF and LST files.

Select "Quit" from the "File" menu to leave the editor.

Open MacLogimo.LOD by double-clicking its icon.

Select "Compile" from the "Commands" menu.

The compiler dialog begins:
```
Modula-2 Compiler
source file >
```

Type "Hi/L" (and don't forget to press the Return key). The compiler will then prompt you for libraries:
```
Libraries >
```

Type "M2Lib:SYM:". M2Lib is the disk with all the libraries and SYM is the folder name containing the needed symbol files.

Note: Disks may be ejected by typing command-shift-1 to eject the disk in the internal drive or command-shift-2 to eject the disk in the external drive. An escape character can be entered by typing a command-[.

Select "Link" from the "Commands" menu.

The linker prompts:
```
ETHZ Linker...
master file >
```

Type "Hi/L" <RETURN>.

The linker will then prompt you for libraries:
```
Libraries >
```

Type "M2Lib:LNK:". M2Lib is the disk with all the libraries and LNK is the folder name containing the needed object files. The program Hi will be linked to the base file MacLogimo.LOD and a file Hi.LOD will be produced.

Select "Execute" from the "Commands" menu. Now you can find Hi.LOD in a standard file dialog box and open it. You might have to click the "Drive" button to find it.

The program displays something and asks you to hit any key then it will promptly crash and ask you to save a core dump. Click the "Save" button. This dumps the memory to the disk, and returns you to the Finder.

Open MacLogimo.LOD again for some source-level post mortem debugging.

Select "Debug" from the "Commands" menu.

Hit <RETURN> once then type /L on the next line and enter "M2Lib:REF:" when prompted for libraries.

If the debugger requests some missing files, just type a command-[to escape and the debugger will continue. Now you are in the Debugger. Try the following input:

```
D T P M C
```

Leave the debugger by typing a "Q".

Hard Disk demo

Double check that your installation is correct with the tools, library folers, and the file Icosahedron.MOD at the root (top) level of your hard disk. Double-click the document Icosahedron.MOD. It appears in the editor.

Select "Quit" from the "File" menu to exit the editor.

Open MacLogimo.LOD by double-clicking its icon.

Select "Compile" from the "Commands" menu.

The compiler-dialog begins:

```
Modula-2 Compiler
source file >
```

Type "icosahedron/L" (and don't forget to press the Return key).

You will prompted for a library:

```
Libraries >
```

Type "M2:SYM:". (In this demo M2 is the name of the hard disk, and SYM is the folder where all the SYM files are stashed). The compiler will then compile the program.

Select "Link" from the "Commands" menu.

The linker prompts:

```
ETHZ Linker...
master file >
```

Type "icosahedron/L" <RETURN>
You will prompted for a library:

```
Libraries >
```

Type "M2:LNK:". The program will be linked to the base file MacLogimo.LOD and a file Icosahedron.LOD will be produced.

Select "Execute" from the "Commands" menu. Now you can find Icosahedron.LOD in a standard file dialog box and open it.

Glossary

The following terms will be used in the discussion of the compiler, linker, and debugger:

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 or main module.
source file:	Input file of the compiler (default extension MOD).
symbol file:	Compiler output file with symbol table generated during compilation of a definition module (default extension SYM).
reference file:	Compiler output file with debugger information generated during compilation of a program module (default extension REF).
link file:	Compiler output file with code in Modula-2 linker format (default extension LNK).
load file:	Linker output file with code in Modula-2 loader format (default extension LOD).
map file:	Linker output file with storage layout information (default extension MAP).
default extension:	When a filename doesn't contain an extension or is terminated with a period, then a default extension is appended to the name.
default libraries:	The default libraries searched are those in the directory of the source file.

Examples

The following example modules are used in this documentation. The names of the imported modules are fictional and any resemblance to the names of real library modules included in MacLogimo Plus are purely coincidental.

```
MODULE Prog1;
...
END Prog1.

MODULE Prog2;
BEGIN
 a := 2
END Prog2.

DEFINITION MODULE Prog3;
EXPORT QUALIFIED ...
...
END Prog3.


IMPLEMENTATION MODULE Prog3;
...
END Prog3.

DEFINITION MODULE Prog4;
IMPORT SystemTypes;
EXPORT QUALIFIED ...
...
END Prog4.

IMPLEMENTATION MODULE Prog4;
IMPORT SystemTypes, Loader, Exceptions;
...
END Prog4.

MODULE Prog5;
IMPORT Prog3, Prog4;
...
END Prog5.
```

The rest of this documentation may require a sound knowledge of the language itself.

