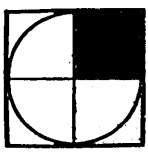


LOGITECH

MODULA 2/86

on ms-dos (pc-dos)

LOGITECH SA - CH-1143 APPLES

Phone (0)21 / 77 45 45 - Telex 458 217 - Fax (0)21 / 77 31 36

Software Engineering Library

Release notes for release 1.0 on DOS 2.0

1) File handling in DOS 2.0

Modula-2/86 now internally uses DOS 2.0 calls for file operations. For this reason you must have a file 'CONFIG.SYS' on the the disk you use to start (boot) your DOS 2.0 system. This file must at least contain the command

```
FILES=12
```

in order to set the maximum number of open files in DOS 2.0 to twelve. This minimum is required for proper operation of the Modula-2/86 system (compiler, linker, debugger). By default, DOS 2.0 allows opening four files only, which is not sufficient operating Modula-2/86.

If the proper number of files is not set in DOS, then an error message 'file not found' will appear when operating the Modula-2/86 compiler. While the file in question may be present, it cannot be opened due to a lack of file descriptors in the operating system.

Also it is recommended that you set the number of buffers to at least thirteen (not just eight as stated in the manual). This can be achieved by the command

```
BUFFERS=13
```

which also goes in the file 'CONFIG.SYS'. While this is not required, it will improve the performance of the Modula-2/86 system.

A sample 'CONFIG.SYS' file is provided on your Modula-2/86 system diskette. After you have changed or installed your 'CONFIG.SYS' file, be sure to re-start (boot) your operating system, in order for these changes to become effective.

2) Installation

The Modula-2/86 system has been greatly improved for operating with DOS 2.0. For the best installation please refer to chapter five on installation in the new Modula-2/86 manual.

3) Command line arguments

When a Modula-2/86 program is executed using the 'm2' command, any text which follows the name of the program is taken as keyboard input. This means, for example, that it works to say 'm2 comp myprog/batch/noaquery'. (The /noaquery compiler option tells the compiler not to request user input when a file cannot be found, and the /batch option tells the compiler not to read type-ahead.) Of course this works for any Modula-2/86 program that does keyboard input using the modules Terminal or InOut.

This allows Modula-2/86 programs to be used more easily with the DOS Batch facility, which requires that all input to a program be on the command line. Because the compiler, linker and debugger accept a space (as well as <CR>) to terminate an argument, multiple arguments may be given on the command line. Example: 'm2 link overlay1/b mainline'

Differences from previous Releases of Modula-2/86

1) Using existing Modula-2 programs with release 1.0

All sources must be recompiled, because the new run-time system (M2.EXE) is incompatible with the code generated by previous versions of the Modula-2/86 compiler. Also, the old reference files (.REF) are incompatible with the new debugger.

2) Running Modula Programs

The Modula-2/86 Resident Part that was present in earlier versions of the system has been removed. In order to run Modula-2 programs you now enter

m2 programname

This also holds for the Modula-2/86 compiler, linker and debugger, which now are started by 'm2 comp', 'm2 link' and 'm2 dbug'.

3) Linking Modula-2 Programs

Because the Modula-2/86 Resident Part has been removed, the Modula-2/86 linker now by default links programs so that they can be run by 'm2 programname'. With previous versions of Modula-2/86 only the modules that were not contained in the Resident Part were included when linking a program. Now all imported modules are by default included by the linker into the resulting '.LOD' file.

4) Specifying filenames

When the Modula-2/86 system (compiler, linker, debugger) prompts for a filename, you now get the default extension by just entering the filename and carriage return. The dot which was required before must not be typed anymore!

5) New library modules

- DiskDirectory
- NumberConversions
- Strings
- RS232Code
- RS232Int
- DiskFiles => replaces module 'DiskSystem'

6) The interfaces (i.e. definition modules) of the following library modules have been changed:

- FileNames
- Options
- Terminal
- Termbase
- Keyboard
- System
- DiskSystem => replaced by module 'DiskFiles'
- FileSystem (additions only)
- Program (additions only)

7) The following libraray modules are no longer supported:

- Comint
- Random

8) Improved handling of compile-time errors:

In case of compile time errors the compilation is stopped after the pass that found the error. The offending line and an error message is displayed for every error found. This display can be interrupted by hitting a key.

9) Other changes:

- The compiler can now generate code for subrange tests, array index tests and arithmetic overflow checking, and this is the default.

- Pointer/address expressions will generate an error 'expression too complicated' in more cases than in previous releases. This problem can usually be solved by introducing temporary variables.

- All the assembly programs of the run-time system have been changed. If you had to adapt these, you might need to re-do your adaption with the new version.

Restrictions

- No emulation of floating point arithmetic is available. The compiler option 'Emulator/Coprocessor' must not be set to 'Emulator'.

- The largest amount of memory that can be allocated by one call to procedure ALLOCATE exported from module Storage is 64K bytes minus one paragraph, i.e. 65536 - 16 = 65520 bytes.

- Certain relatively simple pointer/address expressions may cause the error 'expression too complicated'. This problem can usually be solved by introducing temporary variables.

Bugs

- In the debugger when the text window is displayed for the first time, the arrow indicating the current line is at the wrong position (one line before the current line). The correct position is shown after switchin to another window and coming back to the text window.

- Incorrect test code is generated in case of expressions like:

```
k - i
```

  where

```
CONST k = ... (* 0 <= k <= 32767 *)
VAR i: INTEGER;
```

  The compiler generates test code for CARDINAL (instead of INTEGER) arithmetic, which causes a wrong CARDINAL overflow in case the result is less than zero.

- It is suggested that you do not write a definition module which exports both an enumeration type and also some or all of its elements. There should be no real need to do this, because exporting (the name of) an enumeration type implicitly exports all of its elements.

LOGITECH
Systemes et logiciels pour le traitement du texte
et de l'information
LOGITECH SA CH - 1143 APPLES
Tel. (021) 77 45 45 Telex 458 217

MODULA-2/86

DISTRIBUTION DISKS

Modula-2/86, Release 1.0

Logitech S.A., February 29, 1984

Format: IBM-PC, double sided, 9 track (360K), PC-DOS 2.0

Number of disks: 3

Disk 1: "System"

file description

M2.EXE
Run-time support for Modula-2/86. This program is used to execute Modula-2 programs. Typing 'm2 prog' to the command interpreter of your operating system will start the Modula-2 program contained in the file 'PROG.LOD'.

*.BAT
Command (batch) files for the installation of Modula-2/86. For more information see the section on "Installation" in the manual.

CONFIG.SYS
Example DOS configuration file. You will need to copy (and maybe edit) this file if the disk you use to start your operating system does not yet contain such a file. If you already have your own DOS configuration file you might need to adapt it to the requirements of the Modula-2/86 system.

EXAMP*.MOD
Simple examples for Modula-2 programs.

*.DEF
Definition modules of the Modula-2/86 system and library. These files have been added for documentation only and must not be modified nor re-compiled, since this would cause version conflict errors during execution of the Modula-2/86 compiler and linker.

*.MOD
Implementation modules of some library modules. These sources have been added mainly for your documentation. They may however be adapted by you.

DISPLAY.MOD, KEYBOARD.MOD
These modules are part of the Modula-2/86 system. Their implementations might need some adaptation to your terminal.

COMP.LNK, LINK.LNK, DBUG.LNK
Modula-2 object files of the main programs for the Modula-2/86 compiler, linker and debugger. These object files are needed when re-linking the compiler, linker and debugger after an adaptation of the modules "Display" and/or "Keyboard".

COMPPARA.*
The compiler parameter module may be adapted, e.g. in order to set different default values for the compiler options.

M2COMP.LNK, PUBLIC.LNK, COMPFILE.LNK
Modula-2 object files for some compiler modules. These object files are needed when re-linking 'M2COMP' after

an adaptation of the compiler parameter module "CompPara".

COMP.MAP
Map file for the main program of the Modula-2/86 compiler. This map file is needed when re-linking 'M2COMP' after an adaptation of the compiler parameter module "CompPara".

*.ASM, *.INC
Sources of the run-time support, written in 8086 Macro Assembler language (Intel, Microsoft compatible syntax). Modify this program with utmost care in case you want to include assembly routines or if you have to adapt it to your hardware.

Assembly command: 'MASM RTS;', 'MASM LOADER;', etc. Generate an executable file with:

```
LINK RTS+SERVICES+TRANSFER+PMD+LOADER;
```

After successfull test rename RTS.EXE to M2.EXE.

Disk 2: "Compiler"

file description

COMP.LOD
Main program of the Modula-2/86 compiler.

M2COMP.LOD
Compiler base loaded by the compiler main program 'COMP'.

INIT.LOD
PASS1.LOD
PASS2.LOD
PASS3.LOD
PASS4.LOD
LISTER.LOD
SYMFILE.LOD
Overlays of the Modula-2/86 compiler. All compiler overlays must reside on the same disk (directory) as the compiler base and main program.

*.SYM
Symbol files of the Modula-2/86 system and library modules. These files are required for compilation of modules that use library modules.

Disk 3: "Linker", "Debugger"

file description

LINK.LOD, M2LINK.LOD
Modula-2/86 linker for programs and sub-programs (overlays).

*.LNK
Modula-2 object files of the system and library modules. These files are required when linking programs that use library modules.

DBUG.LOD, M2DBUG.LOD
Modula-2/86 symbolic debugger.

*.REF
Modula-2 reference files of the system and library modules. These files are used for symbolic debugging.

LOGITECH

MODULA-2/86
USER'S MANUAL

CODE AND EDITION

LU-GU101-0
Initial issue: February 1984
Current revision: February 1984

This edition applies to Release 1.0 of the software.

Printed: February 8, 1984

Logitech, Inc.,
805 Veterans Blvd., Redwood City, CA 94063, USA

Logitech SA,
CH-1143 Apples, Switzerland

Logitech Srl,
Corso Nigra 60, 10015 Ivrea TO, Italy

PREFACE

This manual is intended to allow a programmer, even one who is inexperienced with the Modula-2 language, to begin programming in Modula-2 using the Logitech Modula-2/86 system. It is not intended to teach Modula-2 programming, as this is better presented in the book "Programming in Modula-2" by Niklaus Wirth, Second, Corrected Edition, Springer-Verlag 1983. This book defines the Modula-2 language and should be used as a reference for learning the language. The primary purpose of this manual is to familiarize the reader with the implementation-specific features of the Logitech Modula-2/86 system.

Section 1 provides a brief introduction to the Logitech Modula-2/86 system.

The system requirements for running Modula-2/86 are listed in Section 2.

Section 3 provides a step-by-step introduction to using the Modula-2/86 system. This section is intended to give non Modula-2 programmers a head start on the language. For experienced Modula-2 users, it provides a quick introduction to using the Modula-2/86 system.

Basic concepts are presented in Section 4.

Section 5 discusses the installation of the Modula-2 system on your computer.

Operation of the Compiler and Linker programs are described in sections 6 and 7 respectively.

The symbolic debugger is described in section 8.

Section 9 contains a list of the library modules provided with the Logitech Modula-2 system and a general discussion of their use.

The first Appendix is a glossary of terms which have specific meanings in this manual. Additional appendices provide more detailed information on the following subjects: version checking, system dependent facilities, compiler error messages, memory organization, object file format, the standard procedure DOSCALL, system configuration, and detailed descriptions of the library modules.

TABLE OF CONTENTS

* * *

1. INTRODUCTION

Welcome to Modula-2 !

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

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

- extensive library of standard modules
- support for the 8087 for fast, accurate math
- support for the full 1 Megabyte address space of the 8086/8088
- access to underlying hardware and operating system functions
- support for the creation of overlays on very large systems
- a symbolic debugger
- capable of generating ROMable code

2. SYSTEM REQUIREMENTS

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

1. IBM PC or "compatible" with:

   - 192K or more of RAM memory
   - 2 double sided disk drives (300K+ each)
   - PC-DOS, MS-DOS, or CP/M-86 operating system

2. Other systems or configurations supported:

   - 8086 or 8088 based microcumputer running MS-DOS, CP/M-86, Concurrent CP/M-86 or MP/M-86
   - enough disk space to hold the compiler and the other programs (600K minimum)

   Contact Logitech for current list

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

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

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

3. GETTING STARTED

This section provides a brief hands-on introduction to using the Modula-2/86 system to program in Modula-2. Commands are given for PC-DOS and MS-DOS. If you are using a different operating system you may need to refer to the documentation for your operating system.

The following syntactic conventions are used in this manual:

> In this manual, input a user must type on the keyboard is underlined.
>
> Special keys such as 'escape' and 'carriage return' are abbreviated and enclosed in brackets (e.g. <ESC>, <CR>).
>
> Control characters (characters entered while the key marked 'ctrl' is depressed) are preceded by a caret (e.g. ^C, ^X)

3.1. Setting up

The first step is to remove the distribution diskettes from their packaging!

Next, make copies of all of the Modula-2 disks and put the originals away in a safe place. The documentation for your operating system describes how to duplicate diskettes.

Create a working system disk with your operating system and usual utilities on it, including a text editor. Leave about 180K for the Modula-2/86 system. Copy the following files from the Modula-2/86 "SYSTEM" diskette (in drive b:) to your new working disk (in drive a:). This may be accomplished on PC/MS-DOS system by typing b:install0 <CR>

```
M2.EXE (for MS-DOS) or M2.CMD (for CP/M-86)
EXAMP*.MOD
```

Check your disk for the file 'CONFIG.SYS'. If your disk does not contain such a file, you may copy the one that is provided on the Modula-2/86 system diskette. If your disk

already contains a file 'CONFIG.SYS', then you must make sure that this file contains the command 'FILES=12', i.e. that it sets the number of open files to a value of twelve or more. After installing or changing 'CONFIG.SYS', you must re-start (boot) your system for this to take effect.

Hard disk system users may wish to copy all the files on the diskettes onto one directory on the hard disk and to work there. In the steps that follow, hard disk users can ignore all references to disk drives and drive names: everything happens on the hard disk!

Memory disk and other exotics: If you have enough memory you may wish to run Modula-2/86 from a pseudo disk in RAM. Otherwise, it is probably best to follow the dual-floppy pattern, and treat your special device as the system diskette (assumed to be drive A below) and use any floppy for the second device (drive B below).

3.2. Sample program

To provide an example of how to compile and link programs in the Modula-2/86 system, the Modula-2 source file EXAMPl.MOD has been included on the "SYSTEM/INSTALLATION" diskette and should be on your working disk.

```
MODULE  Exampl;
  FROM  Terminal IMPORT WriteString, WriteLn, Read;
  VAR   ch: CHAR;
BEGIN
  WriteString ('The program worked! (Hit a key)');
  WriteLn;
  Read (ch);
END Exampl.
```

In Modula-2 programs, UPPER CASE AND LOWER CASE LETTERS ARE DISTINCT, so be sure you write keywords such as 'MODULE', 'FROM', etc. in upper case.

In the instruction 'FROM Terminal IMPORT...', the module 'Terminal' refers to one of the library modules which is included in your Modula-2/86 package. This module exports the objects 'WriteString' (which writes a string on the screen), 'WriteLn' (which writes <CR><LF> to the screen) and 'Read' (which reads a character from the keyboard). In order to use these objects, their names must be written exactly as shown. (See the appendix on library definitions for more information about the module

3.2.1. Compiling

Now insert the Modula-2/86 COMPILER disk in drive B, and enter

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

The compiler asks you to enter the filename of the program with the prompt 'source file>'. Since the default device and default filetype ('MOD') are correct in this case, you need only enter

```
EXAMP1<CR>
```

The following should be displayed on the screen:

```
A>m2 b:comp<CR>
Modula-2/86 Compiler V m.n
Copyright (c) 1983 Logitech
 source file> EXAMP1<CR>.MOD
p1
 Terminal      in file: B:Terminal.SYM

p2
p3
p4
termination
      The setting of the options was:
            emulator   (E): off
            stacktest  (S): on,
            rangetest  (R): on
            indextest  (T): on
      No code for 8087 Processor generated
      Codesize:       90 bytes
      Datasize:        1 bytes
end compilation

A>
```

p1 .. p4 denote the succession of activated compiler passes.

Your compiled files have been written to the disk drive where your source file was found. After the compilation, you are back in the command interpreter of your operating system. The successful compilation will have created the files 'EXAMP1.LNK' (containing the object code) and 'EXAMP1.REF' (containing debugger information).

3.2.2. Linking

After successful compilation, your program must be linked. Insert the LINKER disk in drive B:. Enter

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

The linker asks you to enter the filename of your compiled program. Enter the filename followed by <CR>. The default disk drive is the current disk drive, and the default filetype is 'LNK'.

```
A>m2 b:link<CR>
Modula-2/86 Linker Vm.n
Copyright (C) 1983 Logitech
   master file  > EXAMP1<CR>.LNK
   linked with:
      Terminal       in file: B:Terminal.LNK
      Termbase       in file: B:Termbase.LNK
      System         in file: B:System.LNK
      Keyboard       in file: B:Keyboard.LNK
      ASCII          in file: B:ASCII.LNK
      Display        in file: B:Display.LNK
   name of output file: A:EXAMP1.LOD
   name of map file: A:EXAMP1.MAP
end linkage

A>
```

3.2.3. Execution

Your linked program has been written to your current disk and can now be executed by the Modula-2 system. Simply enter

```
m2 EXAMP1<CR>
```

and the following will appear on the screen:

```
A>m2 EXAMP1
The program worked! (Hit a key)
```

3.3. Sample program #2

To allow non Modula-2 users to get a better understanding of how to typical I/O functions are implemented in

Modula-2, the following BASIC program was converted to Modula-2.

```
100 '    routine to demonstrate file I/O
110 '
120 INPUT "enter filename, lines to read -";F$,NL
130 OPEN "I",#1,F$
140 INPUT "enter output filename or device -";FO$
150 IF FO$="" THEN FO$="scrn:"
160 OPEN "O",#2,FO$
170 PRINT CHR$(13);"listing of file - ";F$
180 IF EOF(1) THEN 250
190 IF LN=NL THEN 250
200 LINE INPUT #1,L$
210 LN=LN+1
220 PRINT #2,L$
230 GOTO 180
240 '
250 CLOSE #1
260 IF LN<NL THEN PRINT #2,CHR$(13); \
                        "only";LN;" lines in file"
270 STOP
```

The same functionality is attained in Modula-2 by the following program:

```
MODULE Examp2;
  (* Program Module to demonstrate basic file I/O *)

IMPORT InOut;
FROM InOut IMPORT
    OpenInput, OpenOutput, CloseInput,
    CloseOutput, Read, Write, EOL;
IMPORT Terminal;
  (* get qualified access to Terminal routines *)
IMPORT CardinalIO;

CONST ESC = 33C;

VAR Ch: CHAR;
    LinesToCopy, LinesCopied: CARDINAL;

BEGIN
  (* Note that interaction with user is done
   * via Terminal and CardinalIO,
   * because InOut input/output
   * is being redirected to files.
   *)
  REPEAT
    Terminal.WriteString("enter input file:");
    Terminal.WriteLn;
    OpenInput("");   (* input file, no default extension *)
```

```
  UNTIL InOut.Done;  (* keep trying until successful     *)
  REPEAT
    Terminal.WriteString("Lines to copy> ");
    CardinalIO.ReadCardinal(LinesToCopy);
    Terminal.WriteLn;
    Terminal.Read(Ch);  (* read terminator *)
  UNTIL Ch <> ESC;      (* keep until ' ' or EOL *)
  REPEAT
    Terminal.WriteString("enter output file:");
    Terminal.WriteLn;
    OpenOutput("");     (* output file, no default *)
  UNTIL InOut.Done;
  LinesCopied := 0;
  LOOP
    IF LinesCopied >= LinesToCopy THEN EXIT END;
    Read(Ch);           (* read from in file    *)
    IF NOT InOut.Done THEN EXIT END;
    Write(Ch);
    IF Ch = EOL THEN INC(LinesCopied) END;
  END; (* LOOP *)
  IF LinesCopied < LinesToCopy THEN
    Terminal.WriteString("[Only ");
    CardinalIO.WriteCardinal(LinesCopied,0);
    Terminal.WriteString(" lines in file]");
    Terminal.WriteLn;
  END;
  CloseOutput;
  CloseInput;
END Examp2.
```

Modula-2 also allows the creation of libraries, so that functions previously implemented need not be recoded. Brief descriptions of the Library modules provided for use by Modula-2/86 programmers are provided in the section entitled 'System and Library Modules,' and more detailed information appears in the appendix on library definitions.