Reference literature:

Wirth, Niklaus
Programming in MODULA-2
Second, Corrected Edition, Springer Verlag
ISBN 0-387-12206-0 (New York)

McCracken, Daniel D. & William I. Salmon
A Second Course in Computer Science with Modula-2
John Wiley & Sons
ISBN 0-471-63111-6

Editor

Fortunately, ModEdit is the most intuitive component of MacLogimo Plus. It is designed for editing and printing Modula-2 programs. Here are some of its features:

- Modula-2 keywords can be optionally displayed in a bold type face
- Program comments can be optionally displayed in italics
- Grep search and replace and multi-file search
- Up to four files of any size may be edited (if you have enough memory)
- Full cut, copy, paste, and undo
- Text can be scrolled both vertically and horizontally
- Blocks of text can be shifted left or right
- Printer support for ImageWriters and LaserWriters

ModEdit works best with MacLogimo Plus using MultiFinder, but if you can't use MultiFinder, just rename "ModEdit" to "Edit" and use the Transfer menu to go back and forth between MacLogimo Plus and the editor.

ModEdit reads any file of type 'TEXT' and saves files as type 'TEXT' with creator 'PMED'. Of course, you may use any text editor of your choice to create Modula-2 programs as long as the file type is 'TEXT'.

Rather than analyze the obvious usage of an editor, we simply encourage you to explore ModEdit on your own.

Compiler

Compilation of a program module

The compiler is called by selecting "Compile" from the menu. After displaying the string "source file>" the compiler is ready to accept the filename of the 'compilation unit' to be compiled.

Default extension : MOD

```
source file> PROG1<cr>  (* name PROG1.MOD is assumed*)
p1
p2      (* indicates succession of *)
p3      (* activated compiler passes *)
p4
p5
end compilation
```

If syntax errors are found in the compiled unit, compilation stops after the third pass and a listing with error messages is generated:

```
source file> PROG2<cr>
p1
---- error     (* error detected by pass 1 *)
p2
p3
---- error     (* another error detected by pass 3 *)
lister
end compilation
```

Compilation of a definition module

For definition modules the filename extension DEF is recommended. The definition part of a module must be compiled prior to its implementation part. A symbol file is generated for definition modules.

```
source file> PROG3.DEF<cr> (* definition module *)
p1
p2
symfile
lister
end compilation
```

Symbol files needed for compilation

At the compilation of a definition module, a symbol file containing symbol table information for the compiler is generated. This information is needed by the compiler in two cases:

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

According to a program option set at the beginning of the compilation (the Q option), the compiler asks for the needed symbol files or attaches them by a default name (the module name).

```
source file> PROG3<cr>  (* implementation module *)
p1
Prog3: DK:PROG3.SYM
p2
p3
p4
p5
end compilation

source file> PROG4.DEF/QUERY<cr>
p1
SystemTypes> SY:SYTY<cr>
p2
symfile
lister
end compilation

source file> PROG5/QUERY<cr>
p1
Prog3> PROG3<cr>
Prog4> PROG4<cr>
p2
p3
p4
p5
end compilation

source file> PROG6/L
library prefixes> D:,XY:
p1
.....
```

Files generated by the compiler

Several files are generated by the compiler. They all obtain the same filename as the source file with the following extensions:

for definition modules:	listing	*.LST
	symbol file	*.SYM
for program modules:	listing	*.LST
	reference file	*.REF
	code file	*.LNK

The LNK files, which seemingly are not used, must not be erased. They are used for linking secondary imports.

Program options for the compiler

When reading the source file name, the compiler also accepts some program options from keyboard. Program options are marked with a leading character '/' and must be typed sequentially after the file name.

The compiler accepts the options:

/QUERY Compiler explicitly asks for the names of the needed symbol files belonging to modules imported by the compiled unit. /Q is a short form for this option.

/NOQUERY No query for symbol file names. The files are searched corresponding to a default strategy.

/LISTING A listing file must be generated. /LIST is accepted for this option.

/NOLISTING No listing file must be generated. /N is a short form for this option.

/EL A listing will only be generated when there are compilation errors. All errors are also displayed on the terminal.

/VERSION Compiler has to display information about its version. /V is a short form for this option.

/LIBQUERY Library query: the compiler has to ask for the libraries it should search when looking for symbol files. Up to 5 user libraries may be specified, suffixed with ':' and separated by comma. /L is an abbreviation for /LIBQUERY.

/LARGE Large program option. Normally the compiler allocates tables and files for a small to medium sized program. This option forces it to allocate large tables and files. /LA is an abbreviation for /LARGE.

Defaults: NOQUERY and EL are set as default options.

Compilation options in compilation units

Comments in a Modula-2 compilation unit may be used to specify certain compilation options for tests. The following syntax is accepted for compilation options:

Options = Option { "," Option } .
Option = "$" Letter Switch .
Switch = "+" | "-" | "=" .

Options must be the first information in a comment clause. They are not recognized by the compiler if other information preceedes the options in the clause.

Options :
T Array index and case label boundary test.
S Stack overflow test.
P No generation of procedure entry and exit code.
C CLR option. C- means the compiler must not generate CLR instructions.

Switches turn the options on and off. All switches are set to "+" by default.

Switches:

+	Test code is generated.
-	No test code is generated.
=	Previous switch becomes valid again.

Example :

```
MODULE x; (* $T+ *)
...                 test code generated
 ...
 (*$T- *)
 a[i] := a[i+1];       no test code is generated
 (*$T= *)
 ...                 test code is generated
 ...
END x.
```

A remark concerning the C option:

This option is only useful for writing low-level code to directly access memory-mapped device registers (writing such code on the Macintosh guarantees compatibility problems).

Since the CLR instruction of the MC68000 does a READ/WRITE cycle, it should not be generated for absolute variables which are really device registers. The compiler doesn't know it is an absolute variable when it is passed to a procedure as an argument and may erronously generate the CLR instruction inside the procedure. With the C- option in the procedure, the compiler is instructed to generate a MOVE #0,... instead.

Example:

```
MODULE RoboHacker;

VAR
RoboServos    [00CFF1F0h]: CARDINAL;
RoboTargeting [00CFF2F0h]: CARDINAL;
RoboScanner   [00CFF3F0h]: CARDINAL;

PROCEDURE ClearRegister(VAR reg: CARDINA);
BEGIN (*$C-*)
reg:=0          (* MOVE is generated instead of CLR *)
END ClearRegister;

BEGIN
(* Shutdown Robot *)
ClearRegister( RoboTargeting );
ClearRegister( RoboScanner );
ClearRegister( RoboServos );
END RoboHacker.
```

Module key

To 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, the link file, and the load file.

For an implementation module no new key is generated. It gets the same key as the definition module. The module keys of imported modules are also recorded on the symbol files and the link files.

Any mismatch of module keys belonging to the same module will cause an error message at compilation or link time.

CAUTION: 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 directly or indirectly must be recompiled.

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

Differences and restrictions in the implementation

For the implementation of Modula-2 for MC68000-based computers some differences and restrictions must be considered.

FOR statement

The control variable must not be of type CHAR or any enumeration type (byte size) if the step is not -1 or 1. The step must not be greater than 77777B.

CASE statement

The labels of a case statement must not be greater than 77777B.

Dynamic arrays

In a procedure declaration, only the form VAR ARRAY OF <anytype> is available, but not ARRAY OF <anytype>, as opposed to the report.

For this reason, the compatibility rules between formal and actual parameters have been relaxed for open arrays:

- Strings are considered compatible with VAR ARRAY OF CHAR.
- Everything is compatible with VAR ARRAY OF WORD.

Type Transfer Functions between types of unequal size

The type transfer functions to a shorter size (e.g. CHAR(cardinal)) transfer the lower order bits of the source. Type transfers to a longer size transfer the 'value'. For unsigned source types, zeroes are filled into the higher bits (e.g. ADDRESS(char), ADDRESS(cardinal)). For a signed source type, the sign is extended (e.g. ADDRESS(integer)).
PROCESS, NEWPROCESS, TRANSFER, IOTRANSFER, LISTEN, SYSRESET, & module priorities

These language concepts for concurrent processing are NOT implemented on the Macintosh. Coroutines and multiple stacks are difficult to implement, because there is a Stack Sniffer. (Note: there is currently development and testing in this area, so stay tuned for a new SYSTEMX).

MC68000 Modula-2 compiler profile

INTEGER:	-32768..32767
CARDINAL:	0..65535
ADDRESS:	0..4294967295
REAL:	-3.4028232E38..-1.1754943E-38, 0.0, +1.175494E-38..+3.4028232E38 (IEEE 32-bit Format)
Max. Index:	-32768..65535
Max. Range:	32767
ARRAY size:	32766 max. bytes
RECORD size:	32766 max. bytes
Global Variables:	32766 max. bytes per module.

Allocation of Modula-2 types

BOOLEAN	Byte (1)
CHAR	Byte (1)
INTEGER	Word (2)
CARDINAL	Word (2)
REAL	Long (4)
WORD	Word (2)
ADDRESS	Long (4)
BITSE	Word (2)
pointer	Long (4)
enum. (<= 256 elem.)	Byte (1)
enum. (> 256 elem.)	Word (2)
set (<= 8 members)	Byte (1)
set (> 8 members)	Word (2)

Linker

Compiler code output files (called a link file, with extension LNK) must be linked before run time. One or more link files are linked to a program. The generated code file (called a load file, with extension LOD) may be loaded by the loader.