4. BASIC CONCEPTS

This section presents some of the basic concepts needed to write programs using the Modula-2/86 system. First we will present the concept of program modules, definition modules and implementation modules which are the building blocks of Modula-2. We will then present an overview of how the Modula-2/86 system works - from library creation to compilation of a user program. A brief explanation of the file naming conventions is also provided.

4.1. Module types

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

4.1.1. Program Module

A progam module is the main user program. From it all the procedures necessary to execute a user program are called. The examples in the preceding section are program modules. Program modules have the form:

```
MODULE  <modulename>;

  library modules to use
  FROM  <libname>
    IMPORT <list of names separated by commas>;
     or
  IMPORT <libname>;

  variable and procedure definitions (module body)

BEGIN
```

```
    executable program code

END <modulename>.
```

The list of names imported may contain the names of types, variables and procedures exported from a library module. These names must be separated by commas. Refer to the Wirth book for a more detailed explanation.

4.1.2. Definition Modules

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

```
DEFINITION MODULE  <modulename>;

  library modules to include use
  FROM  <libname>
    IMPORT <list of names separated by commas>;
     or
  IMPORT <libname>;

  EXPORT QUALIFIED <list of names>;

  type and variable declarations, procedure headings

END <modulename>.
```

4.1.3. Implementation Modules

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

Implementation modules have the form:

```
IMPLEMENTATION MODULE  <modulename>;

  library modules to use
  FROM  <libname> IMPORT <procedurename(s)>
     or
  IMPORT <libname>

  variable definitions
  procedures

END <modulename>.
```

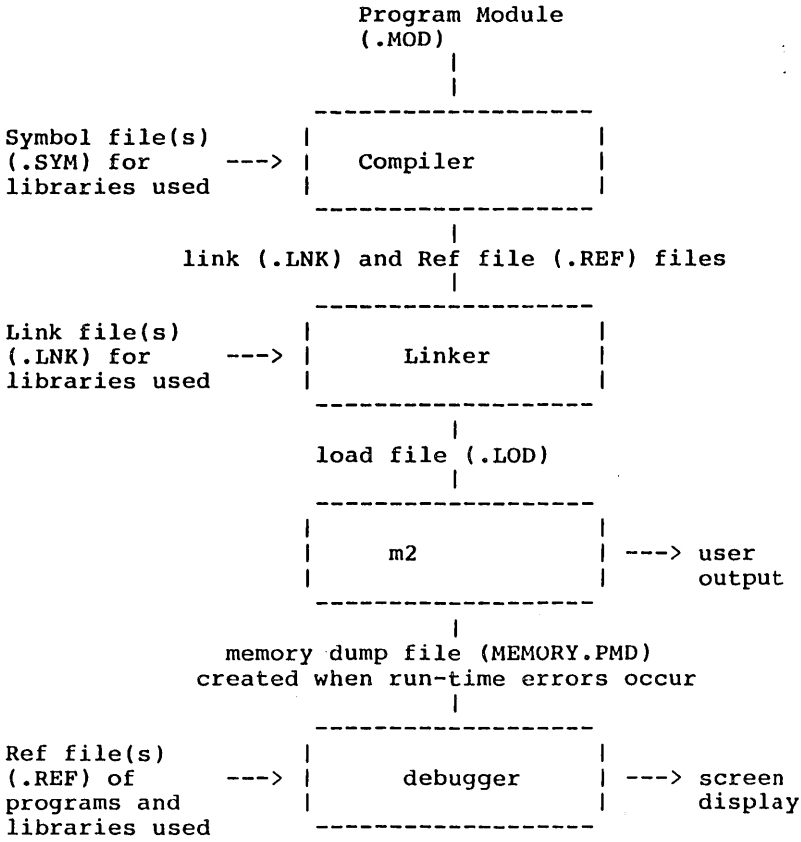
The type and variable definitions given in the corresponding definition module (with the same name) must not be repeated in the implementation: they are known implicitly. However, for every procedure specified in the definition part, a complete procedure (with matching name and parameter list) must be contained in the implementation part.

4.2. How the system works

There are two primary operating modes of the Modula-2/86 system. These are the creation of executable programs and the creation of library modules. An example of creating an executable program was presented under the section on getting started. In this section we will explain what the Modula-2 system is doing in each of these two modes.

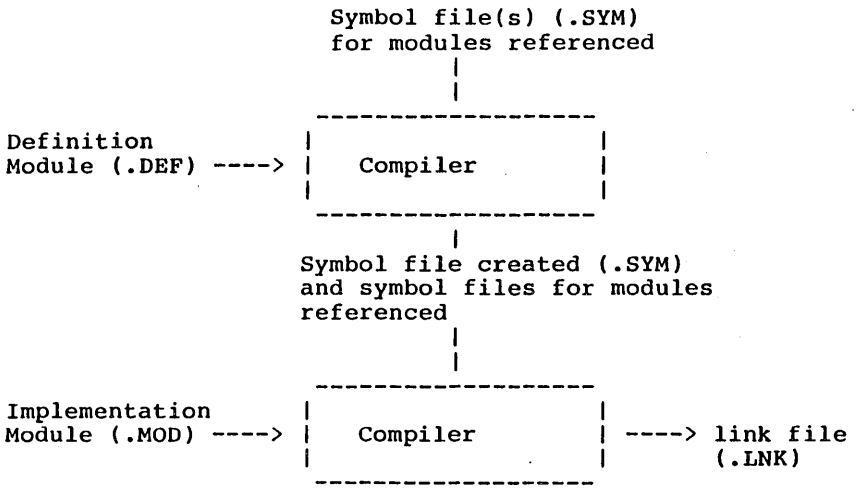
4.2.1. Creation of an executable program

The creation of an executable program begins with a program module file (somename.MOD) which is converted into a .LOD file and executed in the following way:



4.2.2. Library Module creation

The creation of a library module begins with a definition module (somename.DEF) and a matching implementation module (somename.MOD). These are converted into .SYM and .LNK files which may be used in the creation of user programs. Library modules are created in the following way:



4.3. Running Modula-2 programs

In order to run Modula-2 programs you must call the program 'M2' which is supplied with the Modula-2/86 system. You should copy this program to the diskette or directory which you use to develop your Modula-2 programs or to the directory where you generally keep executable programs (see the section on disk installation for details). The program M2 consists mainly of the Modula-2 run-time support (RTS) and has the ability to load and execute Modula-2 programs. The default extension for executable Modula-2 programs is 'LOD'.

A running Modula-2 program can be terminated by typing ^C. This will get you back to the operating system, which will prompt for the next command.

To run a Modula-2 program enter 'M2 <programname>'. The program name may also be preceded by a drive and/or a directory name, in which case the program is loaded from

the specified drive or directory. If a pure file name is entered the Modula-2 program will be searched in the current directory on the current drive, and (if not found) in the directory '\m2lod' on the current drive.

Here is a brief example:

```
C>m2 sieve
Sieve of Eratosthenes benchmark
10 iterations
1899 primes
C>
```

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

4.4. File naming conventions

The file naming conventions (file extensions) used in the PC-DOS/MS-DOS version of Modula-2/86 are listed alphabetically below.

.BAT Batch

Batch files contain commands to the operating system. They are also known as command files.

.DEF Definition

Definition files are the definition parts of Modula-2 modules.

.EXE Executable program file

File directly executable under PC-DOS/MS-DOS. M2.EXE is the only DOS executable program in the Modula-2/86 system.

.LNK Link

Compiler output file with the generated 8086 code in linker format generated during the compilation of an Implementation or Program module.

.LOD Load

Linker output file with the generated 8086 code ready for execution by M2.EXE.

.LST List

Normally generated only if errors occur.

.MAP Map file

Load map produced by the linker when overlays are to be used.

.PMD Program Memory Dump

Output file produced when errors occur when running a program. This is an input file required for the symbolic debugger.

.REF Reference file

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

.SYM Symbol file

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

5. INSTALLATION

In the section "Getting Started", you created a Modula-2/86 work disk. The following section provides more complete instructions on how to install your Modula-2/86 system, and on how to make the best use of it. This is quite simple in the case of a system with floppy disks only. It is a little bit more complicated if you have a hard disk system, and you want to take full advantage of the features of your Modula-2/86 system.

For both cases, some command (batch) files are provided on the Modula-2/86 system diskette, that can help you to perform the necessary operations. These files assume a standard configuration (a floppy disk drive A, and either a floppy disk drive B or a hard disk drive C). You might have to modify them or to do the installation step by step if your system configuration differs from the assumed standard.

5.1. Configuring your operating system

As already mentioned under "Getting Started" you need to configure your operating system in order to run Modula-2/86. This must be done by setting up the file 'CONFIG.SYS' on the the disk that you start (boot) your operating system from. It is recommended that this file contain the following three commands:

```
FILES=12
BUFFERS=8
DEVICE=ANSI.SYS
```

FILES=12

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

BUFFERS=13

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

DEVICE=ANSI.SYS

This command gives access to the "Extended Screen and Keyboard Control. Some parts of Modula-2/86 assume that this driver is used. If this command is omitted in 'CONFIG.SYS' then certain control characters that are written to the display may not have the effect as specified in the definition module 'Terminal' (see appendix on library definitions).

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

For more information, please refer to the documentation for your operating system.

For special installation of the Modula-2/86 system, i.e. if you need to adapt the assembly part (run-time system) or the implementation of heavily hardware-dependent modules (Keyboard, Display), please see the appendix on "system configuration".

5.2. Systems equipped with floppy disks only

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

First prepare a Modula-2 work disk. Insert a copy of the Modula-2/86 system disk into drive B, and an empty (formatted) diskette into drive A. Now copy the file 'M2.EXE' (the Modula-2/86 run-time support) from drive B to your diskette in drive A. Your Modula-2 work disk is now ready, no other files are required.

In order to perform this copy, you may simply type

b:install0<CR>

which will copy the file 'M2.EXE' and the sources of the example Modula-2 programs to your diskette in drive A.

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

- the Modula-2/86 compiler (.LOD),
- the symbol files (.SYM) of the system and

library modules.

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

- the Modula-2/86 linker (.LOD),
- the link files (.LNK) of the system and library modules.

If the Modula-2 linker and debugger disk are the same, your preparations are complete. Otherwise, prepare a copy of the Modula-2/86 debugger disk. Use this disk when debugging Modula-2 programs. Your debugger disk contains

- the Modula-2/86 debugger (.LOD),
- the reference files (.REF) of some library modules.

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

- the Modula-2/86 Run-Time System (M2.EXE),
- your Modula-2 source files,
- any other files you created with your Modula-2/86 system and which are needed for compiling, linking or debugging

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

In order to compile you insert a copy of the compiler disk into drive B.

In order to link you insert a copy of the linker disk into drive B.

After linking, you can execute your program on your work disk in A by typing

m2 programname<CR>

If errors occur during execution that produce a memory dump (A:MEMORY.PMD), insert a copy of the debugger disk into drive B in order to debug.

Note that, depending on the capacity of your disks, you can combine two or more of the above described disks in one disk, e.g., the linker and the debugger disks could be the same physical disk.

If your diskettes have a large capacity, it may even be worthwile for you to study the following section on hard

disk systems and on the environment used by Modula-2/86 (see also the section on 'search strategies').

5.3. Systems equipped with a hard disk

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

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

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

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

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

a:INSTALL1<CR>

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

'ref', and 'map'.

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

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

After these directories have been created (by the running the command file or step by step), your are ready for the next step. With the Modula-2/86 system diskette in drive A execute the second command file. Simply type

a:INSTALL2<CR>

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

For a step by step installation copy all the files with extension 'LOD' from the three distribution diskettes to the directory '\m2lod'. Then copy all the files according to their extension into the corresponding directory. E.g. copy the files with extension 'DEF' to the directory '\m2lib\def', copy the files with extension 'MOD' to the directory '\m2lib\mod', and so on. If you think you will need the sources of the RTS, too, then copy the files from the system diskette with extension 'ASM', 'INC' and 'OBJ' to the directory '\m2lib'.

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

```
SET M2SYM=\M2LIB\SYM
SET M2LNK=\M2LIB\LNK
SET M2REF=\M2LIB\REF
SET M2MAP=\M2LIB\MAP
```

These commands will set up the environment for Modula-

2/86. In this way your Modula-2/86 system can take full advantage of the DOS features and your hard disk.

Some more information on the environment used by Modula-2/86 can be found in the section on 'search strategies' of this manual.

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

5.4. Default names and search strategy

No special manipulation is required to build or to use the library of modules. All on-line modules, residing on hard disk or floppy disk, comprise the library. The compiler and linker automatically search for referenced modules. The default search strategy can be modified by command options. This allows you to change disks during compilation and linkage, thus spreading your library over several floppy disks. Note that during the operation of compiler and linker, all needed files must be on-line.

The compiler, linker, and debugger construct the filename for a library module from its module name, truncating the module name if it is longer than a file name may be, and appending the appropriate extension (SYM, LNK, etc.) to the file name.

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

5.4.1. The default search strategy

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

The first search is always done using the 'source' or 'master' path. The source or master path is the path you specify when entering the name of the file to compile or link. If the file is not found using this path, the

current path (i.e. the current drive and directory as know to DOS) is checked for the file. If it is not found there, a third and last automatic search is done using the path from where the compiler (or linker, debugger) was loaded. If the file still cannot be opened, you will be prompted to type in the (path and) file name.

(If your system has floppy disks only, you may ignore the following, up to the end of this subsection.)

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

```
SET M2SYM=\M2LIB\SYM
SET M2LNK=\M2LIB\LNK
SET M2REF=\M2LIB\REF
SET M2MAP=\M2LIB\MAP
```

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

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

Note that empty paths are not allowed in environment strings used by the Modula-2/86 system. The current disk and directory must be denoted by 'DK:'.

Example:

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

```
SET M2SYS=DK:;\mystuff\mylib;\m2lib\sym
```

and that a compilation was started by

```
C> m2 comp
Modula-2/86 Compiler V m.n Copyright (C) 1983 Logitech
source file>a:myprog<CR>
```

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

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

5.4.2. The query search strategy

The query search strategy is always applied by the Modula-2/86 system when you are prompted to type in the (path and) file name of a library file. This can happen when a file is not found using the default search strategy, or when you specify the 'query' option when compiling (or linking, or debugging).

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

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

You can just enter <CR>, which means that the file name should be constructed from the module name, and that the

default search strategy (as explained above) will be applied.

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

You can just enter a drive name (terminated by ':'). In this case the file name will be constructed from the module name and searched for on the specified drive. Only one attempt to open the file will be made.

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

6. THE COMPILER

This chapter describes the use of the Modula-2 compiler. The compiler is run by typing 'm2 comp.'

After displaying 'source file>', the compiler is ready to accept the filename of the module to be compiled, as well as some options (see the subsection on options below.

Example:

```
A>m2 comp
Modula-2/86 Compiler V m.n - Copyright (C) 1983 Logitech
 source file>
```

The default device is the current disk and the default filetype is 'MOD' (for program modules and implementation modules).

6.1. Compiler Organization

The compiler is organized in a base part and several passes (overlays). The base part remains in memory during the entire compilation and calls the passes sequentially. When loading these passes, the compiler assumes that they are on the same drive as the compiler base. The necessary files are:

COMP.LOD	compiler base
INIT.LOD	initialization
PASS1.LOD	syntax analysis
PASS2.LOD	declaration analysis
PASS3.LOD	body analysis
PASS4.LOD	code generation
SYMFILE.LOD	symbol file generator
LISTER.LOD	lister

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

6.2. Compiler Output Files

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

.SYM symbol file

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

.REF reference file

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

.LNK Object (link) file

Compiler output file with the generated 8086 code in linker format, generated during compilation of an implementation or a program module.

.LST listing file

Normally generated only if errors occur.

6.3. Compilation of a Program Module

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

```
A>m2 b:comp<CR>
Modula-2/86 Compiler V m.n
Copyright (C) 1983 Logitech
 source file> EXAMP1<CR>.MOD
p1
 Terminal      in file: B:Terminal.SYM
p2
p3                     the succession of activated
```

```
p4                      compiler passes is indicated
termination
        The setting of the options was:
              emulator   (E): off
              stacktest  (S): on
              rangetest  (R): on
              indextest  (T): on
        No code for 8087 Processor generated
        Codesize:     90 bytes
        Datasize:      1 bytes
end compilation

A>
```

If errors are detected by the compiler, compilation stops after the pass that found the error. The errors are displayed on the screen and a listing file with error messages is generated.

```
A>m2 b:comp<CR>
Modula-2/86 Compiler V m.n
Copyright (C) 1983 Logitech
 source file> EXAMP1<CR>.MOD
p1
 Terminal      in file: B:Terminal.SYM
 ---- error
lister
   3    VAR ch CHAR;
 ******             ^37
 *  37: ':' expected

termination
end compilation

A>
```

The error display can be interrupted by hitting a key (unless compiler option '/batch' is set).

6.4. Compilation of a Definition Module

Compilation of a definition module (filetype 'DEF') is similar to the compilation of program module. However, as the result of a successful compilation, the compiler produces a symbol file (filetype 'SYM') instead of a linkable object (LNK). The symbol file contains the declarations of the definition part in symbolic, compiler-readable format. It also contains a unique module key which is used for consistency checking. If errors are detected by the compiler, then a listing file is generated instead of the symbol file.

A DEFINITION PART MUST BE COMPILED PRIOR TO ITS IMPLEMENTATION PART.

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

Example:

```
A>m2 b:comp<CR>
Modula-2/86 Compiler V m.n
Copyright (c) 1983 Logitech
 source file> EXAMP3.DEF<CR>
p1
p2
symfile
termination
end compilation

A>
```

6.5. Compilation of an Implementation Module

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

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

```
A>m2 b:comp<CR>
Modula-2/86 Compiler V m.n
Copyright (c) 1983 Logitech
 source file> EXAMP3<CR>.MOD
p1
 Examp3        in file: A:Examp3.SYM
 Storage       in file: B:Storage.SYM
p2
p3                      the succession of activated
p4                      compiler passes is indicated
termination
        The setting of the options was:
              emulator   (E): off
              stacktest  (S): on
              rangetest  (R): on
              indextest  (T): on
        No code for 8087 Processor generated
        Codesize:    234 bytes
        Datasize:     56 bytes
```

```
end compilation

A>
```

6.6. Symbol Files Needed for Compilation

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

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

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

When the compiler asks for a symbol file, the request is repeated until an appropriate file is found or <ESC> is hit. Hitting <ESC> means that the file is not available. The compiler will stop at the end of the first pass, but first it will list all the required symbol files. This allows you to detect any other missing files.

6.7. Compiler Options

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

6.7.1. Table of available options

option	value for ON	value for OFF	default
query	Query	NOQuery	NOQ
autoquery	Aquery	NOAquery	AQ
interactive	Interactive	Batch	I
listing	Listing	NOListing	NOL
errorlisting	EListing	NOEListing	EL
emulator/ coprocessor	Emulator	Coprocessor	C
version	Version	NOVersion	NOV
statistics	STATistics	NOSTATistics	STAT
stacktest	S+	S-	S+
rangetest	R+	R-	R+
overflowtest	T+	T-	T+
header in listing	Header	NOHeader	H
footer in listing	Footer	NOFooter	NOF
date in listing	DAte	NODAte	NODA
debug info	Debug	NODebug	D

The default value for all options can be set in the

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

6.1.2. Description of the options

/Q /NOQ Query option

/A /NOA Autoquery option

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

	query	autoquery	action
	Query	Aquery	ask for filenames
	Query	NOAquery	ask for filenames
default:	NOQuery	Aquery	tries to find file by default strategy if not found it asks for filename
	NOQuery	NOAquery	if not found compiler ends.

/I /B Interactive/Batch option

This option tells the compiler whether it runs interactive or as a batch job. In the interactive mode, it is possible to stop the display of the error message by hitting a key. This facility is turned off in the batch mode. NOTE: the autoquery option is not affected by this option.

/L /NOL Listing option

/EL /NOEL Error listing option

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

Listing	errorlisting	action
Listing	EListing	always a listing file is generated
Listing	NOEListing	as above
NOListing	EListing	if errors detected, a listing file is generated
NOListing	NOEListing	no listing generated

In all cases the compiler writes the lines with errors and the errormessage on the screen.

/E /C Emulator/Coprocessor option

This option affects the code generation for the floating point arithmetic. If it is set to coprocessor, the compiler generates inline code for the Intel 8087 numeric processor. In the other case it generates code for the Intel 8087 Emulator.
Note: Version 1.0 of the Modula-2/86 system does not include a floating point emulator. Therefore, this option must not be set to 'Emulator'.

/V /NOV version option

The compiler displays information about the running version , e.g. processor and operating system flags.

/STAT /NOSTAT statistics option

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

/S+ /S- Stacktest option

/R+ /R- Range test option

/T+ /T- Overflow test option

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

/H /NOH Header listing option

/F /NOF Footer listing option

/DA /NODA Date in listing option

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

Modula-2/86 filename.ext

Date Page nr.

The text for the footerline must be defined in the parameter module.

/D /NOD The option debug:

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

6.8. Compiler Directives in Modules

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

Letter	Meaning
R	Subrange and arithmetic overflow test (default R+).
T	Index test (arrays, case) (default T+).
S	Stack overflow test (default S+).

Setting	Effect
+	Test code is generated.
-	No test code is generated.
=	Revert to setting before last.

Example:

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

6.9. Compiler Error Messages

There are two types of compilation errors:

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

Operational errors which are displayed on the screen.

6.9.1. Source text errors

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

6.9.2. Compiler Operational Errors

----file not found

The source or symbol file was not found and the compiler will repeatedly request the filename. You can exit this loop either by typing the correct filename or by hitting <ESC> if the required file is missing.

----error

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

----symbol files missing

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

----code table overflow

The code length of each procedure and module initialization is limited to 3500 bytes. To prevent this error you must break up your large procedures into multiple small procedures.

----output disk full

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

----file creation failed

Your disk directory is probably full.

----compiler error

We hope that this message will never occur. It is displayed when a compiler self-check is not successful. If you get this error please contact Logitech with a copy of the program which caused the error.

7. THE LINKER

The linker combines all the separately compiled modules into a single load and executable module.

7.1. How to use the Linker

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

```
A>m2 b:link<CR>
Modula-2/86 Linker V m.n
Copyright (c) 1983 Logitech
master file =>
```

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

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

Example:

```
A>m2 b:link<CR>
Modula-2/86 Linker Vm.n
Copyright (c) 1983 Logitech
   master file  > EXAMP1<CR>.LNK
   linked with:
      Terminal       in file: B:Terminal.LNK
      Termbase       in file: B:Termbase.LNK
      System         in file: B:System.LNK
      Keyboard       in file: B:Keyboard.LNK
      ASCII          in file: B:ASCII.LNK
      Display        in file: B:Display.LNK
   name of output file: A:EXAMP1.LOD
   name of map file: A:EXAMP1.MAP
end linkage

A>
```

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

When the linker asks for an object file, the request is repeated until an appropriate file is found or <ESC> is hit. Hitting <ESC> means that the file is not available. The linker will stop at the end of the first pass, but first it will list all the required object files. This allows you to detect any other missing files.

After successful linking, the program is written to a load file with the same name and on the same disk as the master file, filetype LOD.

7.2. Linking Options

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

/B Base layer option (Default B-)

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

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

When the linker prompts for a file name during the linking of a subprogram, <ESC> can be entered to indicate that this module will be in the base layer when the program is run. When entering the file name for the base map an additional option is available:

/Q Query resident modules (Default Q-)

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

If you do not have the map file of the base, (e.g. if you link the subprograms before you link the base) simply type <ESC>. Note: In this case the Q option for the main module must be on (see below).

/M Map file option (Default M+)

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

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

/Q Query option (Default Q-)

If turned off (default) the linker will search for each link file using a default name (the module name) with the filetype 'LNK', on the disk where the master file was found. The linker enters a query mode any time a link file is not found. In this case, the linker prompts for the correct name.

If the 'Q' option is turned on you are asked to enter the name of each link file. You can specify the default name with <CR>.

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

/A Automatic query option (default A+)

If turned on (default) the linker enters a temporary

query mode when a link file is not found. The A-option prevents this automatic query mode and is useful in command files.

7.3. Linker Error Messages

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

----file not found

The link or map file was not found and the linker requests the file name (again) if option A is turned on. This loop is exited by typing the correct file name or by entering <ESC> (see option Q above).

----Linkage aborted: end of medium

Insufficient disk space to write the load file causes the linker to stop execution. To solve the problem, delete superfluous files or copy some files to the disk on the other drive. The load file is written to the disk where the main module is found.

----map file incomplete: end of medium

This message appears when there is insufficient disk space to write the map file. Generation of the load file is not affected by this error. To solve the problem, make more room on the output disk (the disk which holds the main module) or turn off generation of the map file (see option M above).

8. THE SYMBOLIC DEBUGGER

This chapter describes the symbolic, postmortem debugger distributed with Logitech's Modula-2 software development system. When a program stops because of a run-time error, because the HALT statement is executed or because Ctrl-C (BREAK) is typed, the memory is dumped to disk in the file MEMORY.PMD. The debugger is used to inspect this dump file.

When using the debugger, the reference files of the modules which constitute the program should be present. These reference files are generated by the compiler when compiling an implementation module and have the filetype 'REF'. Without them no symbolic examination of program or data is possible. Source files ('MOD') can be used to show the text of the analyzed program inside the debugger.

8.1. How to use the Debugger

The debugger is run by typing 'm2 dbug'. The debugger will display "Modula-2/86 Debugger" and the version number and a copyrigth notice. It prompts you for the name of the dumpfile to be used. The default for this filename is 'MEMORY.PMD', which will be opened on the current disk if not otherwise specified. It will then ask if you want to enter the names of the required files (files of type 'REF' and file names. Any other response is interpreted as 'no' and the debugger will use the default file names (composed of the first 8 characters of the module name plus filetype 'REF' or 'MOD'). The debugger then tries to find these files according to the search strategy explained above. If a file cannot be found, the debugger asks you for the correct filename. Hit <ESC> if the file is not available. Without the 'REF' file, no symbolic debugging is possible for that module.

Some modules are considered to belong to the 'system library' and the debugger does not prompt for their filenames, if they are not found with the default search strategy. This concerns the modules: System, Keyboard, Termbase, Terminal, Program.

The debugger displays 4 different types of information which will be referred to as 'Process window', 'Data window', 'Text window' and 'Memory window'. Only one of the four windows can be displayed at any one time. When you initiate the debugger, the Process window is shown (with

the position at which the error occured).

8.2. The Debugger Commands

The following global commands are available in all four windows and may be used whenever the debugger is ready to accept commands:

8.2.1. Selecting a window

P go to Process window
Displays the process window of the process which caused the dump, i.e the state and procedure call chain of the dead process will be displayed.

D go to Data window
Displays the data of the last selected procedure or module. This selection is made in either the Process window or with the 'M' command in the Data or Text window.

T go to Text window
Displays the text of the last selected procedure or module. This selection is made in either the Process window or with the 'M' command in the Data or Text window. If a procedure is selected in the Process window, the portion of the text, containing the call to the next procedure in the calling-chain is shown. If a module is selected with the 'M' command, the beginning of that module is shown.

C go to Memory window
The memory area around the current address will be displayed in the selected display mode (see 'Memory window'). The current address is the data address of the selected procedure or module in the Process window.

8.2.2. Selecting a process for debugging

@ (at sign) go to process window of a new process
Its main use is to select a PROCESS variable in the Data window and then entering this command. The state and procedure call chain of the selected process will be displayed.

If used in the Process window, the procedure call chain of the dead process is shown.

In the Memory window, the current position is taken as the new process descriptor and its state and procedure call chain is shown.

This command is not available in the Text window.

8.2.3. Leaving the debugger

Q quit the debugger
quit the debugger You will be prompted for a veto. Enter 'y' to terminate the debugging session.

8.2.4. Positioning

A number of commands in the debugger allow you to choose the element in your program that you wish to analyze. When you choose a new element (e.g., a variable in the Data window), this becomes the current element for any subsequent commands.

There are two ways of repositioning:

1) by typing a new element number, or

2) by moving the contents of the screen either up or down relative to your current position with the following commands:

^ (up arrow) move up by half of a screen
Multiple strikes of this key are possible, to move faster. The debugger waits about one second before executing this command.

_ (underscore) move down by half screen
Multiple strikes of this key are possible, to move faster. The debugger waits about one second before executing this command.

(n)+ move n lines up
The parameter n is optionnal, default for n is 1. This command is not available in the Memory window.

(n)- move n lines down
The parameter n is optional, default for n is 1. This command is not available in the Memory window.

8.2.5. Display mode

<ESC>S Scroll Mode
Text is scrolled off the top of the screen and new lines are added at the bottom or vice-versa.

<ESC>P Page Mode
Text is displayed a screen at a time. The screen is cleared before being refilled. This mode is the default.

<ESC>L Lines
You will be asked how many lines should be displayed at once. This command can be used independently in both scroll and page mode. Minimum value is 4.

<ESC>B Bell toggle
With this command you can turn on and off the bell, which is used to signal erroneous input.

8.3. The Four Windows

8.3.1. Process Window

Shows the procedure call chain together with the address of the call to the next procedure in the chain. This address is shown as the position relative to the beginning of the module as well as the line number in the source text. The address shown in the first procedure is the instruction pointer at the moment when the dump occurred. If a reference file is missing, the procedure names are replaced by numbers within the module which refer to the order of procedure declarations within a module.

Typical use: position to one procedure and go to the Data or Text window.

8.3.2. Data Window

Shows variables and parameters of the current procedure or module. It also allows you to view the contents of structured variables. On the first line, the debugger shown the name of the procedure or module being examined. If the content of structured data is shown, the complete 'path' is indicated. This 'path' includes the name of the procedure or module, the name of the variable and the names of the substructures, including indices in case of arrays. The last displayed line gives the current position inside the list of displayed data. Initially, this position is 1. It can be modified by means of the global positioning commands. The current position is used by the commands 'S' and 'A' below.

M Modulename
Select data of another module. The list of all modules in memory at the time when the dump was taken is shown. Select a module from this list by entering its number. This selection is valid for subsequent operations in the Text and Data window.

S Son
Display the data structure beneath the current data element i.e., if the current item is a record, then 'S' will show the fields of this record. Local modules are shown as data of the embedding module. To view the data of the local module, position on that module in the Data window and type 'S'. Multiple 'S' may be typed to skip several levels of data structure. The debugger waits about 1 second before executing the 'S' command, to let the user enter

multiple 'S'.

F Father
Displays the data structure above the current data (the reverse of 'S'). Multiple 'F' may be typed to skip several levels of data structure. The debugger waits about 1 second before executing the 'F' command, to let the user enter multiple 'F'.

A Address
Displays the address of the current data. This can be used before a 'C' command to display the memory area around a variable. This command has the side effect of modifying the 'current address' of the Memory window.

V Variable
Goes back to the first level of the current procedure or module, i.e. shows its variables.

R Right

L Left
Displays the element to the right (or left) of the current element, if the current position is an element of an array. If the current position is inside a record, which is an element of an array, then this command moves to the neighboring element of the array, without changing the field in the record.

The global command '@' (at-sign) can be used most conveniently in the Data window, by selecting a variable of type PROCESS and typing '@'. The Process window of that selected process will be shown.

8.3.3. Text Window

Shows the text of the current module and procedure. An arrow at the beginning of the line shows, where this procedure has been left (call of next procedure or error occurred).

Text window commands are:

M Modulename - select another module for this window. The list of all modules in memory at the time when the dump was taken is shown. Select a module from this list by

entering its number. This selection is valid for subsequent operations in the Text and Data window.

8.3.4. Memory Window

Displays the content of memory around the 'current address'. An arrow indicates the current position. When entering the Memory window, that position is the data address of the procedure, selected in the Process window.

The representation of the data can be chosen by the following commands (number sign, followed by a letter):

#U - unsigned CARDINAL format

#I - INTEGER format

#W - WORDS (hex) format (default mode)

#B - BYTES (hex) format

#C - CHAR format

#T - Text format

#A - ADDRESS format

#R - REAL format

The display mode is kept unchanged, when leaving the Memory window and re-entering it later on.

Modification of the current address:

The selection of a procedure in the Process window sets the current address to the beginning of the procedure's data area.

The global positionning commands modify the current address.

In the Memory window a new current address can be selected by typing a number instead of one of the above commands. Just enter up to 5 hexadecimal digits (0..0FFFFF), terminated by <CR> or space.

Modifications to the current address are lost when leaving the Memory window. When re-entering this window, the data address of the procedure selected in the Process window is

always used as the current address.

9. SYSTEM AND LIBRARY MODULES

Unlike many other programming languages, the Modula-2 language does not provide standard functions that handle input and output. In Modula-2 systems, these functions are typically provided by library modules. Library modules that implement the typical, basic operating system functions are also called 'system modules' or 'low level modules'.

Besides these system modules the Modula-2/86 module library contains a number of additional, more general library modules. Examples are modules that provide for formatted input/output, or modules that provide functions that do not depend on the operating system (e.g. string handling).

This section provides an overview about the system and general library modules available with Modula-2/86. The definition parts of all library modules are given in the appendix on "library definitions".

9.1. The system modules

There is a number of system modules that constitute the interface between the Modula-2/86 system and the application programs. They provide access to the underlying operating system and to the Modula-2/86 system itself. These modules may be used (imported) directly by application programs.

System modules rely on the support of the operating system. For some of them, even the definition part is operating system dependent. These system modules should be considered as internal modules of the Modula-2/86 system. They should not be used directly by application programs, because this would severely affect program portability.

Highly system dependent modules of the Modula-2/86 system that should not be used directly (their filenames are given in parentheses):

```
- DiskFiles          (DISKFILE.DEF)
#;
- Display            (DISPLAY.DEF)
- Keyboard           (KEYBOARD.DEF)
```

Partially system dependent modules of the Modula-2/86

system (should be used with care):

- System (SYSTEM.DEF)
- Termbase (TERMBASE.DEF)

System modules that are part of the interface between the Modula-2/86 system and application programs:

- DiskDirectory (DISKDIRE.DEF)
- FileSystem (FILESYST.DEF)
- Program (PROGRAM.DEF)
- Storage (STORAGE.DEF)
- Terminal (TERMINAL.DEF)

9.2. The general library modules

General library modules provide features that are not typically part of operating systems. In some cases, they provide access to operating system features through higher level concepts.

The Modula-2/86 library includes the following general library modules (their filenames are given in parentheses):

- ASCII (ASCII.DEF)
- CardinalIO (CARDINAL.DEF)
- Conversion (CONVERSI.DEF)
- FileMessage (FILEMESS.DEF)
- FileNames (FILENAME.DEF)
- InOut (INOUT.DEF)
- MathLib0 (MATHLIB0.DEF)
- NumberConversion (NUMBERCO.DEF)
- Options (OPTIONS.DEF)
- Processes (PROCESSE.DEF)
- ProgMessage (PROGMESS.DEF)
- RS232Code (RS232COD.DEF)
- RS232Int (RS232INT.DEF)
- RS232Polling (RS232POL.DEF)
- RealInOut (REALINOU.DEF)
- Strings (STRINGS.DEF)

9.2.1. Brief descriptions of library modules

For quick reference, an alphabetical listing of the library modules with brief functional descriptions of each is provided here. More detailed information is available in the appendix on library module definitions.

MODULE ASCII

Symbolic constants for non-printing ASCII characters

MODULE CardinalIO

Terminal input/output of CARDINALs in decimal and hex

MODULE Conversions

Convert from INTEGER and CARDINAL to string

MODULE DiskDirectory

Interface to directory functions of the underlying OS

MODULE DiskFiles

Interface to disk file functions of the underlying OS

MODULE Display

Low-level Console Output

MODULE FileMessage

Write file status/response to the terminal

MODULE FileNames

Read a file specification from the terminal.

MODULE FileSystem

File manipulation routines

MODULE InOut

Standard high-level formatted input/output.

MODULE Keyboard

Default driver for terminal input.

MODULE MathLib0

Real Math Functions

MODULE NumberConversion

Conversion between numbers and strings

MODULE Options

Read a file specification, with options, from the terminal

MODULE Processes

(pseudo-) concurrent programming with SEND/WAIT

MODULE ProgMessage

Write program status message to the terminal

MODULE Program

Sub-program loading and execution

MODULE RS232Code

High-speed interrupt-driven input/output via the serial port

MODULE RS232Int

Interrupt-driven input/output via the serial port

MODULE RS232Polling

Polled input/output via the serial port

MODULE RealInOut

Terminal input/output of REAL values

MODULE Storage

Standard dynamic storage management

MODULE Strings

Variable-length character strings handler.

MODULE System

Additional system-dependent facilities

MODULE Termbase

Terminal input/output with redirection hooks

MODULE Terminal

Terminal Input/Output

APPENDICES

APPENDIX A - Glossary

The following terms have specific meanings as used in the manual.

Base layer

A program which calls another program is the base layer of the called program. For instance the passes of the Modula-2 compiler are called sequentially by the compiler base, i.e. by their base layer.

Definition module

The definition part of a module. Chapter 24 in "Programming in Modula-2" describes the use of definition modules.

Development system

The entire system (hardware and software) needed to develop a program. This system includes the runtime support as well as the compiler, linker, debugger, and the library.

Language support

An assembly program, which can be viewed as the extension to the hardware which gives the target system the ability to execute programs written in Modula-2. It is part of the run-time support.

Library

The set of all available separately compiled modules.

Implementation module

The actual implementation of the objects defined in the corresponding definition module. The syntax of a module as described in the Modula-2 report applies to the implementation module.

Objects

Anything that can be given a name, i.e. constants, variables, procedures, types, and modules.

Overlay

Overlays in the Modula-2/86 system are provided by the ability to call dynamically loaded (sub)programs. The name 'overlay' comes from the fact that if one subprogram is called and returns, and then another is called, the second one can re-use the memory used by the first.

Program

A set of one or more separately compiled modules, linked together into a load file. A program can be viewed as a Modula-2 procedure that resides in memory only during its execution. When called, it is loaded into memory to perform a specified task and its memory is released as soon as the task is completed. A program can call other programs to any depth, limited only by the size of the workspace. (see the appendix on Global Memory Organization for more details).

Program module

The module which contains the "main program". After initialization of all imported modules, the body of the main module is the actual starting point of the program. A main module may be split into a definition part and an implementation part.

Run-time support (RTS)

An assembly program which includes the language support and further configuration-dependent functions. These functions include typical operating system features such as bootstrapping the resident software, setting up the memory configuration, dumping memory to disk, etc.

Separately compiled module (SCM)

A module as described in "Programming in Modula-2" which is contained in a separate file. These modules can be compiled separately as long as the imported definition modules have already been compiled. If an SCM exports objects it must be split into a definition module and an implementation module; the two modules are then separate compilation units.

Subprogram

A program which is called by another (Modula) program. A subprogram can use objects exported by the calling program. See the module 'Program', in the appendix on library definitions.

System modules

A set of modules, written in Modula-2, that implement higher level operating system functions such as memory management, overlay loading, terminal I/O, file handling, etc.