Call the linker by selecting "Link" from the menu. After displaying itsprompt, the linker is ready to accept the filename of the master file (main program).

```
master file> PROG1<cr>   (* PROG1.LNK is assumed *)
```

The linker generates two files with the same name as the master file and the following extensions:

LOD : Load file with code to be loaded.
MAP : Map with allocation addresses of linked modules (only when option M is set).

A program (overlay) may be linked to an already linked base. Information about this base is needed by the linker. To get this information, the linker has to attach the load file (extension LOD) of this base.

By default the program is linked to a base file. On the Macintosh, this file is called MacLogimo.LOD. It can reside on any mounted volume for the linker to reference, but must not be renamed. The purpose of the base file is to reduce link time and disk storage needs.

All separate modules imported by the main module (or other linked modules) must be linked to your program. If a needed module is not already linked (e.g. in the base), then the
linker requests the corresponding link file. To search this file a default strategy or a query strategy can be chosen (see program option Q). There is also an option to change the libraries for searching (see option /L).

When reading the name of the master file the linker also accepts some program options from keyboard. Program options are marked with a leading character '/' and must be typed sequentially after the file name.

The linker accepts the options:

/B:	Linker has to ask explicitely for the name of the base file.
/Q:	Linker has to ask for the names of link files belonging to imported modules. If Q is not set, then the files are searched corresponding to a default strategy.
/L:	Linker has to ask for the names of the libraries to be searched. Up to 5 user libraries may be specified, suffixed with ':' and separated by comma.
/M:	A map file must be generated.
/T+,/T-:	Writing of Linker Tables to the load file. T+ enables it. The load file can be used as a base file for further linkages. T- inhibits it (default).

/D+,/D-: Writing of Debugger Tables to the load file. D+ enables it. The Debugger can be used for this program (default). D- inhibits it.

/V: Linker has to display information about its version.

/S: Standalone load file will be generated, i.e. there is no base. When linking MacLogimo.LNK, specify /S,/T+/D- !

```
master file> PROG4/Q<cr>  (* base is the operating system *)
end linkage
   (* imported modules are already linked in the resident part *)

master file> PROG5/B/Q/M<cr>
base file> PROG4<cr>    (* DK:PROG4.LOD is accepted *)
 files linked:
Prog3> PROG3<cr>        (* DK:PROG3.LNK is accepted *)
end linkage             (* DK:PROG5.MAP is generated *)
```

Loader

A linked program is ready to be loaded and executed. There are two different ways to load a program:

• pull down the Execute command from the workshop menu and select a load file which has been linked to MacLogimo.LOD.
• in your program, use the procedure "Call" exported from the module "Loader".

Overlay organization

An overlay organization is implemented in the Modula-2 system. It is a simple instrument to overlay program parts.

An overlay part is a load file produced by the linker. It consists of one or more linked modules. An overlay part is loaded and executed when it is called from the base part to which it has been linked (the base file).

The linker permits program overlays to be stacked several levels deep. For example, the compiler modula.LOD is an overlay on top of the base file MacLogimo.LOD. Each of the compiler passes are overlays linked to modula.LOD as their base file.

Call from a program

If a program is not linked to the base layer MacLogimo.LOD, then it must be called directly from the base to which it has been linked using the procedure Loader.Call.

Debugger

If a run time error occurs and a program terminates irregularly, then -if switched on- the Modula-2 system writes the current image of the memory onto the dump file DUMP.COR. This 'post mortem dump' can be inspected with the debugger.

On the Macintosh, only parts of the memory which belong to the Modula-2 program are dumped. This makes for a short dump file.

Starting the debugger

To run the the debugger debug DEBUG from the menu in the Workshop.

The debugger first asks for two file names: the dump file and the load file of the crashed program. You may hit <RETURN> twice and accept the defaults. For the dump file, the default is DUMP.COR, and for the load file, it is the load file name of the crashed program, taken from the dump file. With the load file name, you may set the query or libquery option to steer the access to the reference and source files.

The options which may be given with the load file name are:

/Q Query option. DEBUG should ask for the names of the reference and source files (REF and MOD).

/L Library query option. DEBUG should ask for the names of libraries to be searched for REF and MOD files. This works, as described for the compiler and linker.

When no options are given, then REF and MOD files are searched automatically in the library of the load file.

Example:

```
dump file> DUMP.COR
crashed program XY:PROG1.LOD
load file> XY:PROG1.LOD
```

or

```
load file> XY:PROG1.LOD/L
libraries> A:,AB:

... generating P-chain

 Prog1.REF> XY:PROG1.REF<cr>
```

Files of the debugger

The debugger needs the following files:
• the dump file, DUMP.COR.
• the load file of the crashed program
• the .REF and the .MOD files of the modules to be debugged.