Target system

The system (hardware and software) on which you execute your application programs. In most cases the target system is the same as the development system: however, this is not a requirement. The hardware configuration does not necessarily include a terminal, nor disks. The software configuration may be reduced to the Modula-2 language support and your program.

Workspace

The region of memory allocated to a process for stack, program variables, heap, and subprograms. When a Modula program is started, it begins execution as the 'default process', and it claims the largest available region of memory as its workspace. A subprogram shares the workspace of its base layer. However, when a new process is created (see the appendix on system dependent facilities and the appropriate section of the book "Programming in Modula-2") it must be assigned a workspace. This may be in a

module's data area, on the heap, or even in a procedure's local data area (on the stack). Just make sure that the process doesn't have a longer lifetime than its workspace!

APPENDIX B - Version Checking

Module key and version checking

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

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

Once a definition part has been compiled, it becomes possible to compile its implementation part, or another module which uses the definition part (a 'client' module). These other modules will mention the definition part, and the compiler will find the compiled version of this definition part, and use it to fully check the module being compiled. The module key of the referenced definition parts are included in the compiled output.

At compile, link or load time, it can be verified that all the keys included for a given definition module are the same. This guarantees that all modules which share an interface were compiled using the same version of that interface. The purpose of this is to ensure that the inter-module checking is as good as it would be if each program was all one source file, compiled all at once.

Version errors and how to fix them

If the version consistency rule is broken, you will get a version error during either compilation, linking, or

(sub)program loading. The following sections describe the typical cause and some possible corrections for version errors.

Version errors during compilation

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

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

Suppose that we compile as follows:

```
X.DEF   =>      X.SYM
B.DEF   =>      B.SYM
X.DEF   =>      X.SYM (version 2)
C.DEF   =>      C.SYM
A.MOD   =>      A.LNK
```

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

Version errors during linking

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

```
MAIN.MOD
    imports InOut and Terminal.
INOUT.DEF
    defines InOut and imports nothing.
INOUT.MOD
    implements InOut, and imports Terminal.
```

TERMINAL.DEF
defines Terminal and imports nothing.

TERMINAL.MOD
implements Terminal, and has no imports.

Now suppose these compilations are done:

```
TERMINAL.DEF    =>   TERMINAL.SYM
INOUT.DEF       =>   INOUT.SYM
INOUT.MOD       =>   INOUT.LNK
TERMINAL.MOD    =>   TERMINAL.LNK
TERMINAL.DEF    =>   TERMINAL.SYM
                - source of inconsistency
MAIN.MOD        =>   MAIN.LNK
```

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

Version errors during loading

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

A typical case: There is a program which contains a module 'Windows' with an interface WINDOWS.DEF. This program calls a subprogram, in which there is a module 'Edit' which IMPORTs Windows.

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

The straightforward correction is to recompile WINDOWS.MOD and relink the base, so both WINDOWS and EDIT agree on the Windows interface. This will require recompiling any other modules in the base which use Windows (so that the

base is consistent), and rebuilding any other subprograms which use the older version of the Windows interface.

APPENDIX C - System Dependent Facilities

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

1. Changes to the language proper.

2. Differences in the set of available procedures and data types, particularly those of the standard module SYSTEM.

3. Differences in the internal representation of data.

4. Differences in the set of available modules, in particular those for handling files and peripheral devices.

Whereas the first three causes affect "low-level" programming only, the fourth pervades all levels, because it directly reflects the available software and hardware resources. See "Programming in Modula-2" for a more detailed description of low-level facilities.

This chapter gives an overview of the Modula-2/86 specific low-level features.

WARNING

All these features should be applied with utmost care, since their use might conflict with the basic software, e.g. Modula-2/86 file system, etc.

Language extensions

Modula-2/86 implements the Modula-2 language exactly as defined in the "Report on The Programming Language Modula-2", without any restrictions or extensions.

The Module SYSTEM

The module SYSTEM offers further facilities to programs written in the Modula-2 language. Most of them are dependent upon the implementation or are specific to the target processor. SYSTEM also contains types and procedures which allow 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 need be supplied for this module. However, the declaration of these objects in the import list is required.

For more detailed information see chapter 12 of the Modula-2 report in "Programming in Modula-2".

Objects exported from Module SYSTEM

Types

BYTE

An individually accessible storage unit (one byte). No operations except type conversions are allowed for variables of type BYTE. An actual parameter of any type that uses one byte of storage may be passed to a formal BYTE parameter. For convenience small CARDINAL constants (<=255) are also allowed as parameters.

WORD

One word of memory (two bytes). No operations except type conversions are allowed for variables of type WORD. An actual parameter of any type that uses one word of storage may be passed to a formal WORD parameter.

ADDRESS

The address of any location in storage. The type ADDRESS is compatible with all pointer types and is itself defined as POINTER TO WORD. The arithmetic operators '+' and '-', as well as all comparisons, apply to this type. The standard procedures INC and DEC apply to ADDRESS with an ADDRESS or with a CARDINAL as second parameter. CARDINALS and ADDRESSES must not be mixed in expressions. It is, however, possible to convert a CARDINAL k to type ADDRESS(k) (although CARDINALS and ADDRESSES differ in

size). Also, constants of type ADDRESS have the same form as constants of the type CARDINAL. The compiler distinguishes them by context and by the allowed range (0 - 65535 for CARDINAL, 0 - 0FFFFFH for ADDRESSES).

PROCESS

A type used for process handling.

Procedures

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

Create a new process. p is the procedure to execute, a is the address of the data area for the process (the workspace), n is the size of the workspace in paragraphs, and p1 receives the created PROCESS object. A paragraph is 16 bytes. Allow 10 paragraphs for system overhead in each workspace. Please note: A common error in using Modula-2/86 is to pass this procedure the workspace size in bytes instead of paragraphs.

```
TRANSFER(VAR p1, p2:PROCESS)
```

Save the current process state in p1, resume execution of the process in p2.

```
IOTRANSFER(VAR p1, p2:PROCESS; interruptNumber:CARDINAL)
```

Save the current process state in p1, resume execution of the process in p2. The next occurrence of the designated interrupt (up to program termination), has the effect of TRANSFER(p2,p1).

```
PROCEDURE LISTEN()
```

Temporarily lower the priority of the calling process and allow pending interrupts to come through.

```
PROCEDURE GETREG(reg : CARDINAL; VAR w: WORD);
PROCEDURE SETREG(reg : CARDINAL; w: WORD);
```

These two procedures are used to set and to retrieve the contents of machine registers. They generate in-line code, and are particularly useful in conjunction with the special procedures CODE and SWI (software interrupt) described below. Only the following registers are accessible:

```
AX =0      CX =1      DX = 2      BX = 3
SP =4      BP =5      SI = 6      DI = 7
ES =8      CS =9      SS =10      DS =11
(These are available as constants from module 'System')
NOTE: The registers SP, BP, CS and SS cannot be set with
      SETREG.
```

If the actual argument for these two procedures is a variable in one byte, only the lower half of the register is affected (e.g., in SetReg (0, ch), where ch is declared to be a CHAR, only the AL register is modified).

WARNING

Utmost care must be exercised when using GETREG and SETREG.

It is strongly recommended that only constants or simple local or global variables or parameters are used as parameters of these functions. Expressions and/or array indexing should be avoided. It must be kept in mind that expression evaluation and address computation use registers and therefore might destroy the value of a register already set by SETREG or to be read by GETREG. It is impossible for the compiler to recognize such a situation and the programmer must take the full responsibility.

Unpredictable effects may result from failure to heed this warning.

CODE(b1, b2, ... :BYTE)

A call to CODE inserts the (constant) values b1, b2, etc., in-line as executable code.

PROCEDURE SWI(interruptNumber: CARDINAL);

This procedure is used to generate a software interrupt. It compiles into an 'INT' instruction. The parameter must be a constant.

PROCEDURE ENABLE; PROCEDURE DISABLE;

Calls to ENABLE and DISABLE compile into 'STI' and 'CLI'

instructions, respectively.

```
PROCEDURE INBYTE(port: CARDINAL; VAR w: WORD);
PROCEDURE OUTBYTE(port: CADINAL; w: WORD);
PROCEDURE INWORD(port: CARDINAL; VAR w: WORD);
PROCEDURE OUTWORD(port: CARDINAL; w: WORD);
```

These four procedures are used to handle the I/O ports. The first parameter must be a constant or a variable that specifies the port number. Variables may only be of the following numerical types: CARDINAL, INTEGER, sub-ranges thereof, and WORD.

```
DOSCALL (fct: CARDINAL; ...)
```

Generates a DOS function call (via software interrupt 224). The parameter list is variable, depending on the first parameter, which must be a constant (the number of the DOS function). The appendix contains a detailed description of the available DOSCALLs.

WARNING

Utmost care must be exercised when using DOSCALL.

It is strongly recommended that only constants or simple local or global variables or parameters are used as parameters of these functions. Expressions and/or array indexing should be avoided. It must be kept in mind that expression evaluation and address computation use registers. Because the parameters of DOSCALL must be given to DOS in registers, the compiler might easily run out of registers.

Functions

```
ADR(variable):ADDRESS
```

Storage address of the parameter variable.

```
SIZE(variable):CARDINAL
```

Returns the number of bytes used in storage by the parameter variable. If the variable is of a record type with

variants, then a variant of maximal size is assumed.

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

Yields the number of bytes used in storage by a variable of the substituted type. If the type is a record with variants, then tag constants of the last FieldList (see syntax in "Programming in Modula-2") may be substituted in their nesting order. If some or all tag constants are omitted, then the remaining variant with maximal size is assumed.

Data-representation

Standard data-types have the following internal representation:

BOOLEAN 1 byte, TRUE=1, FALSE=0

CHAR 1 byte, ASCII character set

ENUMERATION 1 byte, elements are numbered 0..255

INTEGER 2 bytes, -32768..32767, 2's complement notation, least significant byte first.

CARDINAL 2 bytes, 0..65535, least significant byte first.

SET

2 bytes, if we number the elements of a set from 0 to 15, the representation in a memory word is:

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

POINTER
PROCEDURE
ADDRESS

4 bytes, the first two bytes (lower address) hold the offset value (with lower byte first) and the second two bytes hold the segment value (lower byte first).

REAL

8 bytes, Intel 8087 double precision real format (IEEE Floating Point standard).

ARRAY

an array is stored as a contiguous sequence of elements, with the indices in ascending order, the right-most index varying most quickly. If the base type fits in one byte (CHAR, BOOLEAN, ENUMERATION) the elements are stored in sequential bytes. Otherwise, each element is stored on a word (even) boundary.

RECORD

fields are allocated in the order that they are declared. The first field has the lowest address. Two fields each containing data in one byte only (CHAR, BOOLEAN, ENUMERATION) are packed into one word, if declared immediately one after the other. "Byte-fields" are not packed if there is another field in between (order is not changed).

SUBRANGE

Same representation as the base type.

APPENDIX D - Compiler error messages

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

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

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

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

50    : ',' expected
51    : error in formal type
52    : error in statement sequence
53    : '.' expected
54    : export at global level not allowed
```

```
55    : body in definition module not allowed
56    : TO expected
57    : nested module in definition module not allowed
58    : ' ' expected
59    : '..' expected

60    : error in FOR statement
61    : IMPORT expected

70    : identifier specified twice in importlist
71    : identifier not exported from qualifying module
72    : identifier declared twice
73    : identifier not declared
74    : type not declared
75    : identifier already declared in module environ-
        ment
76    :
77    : too many nesting levels
78    : value of absolute address must be of type
        CARDINAL
79    : scope table overflow in compiler

80    : illegal priority
81    : definition module belonging to implementation
        not found
82    : structure not allowed for implementation of
        hidden type
83    : procedure implementation different from
        definition
84    : not all defined procedures or hidden types
        implemented
85    : name conflict of exported object or enumera-
        tion constant in environment
86    : incompatible versions of symbolic modules
87    :
88    : function type is not scalar or basic type
89    :

90    : pointer-referenced type not declared
91    : tagfieldtype expected
92    : incompatible type of variant-constant
93    : constant used twice
94    : arithmetic error in evaluation of constant
        expression
95    : incorrect range
96    : range only with scalar type
97    : type-incompatible constructor element
98    : element value out of bounds
99    : set-type identifier expected

100   : structured type too large
101   : undeclared identifier in export list of the
        module
```

```
102   : range not belonging to base type
103   : wrong class of identifier
104   : no such module name found
105   : module name expected
106   :
107   : set too large
108   :
109   : scalar or subrange type expected

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

120   : incompatible types in conversion
121   : this type is not expected
122   : variable expected
123   : incorrect constant
124   : no procedure found for substitution
125   : unsatisfying parameters of substituted procedure

126   : set constant out of range
127   : error in standard procedure parameters
128   : type incompatibility
129   : type identifier expected

130   : type impossible to index
131   : field not belonging to a record variable
132   : too many parameters
133   :
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
153   : expression expected
```

```
154   : expression not allowed
155   : type of function expected
156   : integer constant expected
157   : procedure call expected
158   : identifier not exported from qualifying module
159   : code buffer overflow

160   : illegal value for code
161   : call of procedure with lower priority not
        allowed

170   : global data too large (more than 64K bytes)
171   : local data too large (more than 32K bytes)
172   : parameters too large (more than 32K bytes),
        too many parameters

200   : compiler error
201   : implementation restriction
202   : implementation restriction: FOR step too large
203   : implementation restriction: boolean expression
        too long
204   : implementation restriction: expression too
        complicated
205   : implementation restriction: procedure too long
206   :
207   : implementation restriction: illegal type
        conversion
208   :
209   : expression too complicated: jump table overflow

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

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

APPENDIX E - Module Priorities

Use of priorities at programmer's level

Priorities can be specified in the header of both inner modules and compilation units. When entering such a module (execution of module body or call of an exported procedure) the interrupt priority mechanism of the system (hardware and software) is set such, that interrupts of a level lower or equal to the one specified in that module are not passed to the processor. When leaving a priority module, the interrupt priority system is reset to the state it was prior to entering that module. The standard procedure LISTEN (to be imported from module SYSTEM) allows to lower the priority temporarily. During the execution of the procedure LISTEN the interrupt priority system is set such that all pending interrupts are accepted.

Inside a priority module, calls to procedures of other priority modules having a lower priority than the module with the call statement are not allowed. This situation is detected by the compiler and an approriate error message is produced (error 161). If a procedure of a module with no specified priority is called, the current priority remains unchanged. If a procedure of a module with higher priority is called, that higher priority becomes effective during execution of the called procedure. The old priority is restored upon return from that procedure.

Priorities are attached to processes. Upon a TRANSFER or IOTRANSFER to a process running at another priority, the system's priority is switched to the one of the process to be activated. The same holds for the implicit coroutine transfer which occurs upon an interrupt.

When a program terminates, the system's priority is set to the value which was effective when the terminating program was loaded.

Range of possible priority levels

Both the compiler and the Run-Time-Support (RTS) allow only a fix range of values for priority levels. Eight levels are supported with values from 0 (lowest level) to 7 (highest level). If a module priority is specified with a

value that is not in this range, the compiler produces an appropriate error message (error 80).

The above values are defaults for the Modula-2/86 system. These defaults may be changed during installation of Modula-2/86:

a) limit in the compiler: The compiler-parameter-module initializes a variable to contain the upper limit of the legal range. By assigning another value to that variable, this upper limit can be modified.

b) limit in the RTS: Please refer to the source program of the RTS for the changes required if another range of priorities has to be supported.

Implementation notes

The field 'interruptMask' in the process descriptor holds the mask which becomes effective upon activation of a coroutine. When creating a new process (function NEWPROCESS), the interrupt mask, valid for the 'father process' is copied into the process descriptor of the new process. The 'father process' is the one which executes the NEWPROCESS instriction. If the procedure which constitutes the process is declared within a priority module, its priority becomes effective upon execution of the procedures entry code. This code is executed after the first TRANSFER to the new process.

When executing a coroutine transfer, i.e. call of TRANSFER, IOTRANSFER or upon an interrupt, the interrupt mask of the running process is updated with the actual mask in the interrupt controller. This new mask is compared to the mask of the process to be activated. If they are different, the mask of the new process is output to the interrupt controller.

The compiler generates a call to the Run-Time-Support (RTS) in the procedure entry code (monitor entry function) and the procedure exit code (monitor exit function) for every procedure exported from a priority module. The standard procedure LISTEN is translated to a RTS call (listen function). These three RTS functions are accessed through software interrupt 0E4H. Their interface is as follows:

```
Monitor Entry:  AL = 5, BX holds the new priority
Monitor Exit:   AL = 6, no parameter
Listen:         AL = 7, no parameter.
```

The monitor entry function is called after the possible stack-test and after the stack pointer is decremented by the size of the local data. It saves the current interrupt mask on the stack of the entered procedure. The new priority is used as an index in a table that contains the mask for every priority level. This mask - combined with the currently valid mask - is output to the interrupt controller and stored in the process descriptor. The table containing the priority masks is defined in the RTS. The default value masks bit 0 for priority 0, bit 0 and 1 for priority 1, and so on. These values may be modified upon configuration of the Modula-2/86 compiler, in order to implement any priority schema.

The monitor exit function finds the old interrupt mask topstack. This old mask is ouptut to the interrupt controller and stored in the process descriptor. The information on the stack relative to the priority is removed.

The listen function unmasks all the bits in the mask of the interrupt controller and sets the processor's interrupt flag, thus allowing all pending interrupts to be passed to the processor. After execution of a no-operation instruction, the old mask as well as the old interrupt flag are restored.

APPENDIX F. - Memory Organization

Global Memory Organization

The global memory organization after loading of the Modula-2 Resident part is shown in Figure F-1:

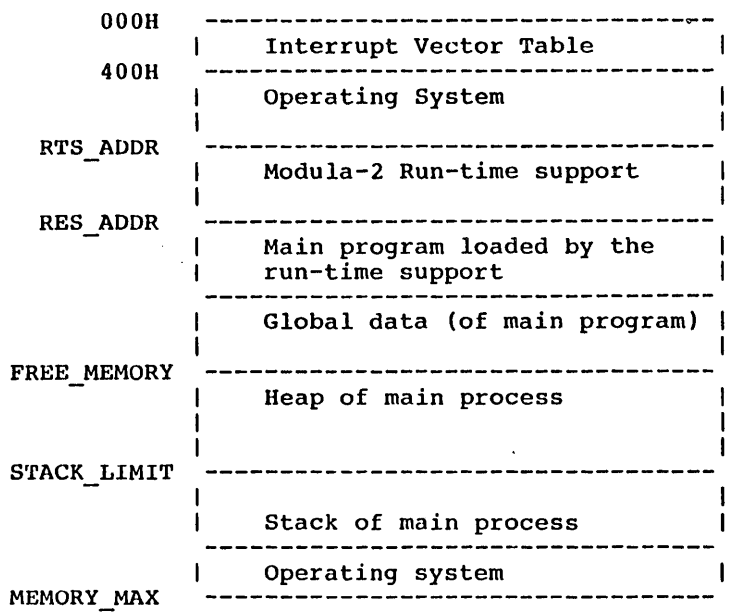


Figure F-1 Initial memory organization

RTS_ADDR

Load address of the Run-Time Support. Depends on the size of the operating system. The RTS is a relocatable program, containing a code and a data segment.

RES_ADDR

Load address for a Modula-2 program. The Modula-2 code is relocatable and loaded immediately after the end of the

RTS code by means of a loader, which is part of the RTS.

FREE_MEMORY

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

STACK_LIMIT

This value is a sort of 'high water marker' of the heap. It varies during the execution of a process as a function of the occupation of the heap. If the stackpointer of a process becomes equal to or less than its STACK_LIMIT, there is a stack overflow.

MEMORY_MAX

End of RAM area.