Windows of the debugger

The debugger shows the crashed program through five windows (one at a time):

P)rocess window:
Shows the active procedures at the moment of the dump. Also, the procedure chains of other PROCESSes may be shown.

M)odule window:
Shows the list of all linked modules. After choosing a module, the T and D window will display data of the selected module rather than the procedure selected in the P window.

D)ata window:
Shows the value of the variables of the active procedures and the value of the global variables of the modules. Even local modules, structured variables and pointer chains may be inspected.

T)ext window:
Shows the source file, located at the point in error. For the T window, the debugger needs the source file and the reference file, which are searched or asked for (depending on options).
Example:
PROG1.MOD> XY:PROG1.MOD <cr>

C)ore window:
Shows an absolute memory dump.

Commands of the debugger

window switching commands:
"P": switch to P window
"M": switch to M window
"D": switch to D window
"T": switch to T window
"C": switch to C window
"Q": quit from debugger

window specific commands:

P window:
"@": re-generate the P-chain of the process which was running when the program was aborted

M window: none

D window:

"S": go to the son level (down)
"F": go back to the father level (up)
"V": go back to the last level with variables
"A": show address of the currently selected data element
"@": generate the P-chain of the currently selected variable or field of type PROCESS or POINTER TO PROCESS, or of a PROCESS variable specified by its address and switch to the P window

T window: none

C window:

"A": ask for a new address and show its environment in the core
"@": take the currently selected word as a new address (indirect addressing) changing the data representation type:
"#C": CHAR
"#B": byte (octal)
"#W": WORD (octal, default)
"#I": INTEGER
"#U": CARDINAL (unsigned)
"#R": REAL

general commands (legal in all windows):

change position:
"+": increment position
"-": decrement position
"^": increment position by half a screen length
"_": decrement position by half a screen length
number: set a new window position (line number)

Library Searching Strategies

The following strategies are applied by the compiler and linker to search the needed symbol or link files. The default strategy or the query strategy can be chosen. Both strategies refer by default to the default libraries (source directory, then any drive in descending 'drive number' order). A default name is generated from the module name.

Default Strategy

All needed files are searched by the default name. A file is searched in the same directory as the source/master file first. If this search fails, then the file is searched on the devices/paths specified by the /L option.

For each file a message is displayed, where it was found or that it was not found. In the latter case the user has the opportunity to type another file name.

The libraries to be searched by the default strategy may be changed for the compiler and the linker by the /L options.

Query Strategy

For each searched file the user has to type a file name. If the search fails, then an error message is displayed and another file name must be typed. If only <cr> is typed, then the file is searched as in the default strategy.

Query for a file name is repeated until the file is found or <esc> is typed. Typing <esc> means, that no file is supplied. On the Macintosh, <esc> is COMMAND-[.

The Module SYSTEM

The module SYSTEM offers some programming elements of Modula-2 that are implementation dependent and/or refer to the given processor. Such kind of elements are sometimes necessary for the so called "low level programming". Programs that depend upon these elements usually are not portable. SYSTEM also contains types and procedures which allow a very basic coroutine handling.

The module SYSTEM is directly known to the compiler, because its exported objects obey special rules, that must be checked by the compiler. If a compilation unit imports objects from module SYSTEM, then no symbol file must be supplied for this module.

Do not confuse the built-in module SYSTEM with the module SYSTEMX. SYSTEMX is private to the compiler and is referenced by the linker, BUT SYSTEMX IS NOT FOR USE BY USER PROGRAMS. Eventually, the SYSTEMX module will be integrated into the compiler's code generator and the mystery and confusion surrounding SYSTEMX will cease.

Here are the objects exported from module SYSTEM for this generic 68000 Version:
(Note: several objects are not yet implemented for the Macintosh!)

Types

WORD

Representation of an individually accessible storage unit (one word = 16 bits on the MC68000). No operations are allowed for variables of type WORD, except assignment. A WORD parameter may be substituted by an actual parameter of any type that uses one word in storage. If the parameter is a value parameter, then values of types using one byte are allowed too for substitution. The same holds for the substitution of a dynamic ARRAY OF WORD parameter by a value of a type with an odd size.

ADDRESS

Byte address of any location in the storage. The type ADDRESS is compatible with all pointer types and is itself defined as POINTER TO WORD. All cardinal arithmetic operators apply to this type, as well as INC and DEC. ADDRESS may be used in the sense of "long cardinal (32bit)". The type ADDRESS is not compatible with the type CARDINAL. Type transfer functions (zero-filling resp. truncating) have to be used in mixed arithmetic expressions. Under the $T- compiler option, INC and DEC are allowed for all pointer variables, not only for ADDRESS.