Overlays

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

The programmer is not concerned about where an overlay is loaded. To load and execute an overlay, one simply calls the procedure 'Call' from module 'Program'. This procedure finds a memory area (on top of stack), large enough to hold the overlay. The relocatable loader then loads the program in that area and control is passed to it. After termination of the overlay (either normal termination or caused by an error) execution continues at the statement that immediately follows the call to the procedure 'Call' and the memory occupied by the overlay is freed.

The size on an overlay is limited only by the physical address space, i.e a maximum of 1 MB.

After loading of an overlay, the memory layout is as shown in Figure F-2 (only the higher part of memory is shown, see also Figure F-1).

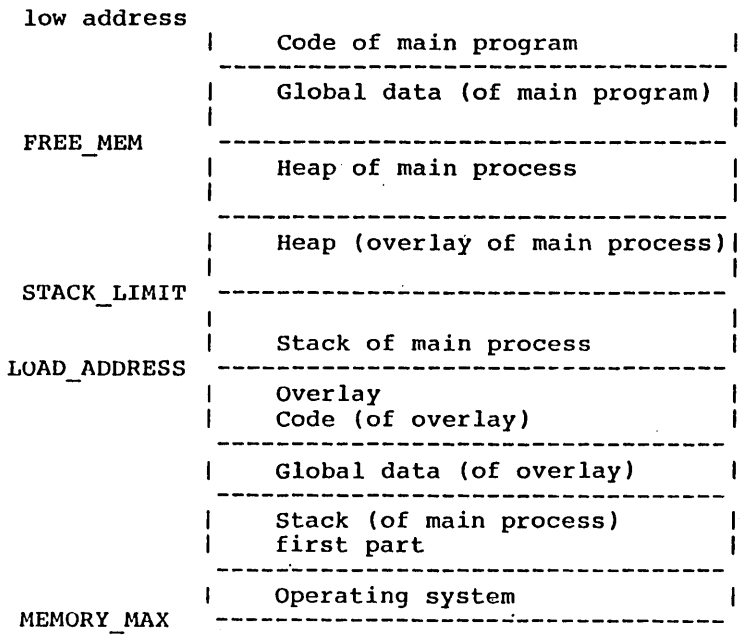


Figure F-2 Loading an overlay

Processes

The workspace of a process is the region of memory allocated to this process for its local and dynamic variables (stack and heap). The workspace holds also code and data of subprograms (overlays) called by this process. A part of the workspace is used for the process descriptor, needed for the TRANSFER mechanism. This process descriptor is initialized through a call of (the standard procedure) NEWPROCESS. Figure F-3 shows an example of a process workspace.

When starting the Modula-2 Run-Time System, the main process is automatically created. This 'default process' gets the largest available region of memory as its workspace. The program which is loaded by the Run-time support runs as the main process. Programs loaded by other Modula-2 programs (through module 'Program') run as overlays to the main process, i.e no new process is created to execute overlays.

When a new process is created (by means of NEWPROCESS), it must be assigned a workspace. This region has to be explicitly defined by the programmer. It is usually a variable, owned by the father process. Such a variable can be global (e.g. an ARRAY declared at the level of a module), dynamic (created on the heap by a call to NEW) or local to a procedure (placed on the stack). If a non-global variable is used, make sure that the process doesn't have a longer lifetime than its workspace!

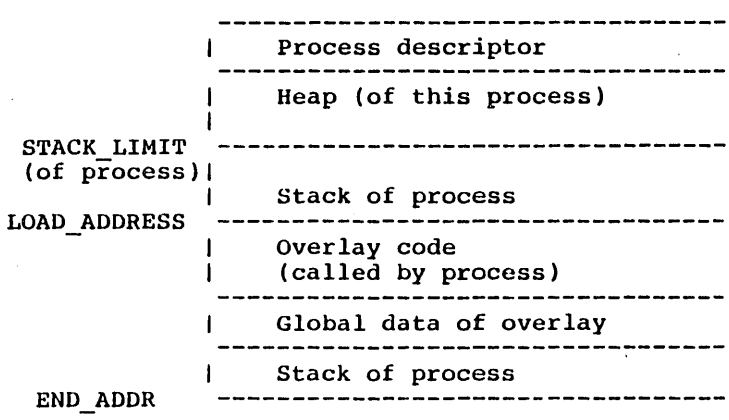


Figure F-3 Process workspace

Heap

The heap is the memory area between the bottom of the workspace (after the process descriptor) and the top of stack. The heap manager allocates and deallocates portions of this memory upon request through the standard procedures NEW and DISPOSE. To avoid unpredictable memory occupation due to the concurrency of multiple processes, the heap of every process is administrated independently.

The library module 'Storage' implements the default heap manager. The same strategy of allocation and deallocation is used for the main process and all user processes. Upon allocation of memory portions, collision with the stack is detected and leads to a termination of the program with the status 'heap-overflow'.

Stack

The stack holds 3 kinds of data:

1. procedure interfaces (parameters, return address)

2. local data of procedures, allocated upon entry in a procedure

3. temporary values during the evaluation of an expression

Every process has its own stack. Upon creation of a process (call of NEWPROCESS), the stack is set such, that the first word pushed occupies the last word (highest even address) in the workspace. The stack grows from the end of the workspace (highest address) to the beginning of the workspace (lowest address). After loading of an overlay, the stack is set to continue at the address just below the load address of the overlay.

The size of a stack can be at most 64K bytes, i.e the stack segment of a process is never modified, except when loading an overlay (which creates a new stack). In most applications however, a process' workspace will be less than 64KB and therefore the stack size is limited by the size of the workspace and the occupation of the heap.

Variable allocation

All variables on 2 or more bytes, including elements of arrays and fields of records, are word aligned. Two consecutively declared variables of 1 byte are stored in the same word. If a 1-byte variable is declared between two longer variables it occupies 1 word, with the significant byte at the lower address. The unused byte is undefined.

Procedure Interface

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

1. the procedure is imported from another module

2. the procedure is exported

3. the procedure is used in an assignement to a procedure variable

4. the procedure is used as the body (starting point) of a process.

If none of these conditions is true, the procedure is called with a NEAR call.

Before calling a procedure, the parameters (if any) are pushed on the stack in the same order as they are declared. A value-parameter on 1 byte occupies 2 bytes on the stack, with the value in the lower byte and an undefined higher byte.

After the parameters are pushed on the stack, the static link of the called procedure is pushed on the stack if its level is higher than 0 (i.e. if the called procedure is nested). The static link is the base pointer of the procedure in which the called procedure is declared. A base pointer is an address inside the stack, pointing to the local data of a procedure. By means of the static link, a nested procedure has access to the local data of its embedding procedure.

After the parameters (if any) and the static link (if needed) are passed, the procedure is called.

The stack layout at this point is shown in Figure F-4:

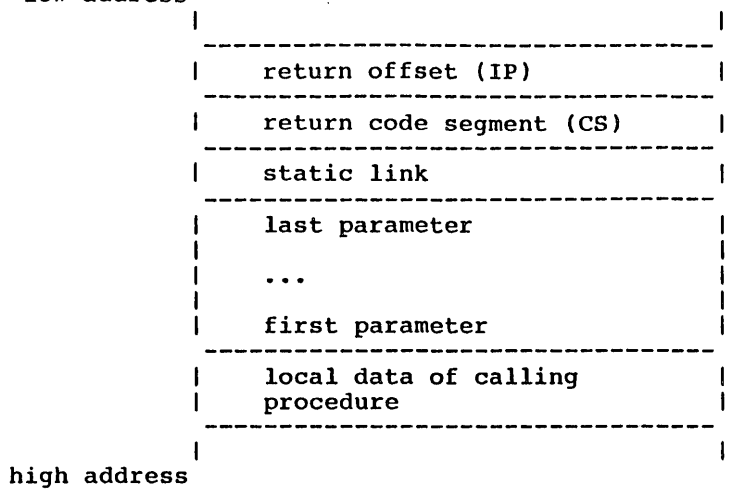


Figure F-4

Now the called procedure gains control and executes the following procedure prologue:

1. The current value of BP is pushed on the stack (to be restored upon return).

2. The new base pointer is set to the current value of the stack pointer.

3. Space is reserved for the local data of the procedure (if any) by reducing the current value of SP by the size of these data.

Then, the executable statements of the procedure body are executed.

Upon termination of the procedure body, the procedure epilogue is executed, performing the following operations:

1. If the procedure has local data, the stack pointer is incremented, to point to the procedure interface, thus discarding the temporary data.

2. The old BP is restored (POP BP).

3. A RET instruction is executed to pass control back to the calling procedure. A FAR or NEAR return is used, according to the type of call that was used to activate the procedure. The parameters and the static link are discarded automatically with the RET instruction.

Return values from functions

A function result is returned as follows, depending on the size of the function type:

1 byte	in BL
2 bytes	in BX
4 bytes	in ES and BX.

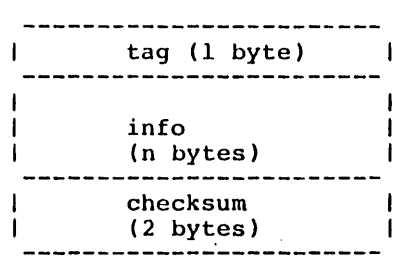
REAL values are passed on top of the stack in the stack of the coprocessor or of the emulator.

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

APPENDIX G - Object File Format

General Format

An object file is a sequence of records, each of which has the following general format :



The "tag" gives the type of the record.

The "info" is the information to be read.

The "checksum" is the sum of all bytes in the file, including the tag and the info fields, but without the checksums. For the computation of the checksum, info is considered as a sequence of bytes. The sum is computed modulo 2**16.

Numerical values on 2 bytes are represented with the lower byte first, values on 3 bytes in the order : low, middle, high.

The Name, file extension and description for each Tag are described below:

0 FormatVersion (.LNK, .LOD)

Defines the version of the object file format and the target system. Used to check compatibility between Compiler- Linker- Loader.

1 ProgramHeader (.LOD)

Gives number of SCModules in this program, total size of code and data.

2 SCModuleHeader (.LNK, .LOD)

Contains name and key of a SCModule. In addition, it gives, for a link-file: size of code and size of data; for a load-file: values of code and data segment, relative to start of program. In both cases it gives also the offset of procedure table (in Filled Data).

3 ImportElement (.LNK, .LOD)

Gives the name and key of an imported SCModule. Assigns an internal number to this element, under which it will be referenced in the fix-up records.

4 FilledData (.LNK)

Gives length and contents of initialized data of a SCModule (string constants, value of DS and procedure table with entries of exported procedures)

5 ProcedureCode (.LNK)

Gives length and contents of the code of a procedure. Assigns a number to the procedure.

6 SCModInitCode (.LNK)

Gives length and contents of module code of a SCModule.

7 ModuleCode (.LOD)

Gives length and content of a complete SCModule (initialized data, module code and all procedure blocks). This allows for fast reading of the contiguous data of one module upon loading.

8 SCModuleCall (.LNK)

Describes a reference to the Module Code of another SCModule. There is exactly one such Call in a link-file; the module codes are called in a chain and the linker has to replace the call in the last SCModule by NOP instructions. This reference is anonymous in the sens that it does not specify which SCModule is called, the order of execution is defined by the linker.

9 RefExtData (.LNK, .LOD)

Describes a reference to data in another SCModule. In Link-files: ref. to any other SCModule In Load-files: ref. to a SCModule in base only. At the described location, the specified DS has to be set.

10 RefExtCode (.LNK, .LOD)

Describes a reference to an object in the code segment of another SCModule. At the described location, the specified CS has to be set. In Link-files: ref. to any other SCModule, in Load-files: ref. to a SCModule of base only.

11 RefExtProc (.LNK, .LOD)

Describes a reference to an external procedure. At the described location, the offset ot the specified procedure has to be set. In Link-files: ref. to a procedure in any other SCModule, in Load-files: ref. to a proc. in the base only.

12 RefOwnData (.LOD)

Describes a reference to the own data segment. Defines the location, where the DS is written in the code segment. This reference is transformed into a RefOwnCod by the linker, which is possible, since the data segments are located immediately after the code segments.

13 RefOwnCode (LOD)

Describes a reference to a segment of the own layer. Since all segments of a program are contiguous and the linker knows their sizes, these. references are partly solved: the offset of the corres- ponding segment, relative to the code segment of the main module is set in the fixup record. At load time, the absolute value of the main CS has to be added to that offset.

14 RefOwnProc (.LNK)

Describes a reference to a procedure in the own SCModule. These references are solved by the linker to simplify the compiler (would require an additional pass for forward ref.).

15 SCModuleEnd (.LNK, .LOD)

Terminates the sequence of records, describing a SCModule. No information- field in this record.

16 ProgramEnd (.LOD)

Terminates a sequence of SCModules, forming one program. No information-field in this record.

Syntax of Object Files

There are two kind of Object Files, which contain both a subset of the record types, defined in the previous chapter :

a) Link Files.

Output of the compiler, input for Linker or Loader. One per separately compiled module.

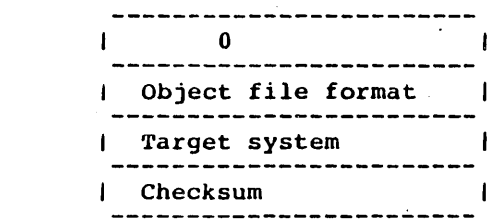
b) Load Files.

Output of Linker, input for Loader. One per program, containing a collection of modules.

Format of the different records

In the following description every box in a diagram stands for one word (two bytes), except the header box that contains the tag. The tag always has a size of just one byte. In special cases the number of bytes represented by a box is indicated.

FormatVersion :

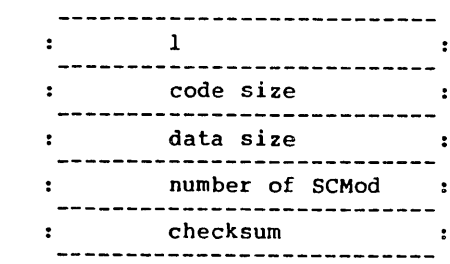


The "object file format" identifies the version of the linker to be used for this object file.

The "target system" is a code that defines the target processor and options.

The fields 'object file format' and 'target system' of the record 'FormatVersion' are of size one byte only.

ProgHeader :



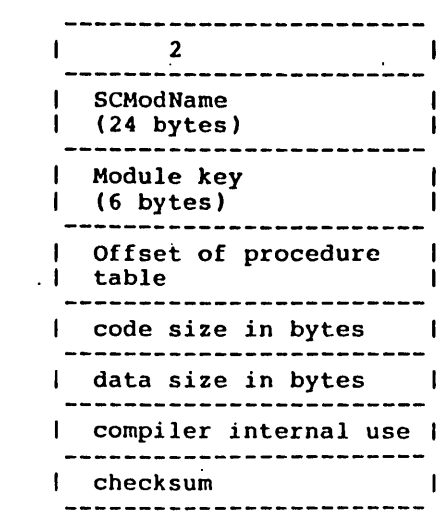
This record gives the size of a linked program. "code size" is the number of paragraphs occupied by code segments of all SCModules, belonging to this program. "data size" is the total number of paragraphs occupied by the data segments. "number of SCMod" is the number of SCModules included in this program.

Note:

1) No entry point of the program is specified. The program starts with procedure 0 of the first module.

2) The codes of all modules have to be loaded as they are encountered in the load file. The room for the data segments has to be reserved immediately after the code.

SCModHeader :



"SCModName" is the name of the module belonging to this link file. A name, shorter than 24 characters is terminated with one Null- character, a longer name is truncated. The "module key" identifies the version of the separate module (unique for every compilation). It is used to guarantee that we link together the same versions that the compiler used for interface-checking.

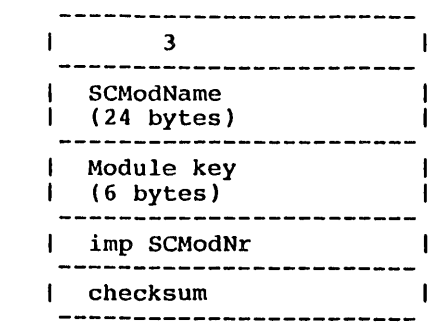
"Offset of Procedure Table" gives the offset in the code segment of a reserved area of 2 bytes per procedure. This procedure table will be filled by the linker with the offsets of the procedures of this module.

"code size" gives the total length of the code segment of this module. It includes the code and all initialized data. This information is redundant in link-files, it allows for consistency checks and for loading unlinked programs.

"data size" gives the total length of the code segment of this module. The data segment contains the variables declared at module level.

The field "internal use" is needed to compensate the checksum of the field "code size", which is filled at the very end.

ImportElement :



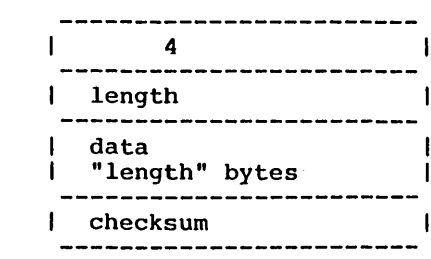
This record describes a SCModule which is imported by this module.

"SCModName" and "Module key" : see SCModHeader.

"imp SCModNr" is the number under which the imported SCModule is referenced in the fixup records.

Note:
In load files only the referenced modules of the base are imported. In order not to leave holes in the numbering, they are given new numbers by the linker. Every module of a load file contains the complete list of the modules imported by itself.

FilledData :

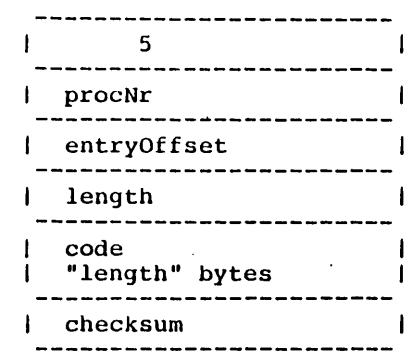


"data" are initialized data that will be loaded in the code segment in the order they are encountered in the link file. They include string constants as well as constants generated by the compiler (value of DS, procedure table, etc).

"length" gives the number of bytes in the field "data".

ProcCode :

This record defines the code of one procedure or the initialization code of an inner module.



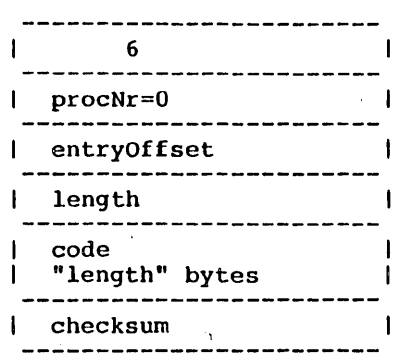
"procNr" is the number under which the procedure is referenced.

"entryOffset" is the entry point of the procedure, relative to the first byte of its code (currently always 0).

"length" is the number of bytes contained in the "code".

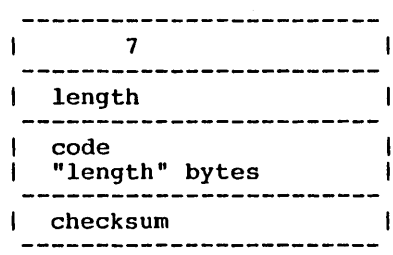
CModInitCode :

his record defines the initialization code of the current CModule.



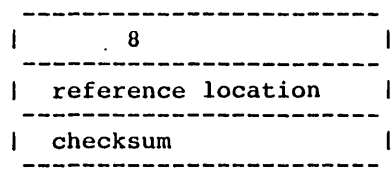
procNr" must be 0. Other fields : see ProcCode.

oduleCode:



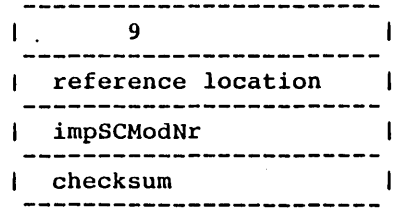
his record contains the entire code segment of a SCModule memory image), including filled data, SCModule init code nd procedure codes. The "length" is rounded up to be a ultiple of 16 (paragraph boundary). The remaining bytes n "code" are filled with zeros.

SCModCall:



Defines a call to the SCModInitCode of an imported module. There is exactly one such record per compilation unit.

RefExtData :



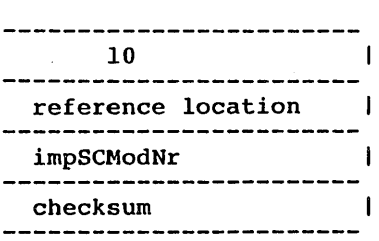
This record describes a reference to the data segment of an imported SCModule. "reference location" is the offset inside the code, where the linker or loader has to put the paragraph address of the corresponding segment. This value will be put at "reference location" (lower byte) and the following byte (higher byte).

"impSCMod" identifies the imported SCModule referenced in this record. Corresponds to the same field in the "ImportElement" record.

In load files, the "RefExt..." records describe references to the base only.

RefExtCode :

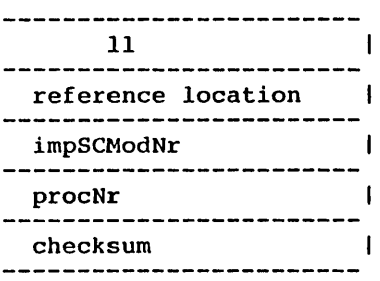
This record describes a reference to the code segment of an imported SCModule.



"reference location" and "impSCModNr" : see RefExtData.

RefExtProc :

This record descibes a reference to a procedure in an imported SCModule.



"reference location" is the offset inside the code, where the linker or locater has to put the offset (relative to the CS) of the procedure entry.

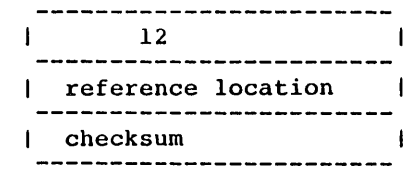
The code segment of that procedure will be set by means of a record of type RefExtCode.

"impSCModNr" identifies the SCModule containing that procedure.

"procNr" identifies the procedure inside the imported SCModule.

If one of these two fields is 0 this record is illegal.

RefOwnData:

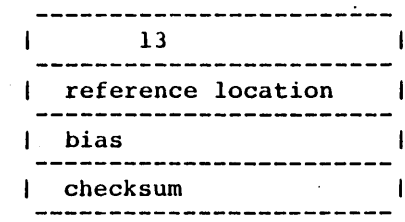


This record describes a reference to the own data segment.

"reference location" is the offset in the code segment where to put the value of the own DS.

There is exactly one such record per link file.

RefOwnCode:

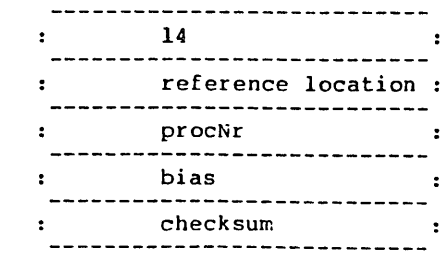


This record describes a reference to the own code segment; in load files only. Used for references to other modules in the same relocatable program.

"bias" is the offset in paragraphs of the referenced code segment, relative to the start of the program.

"reference location" is the offset in the code segment of current module, where the loader has to set the sum of the paragraph address of the loaded program and the "bias".

RefOwnProc :

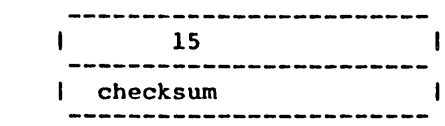


This record describes a reference to a procedure in the current module (= local procedure).

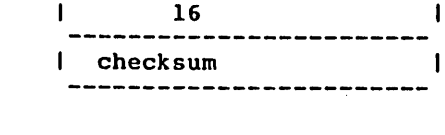
"reference location" is the offset where the linker or loader has to put the offset (relative to the instruction pointer) of the referenced procedure. The value of "bias" has to be substracted from the entry address of "procNr" to get the correct offset (fixup value).

"procNr" identifies the procedure.

SCModuleEnd :



ProgramEnd:



APPENDIX H - DOSCALL

The standard procedure DOSCALL

The procedure DOSCALL must be imported from module SYSTEM. It provides rather simple way to access the underlying operating system from programs written in Modula-2. For the description of each of these functions we refer to the corresponding MS-DOS or PC-DOS Manual. The actual parameters of the procedures should not be too complicated. The compiler could might easily run out of registers.

The first line is a Modula-2 procedure declaration. The second line notes for each parameter the register(s) in which it is passed. The type BYTEWORD (which doesn't exist in Modula-2) means that any type compatible to BYTE or WORD is possible for the actual parameter.

Example:

```
DOSCALL(15; FCBAddr:ADDRESS; VAR returnCode:BYTEWORD);
         AH    DS:DX                      AL
```

possible use:

```
VAR  FCB: ARRAY[0...35] OF CHAR;
     returnVal: CARDINAL

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

The standard procedure DOSCALL has a variable parameter list. This parameter list depends on the first parameter that must be a constant. This constant is the number of the DOS function to be called.

The format of these functions are:

Function 0H: Program Terminate

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

Function 1H: Keyboard Input

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

Function 2H: Display Output

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

Function 3H: Auxiliary Input

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

Function 4H: Auxiliary Output

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

Function 5H: Printer Output

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

Function 6H: Direct Console I/O

```
DOSCALL(6H; OFFH; VAR char:BYTEWORD;
        AH    DL         AL

        VAR ready:BOOLEAN); (input)
             ZF


DOSCALL(6H; char:BYTEWORD); (output)
        AH  DL
```

Function 7H: Direct console Input without echo

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

Function 8H: Console input without echo

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

Function 9H: Print String

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

Function 0AH: Buffered Keyboard input

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

Function 0BH: check standard input status

```
DOSCALL(0BH; VAR status:BYTEWORD);
        AH        AL
```

Function 0CH: Clear standard input buffer and invoke a standard input function

The second parameter (input function) determines the form of the parameter list. It must be one of the constants (functions) 1H, 6H, 7H, 8H or 0AH.

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

DOSCALL(0CH; 6H: VAR char: BYTEWORD);
       AH    AL    AL

       VAR READY: BOOLEAN); (DL : 0FFH implicitly)
       ZF

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

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

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

Function 0DH: Disk reset

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

Function 0EH: Select Disk

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

        VAR returnCode:BYTEWORD);
```

```
                    AL

FUNCTION 0FH: Open File

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

              VAR returnCode:BYTWORD);
                   AL

Function 10H: Close File

      DOSCALL(10H; FCBaddr:ADDRESS;
              AH   DS:DX

              VAR returnCode:BYTEWORD);
                   AL

Function 11H: Search for the first entry

      DOSCALL(11H; FCBaddr:ADDRESS;
              AH   DS:DX

              VAR returnCode:BYTEWORD);
                   AL

Function 12H: Search for the next entry

      DOSCALL(12H; FCBaddr:ADDRESS;
              AH   DS:DX

              VAR returnCode:BYTEWORD);
                   AL

Function 13H: Delete File

      DOSCALL(13H; FCBaddr:ADDRESS;
              AH   DS:DX

              VAR returnCode:BYTEWORD);
                   AL

Function 14H: Sequential Read

      DOSCALL(14H; FCBaddr:ADDRESS;
              AH   DS:DX

              VAR returnCode:BYTEWORD);
                   AL

Function 15H: Sequential Write

      DOSCALL(15H; FCBaddr:ADDRESS;
```

```
              AH    DS:DX

              VAR returnCode:BYTEWORD);
                    AL

Function 16H: Create File

      DOSCALL(16H; FCBaddr:ADDRESS;
              AH    DS:DX

              VAR returnCode:BYTEWORD);
                    AL

Function 17H: Rename File

      DOSCALL(17H; FCBaddr:ADDRESS;
              AH    DS:DX

              VAR returnCode:BYTEWORD);
                    AL

Function 19H: Current Disk

      DOSCALL(19H; VAR currDrive:BYTEWORD);
              AH           AL


Function 1AH: Set Disk Transfer Address

      DOSCALL(1AH; DTA:ADDRESS);
              AH    DS:DX


Function 1BH: Allocation table information

      DOSCALL(1BH; VAR FATaddr:ADDRESS;
              AH         DS:BX

              VAR nrallocUnits, nrSectors,
                    DX                AL

              sectSize:BYTEWORD);
                CX

Function 1CH: (not implemented)


Function 21H: Random Read

      DOSCALL(21H; FCBaddr:ADDRESS;
              AH    DS:DX

              VAR returnCode:BYTEWORD);
```

```
                    AL

Function 22H: Random Write

      DOSCALL(22H; FCBaddr:ADDRESS;
              AH   DS:DX

              VAR returnCode:BYTEWORD);
                     AL

Function 23H: File Size

      DOSCALL(23H; FCBaddr:ADDRESS;
              AH   DS:DX
              VAR returnCode:BYTEWORD);
                     AL

Function 24H: Set Random Record Field

      DOSCALL(24H; FCBaddr:ADDRESS);
              AH   DS:DX


Function 25H: Set Interrupt Vector

      DOSCALL(25H; vectorVal:ADDRESS; INTtype:BYTEWORD);
              AH   DS:DX                AL


Function 26H: Create a new program segment

      DOSCALL(26H; progSegment:BYTEWORD);
              AH       DX



Function 27H: Random Block Read

      DOSCALL(27H; FCBaddr:ADDRESS;
              AH   DS:DX

              VAR nrofBytes:BYTEWORD;
                     CX

              VAR returnCode:BYTEWORD);
                     AL


Function 28H: Random Block Read

      DOSCALL(28H; FCBaddr:ADDRESS;
              AH   DS:DX
```

```
              VAR nrofBytes:BYTEWORD;
                     CX

              VAR returnCode:BYTEWORD);
                     AL


Function 29H: Parse FilenameG

      DOSCALL(29H; FCBaddr:ADDRESS; mode:BYTEWORD;
              AH   ES:DI             AL

              VAR stringaddr:ADDRESS;
                  DS:SI

              VAR returnCode:BYTEWORD);
                     AL


Function 2AH: Get Date

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


Function 2BH: Set Date

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

              VAR returnCode:BYTEWORD);
                     AL

Function 2CH: Get Time

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


Function 2DH: Set Time

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

              VAR returnCode:BYTEWORD);
                     AL


Function 2EH: Set/Reset Verify Switch

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

Extension for DOS 2.0

Function 2FH: Get DTA

```
DOSCALL(2FH; VAR DTAaddr:ADDRYTEWORD;
        AH   AL              DX

        paragraphs:WORD);
```

Function 33H: Ctrl-Break-Check

```
DOSCALL(33H; mode:BYTEWORD; VAR state:BYTE);
        AH   AL                DL
```

Function 35H: Get Vector

```
DOSCALL(35H; vector:BYTEWORD; VAR vector:ADDRESS
        AH   AL                   ES:BX
```

Function 36H: Get disk free space

```
DOSCALL(36H; drive:BYTEWORD; VAR valid:BYTEWORD;
        AH   DL                  AX

        VAR availClusters:BYTEWORD;
            BX

        VAR totclust:BYTEWORD;
            DX

        VAR bytesPerSect:BYTEWORD);
            CX
```

Function 38H: Return Country dependent information

```
DOSCALL(38H; buffAddr:ADDRESS; fctcode:BYTEWORD);
             DS:DX
```

Function 39H: Create a sub-directory (MKDIR)

```
DOSCALL(39H; stringaddr:ADDRESS;
             DS:DX

             VAR error:WORD); error=0 no error
                 AX,CF        other values see:
                            table in DOS manual
```

Function 3AH: Remove a directory entry (RMDIR)

```
DOSCALL(3AH; stringaddr:ADDRESS; VAR error:WORD);
        AH   DS:DX                    AX,CF
```

Function 3BH: Change the current directory (CHDIR)

```
DOSCALL(3BH; stringaddr:ADDRESS; VAR error:WORD);
        AH   DS:DX                    AX,CF
```

Function 3CH: Create a File

```
DOSCALL(3CH; stringaddr:ADDRESS; attrib:BYTEWORD;
        AH   DS:DX               CX

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

Function 3DH: Open a File

```
DOSCALL(3DH; stringaddr:ADDRESS; access:BYTEWORD;
        AH   DS:DX               AL

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

Function 3EH: Close a file handle

```
DOSCALL(3EH; handle:WORD; VAR error:WORD);
             BX               AX,CF
```

Function 3FH: Read from a file or device

```
DOSCALL(3FH; handle:WORD; nrbytes:WORD;
             BX           CX

             buffAddr:ADDRESS;
             DS:DX

             VAR readBytes:BYTEWORD;
                 AX

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

Function 40H: Write to a file or device

```
DOSCALL(40H; handle:WORD; nrbytes:WORD;
```

```
            AH    BX            CX

            buffAddr:ADDRESS;
            DS:DX

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

Function 41H: Delete a file from a specified directory

```
      DOSCALL(41H; stringaddr:ADDRESS; VAR error:WORD);
              AH   DS:DX                  AX,CF
```

Function 42H: Move file read/write pointer

```
      DOSCALL(42H; handle:WORD; method:BYTEWORD;
              AH    BX            AL

              inHigh,inLow:WORD;
                CX     DX

              VAR outHigh,outLow:WORD; VAR error:WORD);
                   DX       AX             AX,CF
```

Function 43H: Change File Mode

```
      DOSCALL(43H; stringaddr:ADDRESS; fctcode:BYTEWORD;
              AH   DS:DX               AL

              VAR mode:BYTEWORD; VAR error:WORD);
                  CX                 AX,CF
```

Function 44H: I/O control for devices

The procedure depends on the value of the second parameter that must be a constant. This parameter determines the function to execute:

Get device info:

```
      DOSCALL(44H; 0; handle:WORD;
              AH   AL BX

              VAR device info:BYTEWORD;
                  DX

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

Set device info:

```
DOSCALL(44H; 1; handle:WORD;
        AH   AL BX

        VAR device info:BYTEWORD;
            DX

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

Read Bytes from device control channel

```
DOSCALL(44H; 2; handle:WORD;
        AH   AL   BX

        nrBytes:BYTEWORD; buffAddr:ADDRESS;
          CX                DS:DX

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

Write Bytes to device control channel

```
DOSCALL(44H; 3; handle:WORD; nrBytes:BYTEWORD;
        AH   AL   BX           CX

        buffAddr:ADDRESS;
         DS:DX

        VAR transferedbytes:BYTEWORD;
             AX

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

Read Bytes from drive control channel

```
DOSCALL(44H; 4; drive:BYTEWORD; nrBytes:BYTEWORD;
        AH   AL   BX              CX

        buffAddr:ADDRESS;
         DS:DX

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

Write Bytes to drive control channel

```
DOSCALL(44H; 5; drive:BYTEWORD; nrBytes:BYTEWORD;
        AH   AL   BX              CX

        buffAddr:ADDRESS;
         DS:DX

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

Get Input Status

```
DOSCALL(44H; 6; VAR status:BYTEWORD; VAR error:WORD);
        AH  AL       AX                  AX,CF
```

Get Output Status

```
DOSCALL(44H; 7; VAR status:BYTEWORD; VAR error:WORD);
        AH   AL     AX                  AX,CF
```

Function 45H: Duplicate a file handle

```
DOSCALL(45H; handle1:WORD; VAR handle2:BYTEWORD;
        AH    BX                AX

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

Function 46H: Force a duplicate of a file

```
DOSCALL(46H; handle1:WORD; VAR handle2:BYTEWORD;
        AH    BX                AX

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

Function 47H: Get Current Directory

```
DOSCALL(47H; drive:BYTEWORD; straddr:ADDRESS;
        AH   DL              DS:SI

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

Function 48H: Allocate Memory

```
DOSCALL(48H; VAR paragraphs:BYTEWORD;
        AH       BX
```

```
              VAR membase:BYTEWORD;
                   AX

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

Function 49H: Free allocated Memory

```
     DOSCALL(49H: segaddr:ADDRES;VAR error:WORD);
              AH   ES must be a      AX,CF
                      paragraph address
```

Function 4AH: SETBLOCK-Modify allocated memory blocks

```
     DOSCALL(4AH; blockaddr:ADDRESS;
             AH   ES must be a paragr
                     address

             VAR parzVagraphs:BYTEWORD;
             aph   BX

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

Function 4BH: Load or execute a program

```
     DOSCALL(4BH; stringaddr:ADDRESS; paramblock:ADDRESS;
             AH    DS:DX                ES:BX

             fctval:BYTEWORD;
              AL

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

Function 4CH: Terminate a process(Exit)

```
     DOSCALL(4CH; returnCode:BYTEWORD);
             AH       AL
```

Function 4DH: Retrieve the return code of a sub-process(Wait)

```
     DOSCALL(4DH; VAR retCode:BYTEWORD);
             AH        AX
```

Function 4EH: Find first matching file

```
DOSCALL(4EH; stringaddr:ADDRESS; attribut:BYTEWORD;
        AH    DS:DX                    CX

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

Function 4FH: Find next matching file

```
DOSCALL(4FH; VAR error:WORD);
         AH       AX,CF
```

Function 54H: Get Verify state

```
DOSCALL(54H; VAR state:BYTE);
        AH           AL
```

Function 56H: Rename a file

```
DOSCALL(56H; fromstring,tostring:ADDRESS;
        AH    DS:DX       ES:DI

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

Function 57H: Get/Set a file's date and time

```
DOSCALL(57H; handle:WORD; mode:BYTEWORD;
        AH     BX           AL

        VAR date,time:BYTEWORD;
            DX    CX

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

APPENDIX I - System Configuration

Configuration for display and keyboard

The Modula-2/86 system comes with modules 'Display' and 'Keyboard' configured for the standard IBM PC and DOS version 2. If this is your system configuration, then there is no need for any adaption. If your system configuration is different, you may adapt these modules. In general, these differences should not affect the operation of the compiler and linker very much.

For this purpose you have received with your Modula-2/86 system the sources of these two modules. Look on the system diskette for the files 'DISPLAY.*' and 'KEYBOARD.*'. You should find there the definition modules (extension DEF) as well as the implementations (extension MOD). Study both of them carefully.

When you know which changes you need to make, take a copy of the implementation module(s) you want to change. After you have made the changes, recompile the implementation module(s). The compiler should find the necessary symbol files as usual. Do not change or recompile the definition modules because this would produce a new symbol file and would introduce incompatibilities with other parts of the Modula-2/86 system.

After this you can replace the old link file(s) (extension LNK) with the new ones. Do not destroy the original link file(s)! Keep a copy of the original(s) in case something has gone wrong. If you have a hard disk and you are using the recommended disk organization, you should put the new link files into the directory 'M2LIB

Now use a small test program that reads from the keyboard and writes to the screen in order to check out the modified version(s). The test program should not import the modules 'Display' or 'Keyboard' directly. It can use the regular read and write procedures from module 'Terminal' or 'InOut'. These modules perform reading and writing through 'Keyboard' and 'Display' respectively.

If you want any existing program to use the new keyboard and/or display implementation you have to re-link these programs. This is also true for the compiler, linker and debugger of the Modula-2/86 system. The debugger in particular uses some of the special codes as defined in the display module.

Re-link these programs only when your are sure that your new version works fine. Keep a backup copy of the files 'COMP.LOD' (on the compiler diskette), 'LINK.LOD' and 'DBUG.LOD' (on the linker/debugger diskette). Your system diskette contains the link files 'COMP.LNK', 'LINK.LNK' and 'DBUG.LNK' that are needed for re-linking. Take a copy of these files and re-link them in the ordinary way. Try the new compiler, linker and debugger.

When everything is fine, you can replace the old versions of 'COMP.LOD', 'LINK.LOD' and 'DBUG.LOD' with the new ones. If your system has a hard disk and you are using the recommended disk organization, then you should copy the new load files to the directory 'M2LOD'.

Configuration of the compiler user interface

The Modula-2/86 system allows you to configure the compiler for the default settings of the compiler options. It is strongly recommended to get first familiar with the compiler and its options, before attempting this step. The section on compiler options explains the use and the initial default settings of these options.