PROCESS

Type used for process handling. (points to the (moving) top of the process stack)

Procedures

NEWPROCESS(p:PROC; a: ADDRESS; n: ADDRESS; VAR p1: PROCESS; ip: ADDRESS)

Procedure to instantiate a new process. At least 300 words are needed for the workspace of a process. ip is the initial priority (optional, may be omitted; default is zero).
NOT IMPLEMENTED FOR MACINTOSH.

TRANSFER(VAR p1, p2: PROCESS)

Transfer of control between two processes.
NOT IMPLEMENTED FOR MACINTOSH.

IOTRANSFER(VAR p1, p2: PROCESS; va: ADDRESS)

Transfer procedure for processes operating on a peripheral device. va is an interrupt vector address (not the vector number).
NOT IMPLEMENTED FOR MACINTOSH.

LISTEN

Procedure to allow an interrupt. The priority is set to 0 and afterwards back to the current priority.
NOT IMPLEMENTED FOR MACINTOSH.

SYSRESET

Procedure to initialize the system.
NOT IMPLEMENTED FOR MACINTOSH.

CODE(c: CARDINAL)

Procedure to allow the use of machine code. c must be constant.

SETREG (r: CARDINAL; v: ....);

Procedure to set a register to a 32-bit value. r = constant (0..7=D0..D7, 8..15=A0..A7). v may be a variable or constant (the value is put into the register) or a procedure (the entry address is put into the register).

Functions

ADR(variable): ADDRESS

Returns the storage address of the substituted variable.

SIZE(variable): CARDINAL

Returns the number of bytes used by the substituted variable in the storage. If the variable is of a record type with variants, then the variant with maximal size is assumed. If variable is not a dynamic array, then SIZE is compatible to CARDINAL, INTEGER and ADDRESS (like a numerical constant). If variable is a dynamic array, then SIZE is compatible to CARDINAL only (like a CARDINAL variable).

TSIZE(type): CARDINAL
TSIZE(type, tag1const, tag2const, ...): CARDINAL

Returns the number of bytes used by a variable of the substituted type in the storage. If the type is a record with variants, then tag constants of the last FieldList (see Modula-2 syntax) may be substituted in their nesting order. If none or not all of the tag constants are specified, then the remaining variant with maximal size is assumed. TSIZE is compatible to CARDINAL, INTEGER and ADDRESS (like a numerical constant).

REGISTER(num: CARDINAL): ADDRESS

Returns the contents of the specified register. num must be a constant: 0..7=D0..D7, 8..15=A0..A7. Register A5 points to the process descriptor record, A6 to the data of the current procedure (dynamic link), and A7 to the top of stack.

ASH (x, k): <TYPE OF x>

Shift x arithmetically by k bits. x may be CARDINAL, INTEGER, ADDRESS or a set. ASH assumes the type of x. k may be INTEGER or CARDINAL. k>0 shifts left, k<0 shifts right. k is only evaluated MOD 64.

BIT(i: CARDINAL OR INTEGER): BITSET;

Dynamic bit set function for {i}. i may be variable or constant (whereas in {i} of standard MODULA, i must be a constant; only a procedure is provided for variable i (INCL)).

Workfiles

The compiler and the linker require workfiles which are always allocated in RAM for improved performance. However, certain compiler options or low memory conditions may cause the workfiles to be written to disk. The MacLogimo Plus Modula-2 runtime system creates a memory dump file which is needed for debugging. It is possible you may be fortunate enough to never see one of these files, but if you aren't so lucky, you may have to restart your Macintosh before you can delete them and reclaim the disk space.

Here are the workfile names and maximum sizes:

IL1.WORK	70 KB
IL2.WORK	50 KB
ASCII.WORK	9 KB
LINKER.WORK	10 KB
DEBUG.COR	used memory size (data only) + 1 KB

Compiler Error Messages

0 : illegal character
1 : eof in E+ option
2 : constant out of range
3 : open comment at end of file
4 : string terminator not in this line
5 : too many errors
6 : string too long
7 : too many identifiers (identifier table full)
8 : too many identifiers (hash table full)
20 : identifier expected
21 : INTEGER constant expected
22 : ']' expected
23 : ';' expected
24 : block name at the END does not match
25 : error in block
26 : ':=' expected
27 : error in expression
28 : THEN expected
29 : error in LOOP statement
30 : constant must not be CARDINAL
31 : error in REPEAT statement
32 : UNTIL expected
33 : error in WHILE statement
34 : DO expected
35 : error in CASE statement
36 : OF expected
37 : ':' expected
38 : BEGIN expected
39 : error in WITH statement
40 : END expected
41 : ')' expected
42 : error in constant
43 : '=' expected
44 : error in TYPE declaration
45 : '(' expected
46 : MODULE expected
47 : QUALIFIED expected
48 : error in factor
49 : error in simple type
50 : ',' expected
51 : error in formal type
52 : error in statement sequence
53 : '.' expected
54 : export at global level not allowed
55 : body in definition module not allowed
56 : TO expected
57 : nested module in definition module not allowed
58 : '}' expected
59 : '..' expected