If you want to change the default settings of the compiler options, you have to modify the implementation of the compiler module 'CompPara'. Check the system diskette for the files 'COMPPARA.*'. You can also find a copy of the definition module 'CompPara' at the end of this appendix. Study the definition and implementation modules.

When you have decided what to change, take a copy of the implementation module 'COMPPARA.MOD'. Do not destroy the original version. Also, you must not change or recompile the definition module. This would produce a new symbol file and introduce incompatibilities with other parts of the compiler. You would not be able to re-link the compiler properly because of version conflict errors. Change and recompile only the implementation module. For the recompilation you need to take a copy of the symbol file 'COMPPARA.SYM', which is also contained on the system diskette. Copy it to the diskette or directory where you already have a copy of 'COMPPARA.MOD'.

After successfull compilation of the new implementation module of 'CompPara', you must relink the compiler overlay 'M2COMP'. You will find all the necessary map and link files on the system diskette. Copy them to the diskette or directory where you have already the link file

'COMPPARA.LNK'. Copy the following files:

- COMP.MAP
- COMPFILE.LNK
- PUBLIC.LNK
- M2COMP.LNK

Now you can re-link 'M2COMP'. You must link it as an overlay to 'COMP'. See the chapter on the linker for information on linking overlays. Specify 'M2COMP' as master file to be linked, and use the linker option '/B' to indicate that you are linking an overlay. Specify 'COMP.MAP' as the base file for the linkage.

Be sure to keep a copy of the old version of 'M2COMP.LOD' before you replace it with the new one. If you are working with diskettes make first a copy of your compiler diskette and make the replacement and the testing with the copy. Now you can replace the old version of 'M2COMP.LOD' with the new one. If your system has a hard disk and you are using the recommended disk organization, you should copy the new load file 'M2COMP.LOD' to the directory 'M2LOD'.

The library module CompPara

```
DEFINITION MODULE CompPara;

  EXPORT QUALIFIED
    CPpageLength, CPpageWidth,      listing definition
    CPffAtEnd, CPffAtBegin,
    CPheader, CPdate,
    CPfooter, CPfooterText,
    CPpriorityLevels,               interrupt system
                                    definition: 8259A
    CPRTSfunctVector,               interrupt vector
                                    to access RTS
    CPemulator,                     default settings
                                    of compiler options
    CPinteractiv,
    CPquery, CPautoquery,
    CPdebug, CPversion,
    CPlister, CPerrorLister,
    CPafterPassl, CPafterPass2,     time of listing gen.
    CPstacktest, CPrangetest,       default settings of
                                    program-source options
    CParithmetictest;

  VAR

     the following variables are used to define the
     format of the listing generated by the compiler:

 CPpageLength,
     number of lines per page       : initial value is 60
     valid range: 40..65535 (if out of range: 60 is taken)

 CPpageWidth: CARDINAL;
     number of characters per line  : initial value is 79
     valid range: 50..150  (if out of range: 79 is taken).

     the next two parameters are used to define the
     page eject

 CPffAtEnd: BOOLEAN;
     defines whether a formfeed at last character
     is generated or not: initial value is TRUE

 CPffAtBegin: BOOLEAN;
     defines whether a formfeed at first character
     is generated or not: initial value is FALSE

     the next two parameters define the header of
     each page

 CPheader: BOOLEAN;
```

```
    defines whether a headerline is generated on each
    page or not: initial value is TRUE

 CPdate: BOOLEAN;
    defines whether date in headerline is generated
    or not: initial value is FALSE


    a header line has the following format:
Modula-2/86      filename.ext    Date         Page n
    e.g.
Modula-2/86      COMPPARA.DEF    Nov 16'83    Page 1


    the next two parameters define the footer of each
    page

  CPfooter: BOOLEAN;
    defines whether a footerline is generated on each
    page or not: initial value is FALSE

  CPfooterText: ARRAY [0..149] OF CHAR;
    string of pagewidth characters that defines the
      footerline
    normally used for Copyright text: initially an
      empty string


    definition of interrupt system: number of priority
    levels

  CPpriorityLevels: CARDINAL;
    initial value of distributed version is 8:
    defined by the (8259A) interrupt controller(s)


    definition of interface to Run-Time-Support system

  CPRTSfunctVector: CARDINAL;
    defines the interrupt vector to acces the RTS
    RTS: Runtime Support: Assembley part of Modula-2/86
    System normally 228: may be only changed if RTS
    is changed


    the following boolean variable defines whether
    code for an 8087 coprocessoror an 8087 emulator
    is generated by the compiler

  CPemulator: BOOLEAN;    initial value is FALSE

    the following options define the interactive
    behavior of the compiler
```

```
CPinteractiv: BOOLEAN;     initial value is TRUE
   whether compiler may be stopped by typing a key

CPquery: BOOLEAN;     initial value is FALSE
   whether compiler asks for the symbol files of
   the imported modules or tries to find them by
   the default strategy:
   build filename from module name by taking
   the 8 first characters and the extension 'SYM'

CPautoquery: BOOLEAN;     initial value is TRUE
   whether compiler falls automatically in query mode
   if it didn't find the file by the default mechanism

CPdebug: BOOLEAN;     initial value is TRUE
   whether compiler generates a reference file
   that would be used by the debugger, or not

CPversion: BOOLEAN;     initial value is FALSE
   whether compiler displays version info or not

CPlister: BOOLEAN;     initial value is FALSE
   whether compiler generates a listing file or not

CPerrorLister: BOOLEAN;     initial value is TRUE
   whether compiler generates automatically an
   error listing if errors occured or not

   moment of listing generation in case of errors

   if both variables are set to FALSE the listing
   will be generated after pass3
   the functions of the different passes are:
   pass1 checks syntactic of program
   pass2 checks declaration parts (allocation)
   pass3 checks bodys (compatibility test)


CPafterPass1: BOOLEAN;     initial value: TRUE
   whether compiler goes to lister if error detected
   in pass1 and terminates compilation, or not

CPafterPass2: BOOLEAN;     initial value: FALSE
   whether compiler goes to lister if error detected
   in pass2 and terminates compilation, or not

   the following boolean variables are used to define
   the default setting of the corresponding testcode
   options:

   TRUE means:  default is '+';
   FALSE means: default is '-'
```

```
  CPstacktest: BOOLEAN;
     for option 'S': initial value is TRUE

  CPrangetest: BOOLEAN;
     for option 'R': initial value is TRUE

  CParithmetictest: BOOLEAN;
     for option 'T': initial value is TRUE

END CompPara.
```

1. APPENDIX J - Library definitions

```
DEFINITION MODULE ASCII;

   Symbolic constants for non-printing ASCII characters

EXPORT QUALIFIED

   nul, soh, stx, etx, eot, enq, ack, bel,
   bs,  ht,  lf,  vt,  ff,  cr,  so,  si,
   dle, dc1, dc2, dc3, dc4, nak, syn, etb,
   can, em,  sub, esc, fs,  gs,  rs,  us,
   del;

CONST

   nul = 00C;  soh = 01C;  stx = 02C;  etx = 03C;
   eot = 04C;  enq = 05C;  ack = 06C;  bel = 07C;
   bs  = 10C;  ht  = 11C;  lf  = 12C;  vt  = 13C;
   ff  = 14C;  cr  = 15C;  so  = 16C;  si  = 17C;
   dle = 20C;  dc1 = 21C;  dc2 = 22C;  dc3 = 23C;
   dc4 = 24C;  nak = 25C;  syn = 26C;  etb = 27C;
   can = 30C;  em  = 31C;  sub = 32C;  esc = 33C;
   fs  = 34C;  gs  = 35C;  rs  = 36C;  us  = 37C;
   del = 177C;

END ASCII.
```

```
DEFINITION MODULE CardinalIO;

  Terminal input/output of CARDINALs in decimal and hex

Derived from the Lilith Modula-2 system developed by the
group of Prof. N. Wirth at ETH Zurich, Switzerland.

EXPORT QUALIFIED

   ReadCardinal, WriteCardinal, ReadHex, WriteHex;
```

```
PROCEDURE ReadCardinal (VAR c: CARDINAL);

- Read an unsigned decimal number from the terminal.

out:    c       the value that was read.

The read terminates only on ESC, EOL, or blank, and the
terminator must be re-read, for example with
Terminal.Read.

If the read encounters a non-digit, or a digit which would
cause the number to exceed the maximum CARDINAL value, the
bell is sounded and that character is ignored. No more
than one leading '0' is allowed.
```

```
PROCEDURE WriteCardinal (c: CARDINAL; w: CARDINAL);

- Write a CARDINAL in decimal format to the terminal.

in:     c       value to write,
        w       minimum field width.

The value of c is written, even if it takes more than w
digits. If it takes fewer digits, leading blanks are out-
put to make the field w characters wide.
```

```
PROCEDURE ReadHex (VAR c: CARDINAL);
```

- Read a CARDINAL in hexadecimal format from the terminal. [see ReadCardinal above]

```
PROCEDURE WriteHex (c: CARDINAL; digits: CARDINAL);
```

- Write a CARDINAL in hexadecimal format to the terminal. [see WriteCardinal above]

```
END CardinalIO.
```

```
DEFINITION MODULE Conversions;

   Convert from INTEGER and CARDINAL to string
```

Derived from the Lilith Modula-2 system developed by the group of Prof. N. Wirth at ETH Zurich, Switzerland.

```
EXPORT QUALIFIED
  ConvertOctal, ConvertHex,
  ConvertCardinal, ConvertInteger;
```

```
PROCEDURE
  ConvertOctal(num, len:  CARDINAL;
               VAR str: ARRAY OF CHAR);
```

- Convert number to right-justified octal representation

```
in:     num     value to be represented,
        len     minimum   width   of   representation,
out:    str     result string.
```

If the representation of 'num' uses fewer than 'len' digits, blanks are added on the left. If the representation will not fit in 'str', it is truncated on the right.

```
PROCEDURE ConvertHex(num, len:  CARDINAL;
                     VAR str:  ARRAY OF CHAR);
```

- Convert number to right-justified hexadecimal representation. [see ConvertOctal]

```
PROCEDURE
  ConvertCardinal(num, len: CARDINAL;
                  VAR str: ARRAY OF CHAR);
```

- Convert a CARDINAL to right-justified decimal

representation. [see ConvertOctal]

```
PROCEDURE
  ConvertInteger(num: INTEGER; len: CARDINAL;
                 VAR str: ARRAY OF CHAR);
```

- Convert an INTEGER to right-justified decimal representation. [see ConvertOctal] Note that a leading '-' is generated if num < 0, but never a '+'.

```
END Conversions.
```

```
DEFINITION MODULE DiskDirectory;

   Interface to directory functions of the underlying OS

Derived from the Lilith Modula-2 system developed  by  the
group of Prof. N. Wirth at ETH Zurich, Switzerland.


EXPORT QUALIFIED
    CurrentDrive, SelectDrive,
    CurrentDirectory, ChangeDirectory,
    MakeDir, RemoveDir,
    ResetDiskSys, ResetDrive;
```

```
PROCEDURE CurrentDrive (VAR drive: CHAR);

- Returns the current default drive.

out:  drive  the default drive, given in character format.
```

```
PROCEDURE SelectDrive (drive: CHAR; VAR done: BOOLEAN);

- Set default drive.

in:   drive   name of drive to make default, specified
              in char format.

out:  done    TRUE if operation was successful.

The default drive will be used by all  routines  referring
to DK:  .


PROCEDURE
  CurrentDirectory (drive: CHAR;
                    VAR dir: ARRAY OF CHAR);

- Gets the current directory for the specified drive.
```

```
in:     drive   0C for the current drive,
                1C for drive "A", etc.

out:    dir     current directory for that drive.
```

Under DOS 1.1, dir[0] will be set to nul (0C).

```
PROCEDURE
  ChangeDirectory (dir: ARRAY OF CHAR;
                   VAR done: BOOLEAN);
```

- Set the current directory

```
in:     dir     drive and directory path name.

out:    done    TRUE if successful; FALSE if the
                directory does not exist.
```

Under DOS 1.1, this function has no effect and 'done' is FALSE.

```
PROCEDURE MakeDir (dir: ARRAY OF CHAR;
                   VAR done: BOOLEAN);
```

- Create a sub-directory

```
in:     dir     drive, optional pathname and name of
                sub-directory to create.

out:    done    TRUE if successful; FALSE if path or
                drive does not exist.
```

Under DOS 1.1, this function has no effect and 'done' is FALSE.

```
PROCEDURE RemoveDir (dir: ARRAY OF CHAR;
                     VAR done: BOOLEAN);
```

- Remove a directory

in: dir drive and name of the sub-directory to remove.

out: done: TRUE if successful; FALSE if directory does not exist.

The specified directory must be empty or the procedure returns FALSE. Under DOS 1.1, this function has no effect and 'done' is FALSE.

```
PROCEDURE ResetDiskSys;
  - MS-DOS disk reset
```

```
PROCEDURE ResetDrive (d: CHAR): CARDINAL;
```

- This function has no effect and always returns 255. It is part of this definition module for reasons of compatibility with other implementations.

```
END DiskDirectory.
```

```
DEFINITION MODULE DiskFiles;
```

Interface to disk file functions of the underlying OS.

Derived from the Lilith Modula-2 system developed by the group of Prof. N. Wirth at ETH Zurich, Switzerland.

```
FROM FileSystem IMPORT File;

EXPORT QUALIFIED  InitDiskSystem,                    Disk-
FileProc, DiskDirProc;
```

```
PROCEDURE InitDiskSystem;
```

- Initialize mediums for further disk file operations

This procedure has to be imported by FileSystem. This has the side-effect, that this module is referenced and will therefore be linked to the user program.

```
PROCEDURE DiskFilePROC (VAR f: File);
```

- low-level interface for disk operations within a file

This procedure is passed as a parameter to the procedure CreateMedium in FileSystem.

```
PROCEDURE DiskDirProc (VAR f: File;
                       name: ARRAY OF CHAR);
```

- low-level interface for disk operations within a directory

This procedure is passed as a parameter to the procedure CreateMedium in FileSystem.

```
END DiskFiles.
```

```
DEFINITION MODULE Display;

   Low-level Console Output

Derived from the Lilith Modula-2 system developed by the
group of Prof. N. Wirth at ETH Zurich, Switzerland.
[Private module of the Modula-2 system]

EXPORT QUALIFIED Write;
```

```
PROCEDURE Write (ch: CHAR);

- Display a character on the console.

in:     ch      character to be displayed.

The following codes are interpreted:
   System.EOL (36C) = go to beginning of next line
   ASCII.ff   (14C) = clear screen and set cursor home
   ASCII.del (177C) = erase the last character on the
                      left
   ASCII.bs   (10C) = move 1 character to the left
   ASCII.cr   (15C) = go to beginning of current line
   ASCII.lf   (12C) = move 1 line down, same column

Write uses direct console I/O.

END Display.
```

```
DEFINITION MODULE FileMessage;

   Write file status/response to the terminal

FROM FileSystem IMPORT Response;

EXPORT QUALIFIED WriteResponse;
```

```
PROCEDURE WriteResponse (r: Response);
```

- Write a short description of a FileSystem response on the terminal.

in: r the response from some FileSystem operation.

The actual argument for 'r' is typically the field message is up to 32 characters long.

```
END FileMessage.
```

```
DEFINITION MODULE FileNames;

   Read a file specification from the terminal.

Derived from the Lilith Modula-2 system developed by the
group of Prof. N. Wirth at ETH Zurich, Switzerland.


EXPORT QUALIFIED  FNParts, FNPartSet, ReadFileName;

TYPE FNParts = (FNDrive, FNPath, FNName, FNExt);
     FNPartSet = SET OF FNParts;
```

```
PROCEDURE
  ReadFileName(VAR resultFN: ARRAY OF CHAR;
               defaultFN: ARRAY OF CHAR;
               VAR ReadInName: FNPartSet);

  - Read a file specification from terminal.

in:    defaultFN     default file specification,

out:   resultFN      the specification that was accepted.
       ReadInName    which parts are in specification

Reads until a <cr>, blank, <can>, or <esc> is typed.
After a call to ReadFileName, Terminal.Read must be called
to read the termination character.  The format of the
specifications depends on the host operating system.


END FileNames.
```

```
DEFINITION MODULE FileSystem;

File manipulation routines

Derived from the Modula-2 system developed by the group of
Prof. N. Wirth, ETH Zurich, Switzerland.


FROM SYSTEM IMPORT ADDRESS, WORD;

EXPORT QUALIFIED
       file operations:
    File, Response, Command,
    Create, Close, Lookup, Rename, Delete,
    SetRead, SetWrite, SetModify, SetOpen,
    Doio, SetPos, GetPos, Length,

       Streamlike I/O:
    Flag, FlagSet,
    Reset, Again,
    ReadWord, ReadChar, ReadByte, ReadNBytes,
    WriteWord, WriteChar, WriteByte, WriteNBytes,

       Medium Handling:
    FileProc, DirectoryProc,
    MediumType, CreateMedium, RemoveMedium,

    FileNameChar;


TYPE MediumHint = CARDINAL;
       - medium index used in DiskFiles

     MediumType = ARRAY [0..2] OF CHAR;
       - medium name (A, B...)

     Flag     = (er, ef, rd, wr, ag, txt);
       - status flag for file operations

er = error occured, ef = end-of-file reached, rd = in read
mode,  wr  =  in  write mode, ag = "Again" has been called
after last read, txt = text-file (the last access  to  the
file was a 'WriteChar' or 'ReadChar').


     FlagSet  = SET OF Flag;
       - status flag set

     Response = (done, notdone, notsupported,
                 callerror, unknownmedium,
                 unknownfile, paramerror,
                 toomanyfiles, eom,
```

```
            userdeverror);
  - result of a file operation

Command   = (create, close, lookup, rename, delete,
             setread, setwrite, setmodify,
             setopen, doio, setpos, getpos,
             length);
  - commands passed to DiskFiles

BuffAdd   = POINTER TO ARRAY [0..0FFFEH] OF CHAR;
  - file buffer pointer type

File      = RECORD
              bufa: BuffAdd;
                Buffer Address
              buflength: CARDINAL;
                size of buffer in bytes. In the
                current release
                it is always a multiple of 128
              validlength: CARDINAL;
                Number of valid bytes in the buffer.
              bufind: CARDINAL;
                Byte-Index of current position in
                the buffer
              flags: FlagSet;   status of the file
              eof: BOOLEAN;
                TRUE, if last access was past end of
                file
              res: Response;
                result of last operation
              lastRead:CARDINAL;
                the last read word or byte (char)
              mt: MediumType;
                selects the driver that supports
                that file
              fHint: CARDINAL;
                internally used by the device-driver
              mHint: MediumHint;
                internally used by medium-handler
              CASE com: Command OF
                  lookup:  new: BOOLEAN;
                | setpos, getpos, length:  highpos,
                  lowpos: CARDINAL;
              END;
            END;
  - file structure used for bookkeeping
    by DiskFiles
```

```
PROCEDURE
  Create (VAR f: File; mediumName: ARRAY OF CHAR);
```

- create a temporary file

in: mediumName name of medium to create file on, in char format

out: f initialized file structure

A temporary file is characterised by an empty name. To make the file permanent, it has to be renamed with a non-empty name before closing it. For subsequent operations on this file, it is referenced by "f".

```
PROCEDURE Close (VAR f: File);
```

- Close a file

in: f structure referencing an open file

out: f the field f.res will be set appropriately.

Terminates the operations on file "f". If "f" is a temporary file, it will be destroyed, whereas a file with a non-empty name remains on its medium and is accessible through "Lookup". When closing a text-file after writing, the end-of-file code 32C is written on the file (MS-DOS and CP/M convention).

```
PROCEDURE Lookup (VAR f: File; filename: ARRAY OF CHAR;
                  newFile: BOOLEAN);
```

- look for a file

in: filename drive and name of file to search for
newFile TRUE if file should be created if not found

out: f initialized file structure; f.res will be set appropriately.