60 : error in FOR statement
61 : IMPORT expected
70 : identifier specified twice in importlist
71 : identifier not exported from qualifying module
72 : identifier declared twice
73 : identifier not declared
74 : type not declared
75 : identifier already declared in module environment
76 : dynamic array must not be value parameter
77 : too many nesting levels
78 : value of absolute address must be of type CARDINAL
79 : scope table overflow in compiler
80 : illegal priority
81 : definition module belonging to implementation module not found
82 : structure not allowed for implementation or hidden type
83 : procedure implementation different from definition
84 : not all defined procedures or hidden types implemented
86 : incompatible versions of symbolic modules
88 : function type is not scalar or basic type
90 : pointer-referenced type not declared
91 : tagfieldtype expected
92 : incompatible type of variant-constant
93 : constant used twice
94 : arithmetic error in evaluation of constant expression
95 : range not correct
96 : range only with scalar types
97 : type-incompatible constructor element
98 : element value out of bounds
99 : set-type identifier expected
101 : undeclared identifier in export-list of the module
103 : wrong class of identifier
104 : no such module name found
105 : module name expected
106 : scalar type expected
107 : set too large
108 : type must not be INTEGER or CARDINAL or ADDRESS
109 : scalar or subrange type expected
110 : variant value out of bounds
111 : illegal export from program module
120 : incompatible types in conversion
121 : this type is not expected
122 : variable expected
123 : incorrect constant
124 : no procedure found for substitution
125 : unsatisfying parameters of substituted procedure
126 : set constant out of range
127 : error in standard procedure parameters
128 : type incompatibility
129 : type identifier expected
130 : type impossible to index

131 : field not belonging to a record variable
132 : too many parameters
134 : reference not to a variable
135 : illegal parameter substitution
136 : constant expected
137 : expected parameters
138 : BOOLEAN type expected
139 : scalar types expected
140 : operation with incompatible type
141 : only global procedure or function allowed in expression
142 : incompatible element type
143 : type incompatible operands
144 : no selectors allowed for procedures
145 : only function call allowed in expression
146 : arrow not belonging to a pointer variable
147 : standard function or procedure must not be assigned
148 : constant not allowed as variant
149 : SET type expected
150 : illegal substitution to WORD parameter
151 : EXIT only in LOOP
152 : RETURN only in procedure or function
153 : expression expected
154 : expression not allowed
155 : type of function expected
156 : INTEGER constant expected
157 : procedure call expected (or ':=' misspelled)
158 : identifier not exported from qualifying module
161 : call of procedure with lower priority not allowed
198 : CARDINAL constant expected
199 : BITSET type expected
200 : size of structured type too large for this processor
201 : array index too large for this element type
202 : array element size too large for this processor
203 : array index type too large for this processor
204 : subrange too large for this processor
206 : illegal subrange type
207 : case label range too large, use IF statement
208 : global data too large for this processor
209 : local data too large for this processor
210 : parameter data too large for this processor
211 : offset of record field too large for this processor
225 : too many unnamed types (typetable full)
226 : reference file too long
300 : index out of range
301 : division by zero
303 : CASE label defined twice
304 : this constant is not allowed as case label
400 : expression too complicated (register overflow)
401 : expression too complicated (codetable overflow)
402 : expression too complicated (branch too long)

403 : expression too complicated (jumptable overflow)
404 : too many globals, externals and calls
405 : procedure or module body too long (codetable)
410 : (UNIX only) no priority specification allowed
411 : (UNIX only) global variable expected
923 : standard procedure or function not implemented
924 : parameter must not be accessed with a WITH
940 : compiler error, adr. cat. violation in PASS4
941 : displacement overflow in ari addressing mode
942 : 32bit by 32bit multiply/divide not yet implemented
943 : index range must not exceed positive integer range
944 : jump too long (overflow in pc-relative offset)
945 : offset too long (overflow in pc-relative offset)
946 : FOR control variable is not of simple addressing mode
974 : step 0 in FOR statement
981 : constant out of legal range
982 : overflow/underflow in index/offset/address calculation
983 : use UNIXCALL instead of standard procedure call
990 : too many WITH nested
991 : CARDINAL divisor too large (> 8000H)
992 : FOR control variable must not have byte size (for step # -1, 1)
993 : INC, DEC not implemented with 2nd argument for byte variable
994 : too many nested procedures
995 : FOR step too large (> 7FFFH)
996 : CASE label too large (> 7FFFH)
997 : type transfer function not implemented
998 : FOR limit too large
999 : missing symbol file(s)

History & Acknowledgements

A long time ago at the Institut für Informatik, ETH Zürich, Leo Geissmann wrote a multi-pass Modula-2 compiler for the PDP 11 and Hermann Seiler later wrote a 68000 code generator.

In the Spring of 1985, ETH Zürich released a Modula-2 system in the public domain. It included a compiler, a linker, library modules, and a loader. It was the first native code Modula-2 compiler for the Macintosh. This implementation was a combination and translation of the compilers written by Geissmann and Seiler.

Soon after the public domain release, MacLogimo appeared. It was written by Peter Fink and Franz Kroenseder at ETH. It included a rather complete Macintosh Toolbox library for its time.

In 1986, Tim Myers (while working for Dr. Robert Burton at Signetics) bootstrapped MacLogimo to work on the Macintosh Plus and named it MacLogimo Plus. Dr. Burton graciously allowed MacLogimo Plus to be released into the public domain.

The logistics of distributing a large development system for free proved to be difficult and Project Modula was formed to promote the widespread use of Modula-2 by distributing MacLogimo Plus by mail and through computer information services. Project Modula charges a nominal materials and handling fee of $20 when MacLogimo Plus is ordered by mail. The source code for MacLogimo Plus is also available from Project Modula on condition that it will not be commercially exploited and your derivative sources will be made available under similar terms. Please write for details.

Portions of this document were derived from the original GUIDE.TXT file from ETH.

DISCLAIMER OF WARRANTY

Some may find the following disclaimer of warranty to be offensive; others may think it is humorous. Nevertheless, we live in a venomous world replete with rapacious plaintiffs and lawyers. The legal need to disclaim warranties and limit liability is ample proof.

Since Project Modula has placed MacLogimo Plus in the PUBLIC DOMAIN, all warranties are disclaimed. We strongly urge you not to use MacLogimo Plus nor any other public domain software for mission critical applications.

There are two ways you may have aquired the software:

1. Direct

If you have ordered this software package through the mail DIRECTLY FROM PROJECT MODULA, you may have paid a nominal fee for the materials and labor of distribution and not the software package itself. Please read the following warranty disclaimer before using the software package. If you do not agree to the terms, return the diskettes in good condition for a prompt refund. If you use the software package, we will assume you have, read, understood, and agreed to our warranty terms.

2. Indirect

IF YOU HAVE NOT RECEIVED THIS SOFTWARE PACKAGE DIRECTLY FROM PROJECT MODULA, the integrity of the software package is questionable. Please read the following warranty disclaimer before using the software package. If you do not agree to the terms, erase your copies of the software or if you can't erase it (i.e. it's on a CD-ROM), just stop using it. If you continue to use the software package, we will assume you have, read, understood, and agreed to our warranty terms.

THIS SOFTWARE PACKAGE IS DISTRIBUTED "AS IS" AND WITHOUT WARRANTIES AS TO PERFORMANCE OR MERCHANTABILITY. INDIVIDUALS INVOLVED IN THE DISTRIBUTION OF THE SOFTWARE MAY HAVE MADE STATEMENTS ABOUT THIS SOFTWARE. ANY SUCH STATEMENTS DO NOT CONSTITUTE WARRANTIES AND SHALL NOT BE RELIED ON BY THE USER IN DECIDING WHETHER TO USE THE PROGRAM.

THIS PROGRAM IS DISTRIBUTED WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES WHATSOEVER. BECAUSE OF THE DIVERSITY OF CONDITIONS AND HARDWARE UNDER WHICH THIS PROGRAM MAY BE USED, NO WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE IS OFFERED. THE USER IS ADVISED TO TEST THE PROGRAM THOROUGHLY BEFORE RELYING ON IT. THE USER MUST ASSUME THE ENTIRE RISK OF USING THE PROGRAM. PROJECT MODULA MAY NOT BE HELD LIABLE FOR ANY DAMAGES.

IN NO EVENT SHALL PROJECT MODULA BE RESPONSIBLE FOR ANY INDIRECT, SPECIAL, INCIDENTAL, CONSEQUENTIAL, OR SIMILAR DAMAGES OR LOST DATA OR PROFITS TO THE USER OR ANY OTHER PERSON OR ENTITY REGARDLESS OF THE LEGAL THEORY, EVEN IF PROJECT MODULA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

PROJECT MODULA MAKES NO WARRANTY OF THE PERFORMANCE OF THE LIBRARIES WHEN USED IN YOUR SOFTWARE. YOU AGREE TO INDEMNIFY PROJECT MODULA FROM ALL CLAIMS BY THIRD PARTIES ARISING IN CONNECTION WITH THE USE OF YOUR SOFTWARE.