Searches the medium specified in "filename" for a file

that matches the name and type given in "filename". If the file is not found and "newFile" is TRUE, a new (permanent) file with the given name and type is created. If it is not found and "newFile" is FALSE, no action takes place and "notdone" is returned in the result field of "f".

```
PROCEDURE Rename (VAR f: File;
                  newname: ARRAY OF CHAR);
```

- rename a file

in:	f	structure referencing an open file
	newname	filename to rename to, with device: name.type specified
out:	f	file name in f will be changed and the f.res field will be set appropriately.

The medium on which the files reside can not be changed with this command. The medium name inside "newname" has to be the old one.

```
PROCEDURE Delete (name: ARRAY OF CHAR;
                  VAR f: File);
```

- delete a file

in:	name	name of file to delete, with dev:name.type specified
out:	f	the result field f.res will be set appropriately.

```
PROCEDURE ReadWord (VAR f: File; VAR w: WORD);
```

- Returns the word at the current position in f

```
in:     f       structure referencing an open file

out:    w       word read from file
        f       the result field f.res will be set
                appropriately.
```

the file will be positioned at the next word when the read is done.

```
PROCEDURE WriteWord (VAR f: File; w: WORD);
```

- Write one word to a file

```
in:     f       structure referencing an open file
        w       word to write

out:    f       the field f.res will be set appropriately.
```

```
PROCEDURE ReadChar (VAR f: File; VAR ch: CHAR);
```

- Read one character from a file

```
in:     f       structure referencing an open file

out:    ch      charatcter read from file
        f       the result field f.res will be set
                appropriately.
```

the file will be positioned at the next character when the read is done.

```
PROCEDURE WriteChar (VAR f: File; ch: CHAR);
```

- Write one character to a file

```
in:     f       structure referencing an open file
        ch      character to write
```

```
out:    f       the result field f.res will be set
                apporopriately.
```

```
PROCEDURE ReadByte (VAR f: File; VAR b: CHAR);

  - Read one byte from a file

in:     f       structure referencing an open file

out:    b       byte read from file
        f       the result field f.res will be set
                appropriately.

the file will be positioned at the next byte when the rea
is completed.
```

```
PROCEDURE WriteByte (VAR f: File; b: CHAR);

  - Write one byte to a file

in:     f       structure referencing an open file
        b       byte to write

out:    f       the result field f.res will be set
                appropriately.
```

```
PROCEDURE
  ReadNBytes (VAR f: File; bufPtr: ADDRESS;
              requestedBytes: CARDINAL;
              VAR read: CARDINAL);

-Read a specified number of bytes from a file

in:     f               structure referencing an open
                        file
        bufPtr          pointer to buffer area to read
                        bytes into
        requestedBytes  number of bytes to read
```

```
out:    bufPtr^         bytes read from file
        f               the result field f.res will be
                        set appropriately.
        read            the number of bytes actually read
```

the file will be positioned at the next byte after the requested string.

```
PROCEDURE
  WriteNBytes (VAR f: File; bufPtr: ADDRESS;
               requestedBytes: CARDINAL;
               VAR written: CARDINAL);
```

- Write a specified number of bytes to a file

```
in:     f               structure referencing an open file
        bufPtr          pointer to string of bytes to write
        requestedBytes  number of bytes to write

out:    f               the result field f.res will be
                        set appropriately.
        written         the number of bytes actually written
```

```
PROCEDURE Again (VAR f: File);
```

- returns a character to the buffer to be read again

```
in:     f       structure referencing an open file

out:    f       the f.res field will be set appropriately.
```

This should be called after a read operation only (it has no effect otherwise). It prevents the subsequent read from reading the next element; the element just read before will be returned a second time. Multiple calls to Again without a read in between have the same effect as one call to Again. The position in the file is undefined after a call to Again (it is defined again after the next read operation).

PROCEDURE SetRead (VAR f: File);

- Set the file in reading-state, without changing the current position.

in: f structure referencing an open file

out: f f.res will be set appropriately.

Upon calling SetRead, the current position must be before the eof. In reading-state, no writing is allowed.

PROCEDURE SetWrite (VAR f: File);

-Sets the file in writing-state, without changing the current position.

in: f structure referencing an open file

out: f f.res will be set appropriately.

Upon calling SetWrite, the current position must be a legal position in the file (including eof). In writing-state, no reading is allowed, and a write always takes place at the eof. The current implementation does not truncate the file.

PROCEDURE SetModify (VAR f: File);

- Sets the file in modifying-state, without changing the current position.

in: f structure referencing an open file

out: f f.res will be set appropriately.

Upon calling SetModify, the current position must be before the eof. In modifying-state, reading and writing are allowed. Writing is done at the current position, overwriting whatever element is already there. The file is not truncated.

PROCEDURE SetOpen (VAR f: File);

- Set the file to opened-state, without changing the current position.

in: f structure referencing an open file

out: f f.res will be set appropriately.

The buffer content is written back on the file, if the file has been in writing or modifying status. The new buffer content is undefined. In opened-state, neither reading nor writing is allowed.

PROCEDURE Reset (VAR f: File);

- Set the file to opened state and position it to the top of file.

in: f structure referencing an open file

out: f f.res will be set appropriately.

PROCEDURE SetPos (VAR f: File; high, low: CARDINAL);

- Set the current position in file

in: f structure referencing an open file
high high part of the byte offset
low low part of the byte offset

out: f f.res will be set appropriately.

The file will be positioned (high*2^16 +low) bytes from top of file.

```
PROCEDURE GetPos (VAR f: File; VAR high, low: CARDINAL);

- Return the current byte position in file

in:     f       structure referencing an open file

out:    high    high part of byte offset
        low     low part of byte offset
```

The actual position is (high*2^16 +low) bytes from the top of file.

```
PROCEDURE Length (VAR f: File; VAR high, low: CARDINAL);

  - Return the length of the file in bytes.

in:     f       structure referencing an open file.

out:    high    high part of byte offset
        low.    ;pw part of byte offset
```

The actual length is (high*2^16 +low) bytes.

```
PROCEDURE Doio (VAR f: File);

  -  Do various read/write operations on a file

in:     f       structure referencing an open file

out:    f       f.res will be set appropriately.
```

The exact effect of this command depends on the state of the file (flags):

```
        opened    = NOOP.
        reading   = reads the record that contains
                    the current byte from the file.
                    The old content of the buffer
                    is not written back.
        writing   = the buffer is written back.  It
                    is then assigned to the record,
                    that contains the current
                    position. Its content is not
```

changed.

modifying = the buffer is written back and the record containing the current position is read.

Note that 'Doio' does not need to be used when reading through the 'stream-like' I/O routines. Its use is limited to special applications.

PROCEDURE FileNameChar (c: CHAR): CHAR;

- Check the character c for legality in a MS-DOS filename.

in: c character to check

out: 0C for illegal characters and c otherwise; lowercase letters are transformed into uppercase letters.

TYPE FileProc = PROCEDURE (VAR File);

- Procedure type to be used for internal file operations

A procedure of this type will be called for functions: setread, setwrite, setmodify, setopen, doio, setpos, getpos, length, setprotect, getprotect, setpermanent, getpermanent.

DirectoryProc = PROCEDURE (VAR File, ARRAY OF CHAR);

- Procedure type to be used for operations on an entire file

A procedure of this type will be called for functions: create, close, lookup, rename, delete.

```
PROCEDURE CreateMedium (mt: MediumType;
                        fproc: FileProc;
                        dproc: DirectoryProc;
                        VAR done: BOOLEAN);
```

- Install the medium "mt" in the file system

```
in:   mt      medium type to install
      fproc   procedure to handle internal file operation
      dproc   procedure to handle operations on an
              entire file

out:  done    TRUE if medium was successfully installed.
```

Before accessing or creating a file on a medium, thi medium has to be announced to the file system by means o the routine CreateMedium. FileSystem calls "fproc" an "dproc" to perform operations on a file of this medium. Up to 24 mediums can be announced.

```
PROCEDURE      RemoveMedium      (mt:      MediumType;
                        VAR done: BOOLEAN);
```

- Remove the medium "mt" from the file system

```
in:     mt      medium type to remove

out:    done    true if medium was successfully removed
```

Attempts to access a file on this medium result in a error (unknownmedium).

```
END FileSystem.
```

```
DEFINITION MODULE InOut;

  Standard high-level formatted input/output


Derived from the Lilith Modula-2 system developed by the
group of Prof. N. Wirth at ETH Zurich, Switzerland.


FROM SYSTEM IMPORT WORD;
FROM FileSystem IMPORT File;
EXPORT QUALIFIED
  EOL, Done, in, out, termCH,
  OpenInput, OpenOutput, CloseInput, CloseOutput,
  Read, ReadString, ReadInt, ReadCard, ReadWrd,
  Write, WriteLn, WriteString, WriteInt,
  WriteCard, WriteOct, WriteHex, WriteWrd;

CONST EOL = 36C;

VAR Done:  BOOLEAN;
         Done is set by several procedures, TRUE if the
       * operation was successful, FALSE otherwise.
    termCH:  CHAR;
         terminating character from ReadString, ReadInt,
         ReadCard.
    in, out: File;
         The currently open input and output files,
         respectively.

       * Use for exceptional cases only.
```

```
PROCEDURE OpenInput(defext: ARRAY OF CHAR);

- Accept a file name from the terminal and open it for
input.

in:     defext  default filetype or 'extension'.

If the file name that is read ends with '.', then 'defext'
is appended to the file name. If OpenInput succeeds, Done
= TRUE and subsequent input is taken from the file until
CloseInput is called.
```

```
PROCEDURE OpenOutput(defext: ARRAY OF CHAR);
```

- Accept a file name from the terminal and open it for output.

in: defext default filetype or 'extension'.

If the file name ends in '.', 'defext' is appended. If OpenOutput succeeds, Done = TRUE and subsequent output is written to the file until CloseOutput is called.

```
PROCEDURE CloseInput;
```

- Close current input file and revert to terminal for input.

```
PROCEDURE CloseOutput;
```

- Close current output file and revert to terminal for output.

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

- Read the next character from the current input.

out: ch the character read. (EOL for end-of-line.)

Done = TRUE unless the input is at end of file.

```
PROCEDURE ReadString(VAR s: ARRAY OF CHAR);
```

- Read a string from the current input.

out: s the string that was read, excluding terminator.

Leading blanks are accepted and thrown away, then characters are read into 's' until a blank or control character is entered. ReadString truncates the input string if it is too long for 's'. The terminating character is left in 'termCH'. If input is from the terminal, BS and DEL are allowed for editing.

PROCEDURE ReadInt(VAR x: INTEGER);

- Read an INTEGER representation from the current input.

out: x the value read.

ReadInt is like ReadString, but the string is converted to an INTEGER value if possible, using the syntax: ["+"|"-"] digit { digit }. Done = TRUE if some conversion took place.

PROCEDURE ReadCard(VAR x: CARDINAL);

- Read an unsigned decimal number from the current input.

out: x the value read.

ReadCard is like ReadInt, but the syntax is: digit { digit }.

PROCEDURE ReadWrd(VAR w: WORD);

- Read a WORD value from the current input.

out: w the value read.

Done is TRUE if a WORD was read successfully. (A WORD cannot be read from the terminal.) Note that the meaning of WORD is system-dependent.

```
PROCEDURE Write(ch: CHAR);
```

- Write a character to the current output.

in: ch character to write.

```
PROCEDURE WriteLn;      terminate line
```

- Write a new-line sequence to the current output.

```
PROCEDURE WriteString(s: ARRAY OF CHAR);
```

- Write a string to the current output.

in: s string to write.

```
PROCEDURE WriteInt(x: INTEGER; n: CARDINAL);
```

- Write an integer in right-justified decimal format.

in: x value to be output
n minimum field width.

The decimal representation of 'x' (including '-' if x i
negative) is output, using at least n characters (but mor
if needed). Leading blanks are output if necessary.

```
PROCEDURE WriteCard(x,n: CARDINAL);
```

- Output a CARDINAL in decimal format.

in: x value to be output,
n minimum field width.

The decimal representation of the value 'x' is output, using at least n characters (but more if needed). Leading blanks are output if necessary.

PROCEDURE WriteOct(x,n: CARDINAL);

- Output a CARDINAL in octal format.

in: x value to be output,
n minimum field width. [see WriteCard above]

PROCEDURE WriteHex(x,n: CARDINAL);

- Output a CARDINAL in hexadecimal format.

in: x value to be output,
n minimum field width.

Four uppercase hex digits are written, with leading blanks if n > 4.

PROCEDURE WriteWrd(w: WORD);

- Output a WORD

in: w WORD value to be output.

Note that the meaning of WORD is system-dependent, and that WORDs cannot be written to the terminal.

END InOut.

```
DEFINITION MODULE Keyboard;

  Default driver for terminal input.

Derived from the Lilith Modula-2 system developed by the
group of Prof. N. Wirth at ETH Zurich, Switzerland.
[Private module of the Modula-2 system]

EXPORT QUALIFIED Read, KeyPressed;
```

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

  - Read a character from the keyboard.  out:    ch

If necessary, Read waits for a character to be entered.
Characters that have been entered are returned immedi-
ately, with no editing or buffering.

 - CTRL-C terminates the current program
 - ASCII.cr is transformed into System.EOL
```

```
PROCEDURE KeyPressed (): BOOLEAN;

  - Test if a character is available from the keyboard.

END Keyboard.
```

```
DEFINITION MODULE MathLib0;

  Real Math Functions

From 'Programming in Modula-2' by N. Wirth, 2nd ed.


EXPORT QUALIFIED
  sqrt, exp, ln, sin, cos, arctan, real, entier;
```

```
PROCEDURE sqrt(x: REAL): REAL;

   x must be positive
```

```
PROCEDURE exp(x: REAL): REAL;

   returns e^x where e = 2.71828..
```

```
PROCEDURE ln(x: REAL): REAL;

returns natural logarithm with base e = 2.71828..   of   x:
where x must be positive and not zero
```

```
PROCEDURE sin(x: REAL): REAL;

returns sin(x) where x is given in radians
```

```
PROCEDURE cos(x: REAL): REAL;
```

returns cos(x) where x is given in radians

```
PROCEDURE arctan(x: REAL): REAL;
```

returns arctan(x) in radians

```
PROCEDURE real(x: INTEGER): REAL;
```

type conversion from INTEGER to REAL

```
PROCEDURE entier(x: REAL): INTEGER;
```

returns the integral part of x. If this cannot be represented in an INTEGER, the result is undefined.

```
END MathLib0.
```

```
DEFINITION MODULE NumberConversion;

   Conversion between numbers and strings


 The routines that convert a string to a number:
     - skip leading blanks,
     - accept always a '+' sign and for integers
       also a '-' sign
     - skip blanks between sign and number
 Done is TRUE if the conversion is successful
 The routines that convert a number to a string:
     - if the string is too small the number is truncated
     - if the number has less digits than width,leading
       blanks are added


EXPORT QUALIFIED
  MaxBase, BASE,
  StringToCard, StringToInt, StringToNum,
  CardToString, IntToString, NumToString;

CONST MaxBase = 16;

TYPE BASE = [2..MaxBase];
```

```
PROCEDURE StringToCard(str: ARRAY OF CHAR;
                       VAR num: CARDINAL;
                       VAR done: BOOLEAN);

- Convert a string to a CARDINAL number.

in:     str    string to convert

out:    num    converted number
        done   TRUE if successful conversion,
               FALSE if number out of range,
               or contents of string non numeric.
```

```
PROCEDURE StringToInt(str: ARRAY OF CHAR;
                      VAR num: INTEGER;
                      VAR done: BOOLEAN);
```

```
- Convert a string to an INTEGER number.

in:     str     string to convert

out:    num     converted number
        done    TRUE if successful conversion,
                FALSE if  number out of range,
                or contents of string non numeric.
```

```
PROCEDURE StringToNum(str: ARRAY OF CHAR;
                      base: BASE;
                      VAR num: CARDINAL;
                      VAR done: BOOLEAN);

- Convert a string to a CARDINAL number.

in:     str     string to convert
       base     the base of the number represented in
                the string

out:   num      converted number
       done     TRUE if successful conversion,
                FALSE  or number out of range,
                or contents of string not within base.
```

```
PROCEDURE CardToString(num: CARDINAL;
                       VAR str: ARRAY OF CHAR;
                       width: CARDINAL);

- Convert a CARDINAL number to a string.

in:     num     number to convert

out:    str     returned string representation of the
number

in:   width     width of the returned string
```

```
PROCEDURE IntToString(num: INTEGER;
                      VAR str: ARRAY OF CHAR;
                      width: CARDINAL);
```

- Convert an INTEGER number to a string.

in: num number to convert

out: str returned string representation of the number

in: width width of the returned string

```
PROCEDURE NumToString(num: CARDINAL;
                      base: BASE;
                      VAR str: ARRAY OF CHAR;
                      width: CARDINAL);
```

- Convert a number to the string representation in the specified base.

in: num number to convert
base the base of conversion
width width of the returned string

out: str returned string representation of the number

```
END NumberConversion.
```

```
DEFINITION MODULE Options;

   Read a file specification, with options, from the
terminal


Derived from the Lilith Modula-2 system developed by the
group of Prof. N. Wirth at ETH Zurich, Switzerland.


EXPORT QUALIFIED
  NameParts, NamePartSet, Termination,
  FileNameAndOptions, GetOption;

TYPE Termination = (norm, empty, can, esc);
     NameParts = (NameDrive, NamePath, NameName
                  NameExt);
     NamePartSet = SET OF NameParts;
```

```
PROCEDURE
  FileNameAndOptions(default: ARRAY OF CHAR;
                     VAR name: ARRAY OF CHAR;
                     VAR term: Termination;
                     acceptOption: BOOLEAN);
                     VAR ReadInName: NamePartSet):

  - Read file name and options from terminal.

in:     default          the file name to use if one
                         is not entered,
        acceptOption     if TRUE, allow options to be
                         entered,
out:    name             the filename,
        term             how the read ended.
        ReadInName       which parts of the
                         specification are present

      norm   : normally terminated
      empty  : normally terminated, but name is empty
      can    : <can> is typed, input line cancelled
      esc    : <esc> is typed, no file specified.

Input is terminated by a <cr>, blank, <can>, or <esc>.
<bs> and <del> are allowed while entering the file name.
```

```
PROCEDURE GetOption(VAR optStr: ARRAY OF CHAR;
                    VAR length: CARDINAL);
```

- Get another option from the last call to FileNameAndOptions.

```
out:    optStr          text of the option,
        length          length of optStr.
```

Calls to GetOption return the options from the last call to FileNameAndOptions, in the order they were entered. When there are no more options, a length of 0 is returned.

```
END Options.
```

```
DEFINITION MODULE Processes;

  (pseudo-)concurrent programming with SEND/WAIT

From the book 'Programming in Modula-2' [Wirth]


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


TYPE SIGNAL;

   SIGNAL's are the means of synchronization between
   processes.
```

```
PROCEDURE StartProcess (P: PROC; n: CARDINAL);

  - Start up a new process.

in:     P       top-level procedure that will execute
                in this process.
        n       number of bytes of workspace to be
                allocated to it.

Allocates (from Storage) a workspace of n bytes, and
creates a process executing procedure P in that workspace.
Control is given to the new process.

Caution: The caller must ensure that the workspace size
is sufficient for P. Errors: StartProcess may fail due
to insufficient memory.
```

```
PROCEDURE SEND (VAR s: SIGNAL);

  - Send a signal

in:     s       the signal to be sent. [Must have been
                Init'd]

out:    s       the signal with one less process
```

waiting for it.

If no process is waiting for s, SEND has precisely no effect. Otherwise, some process which is waiting for s is given control and allowed to continue from WAIT.

```
PROCEDURE WAIT (VAR s: SIGNAL);
```

- Wait for some other process to send a signal.

in: s the signal to wait for. [Must have been Init'd]

The current process waits for the signal s. At some later time, a SEND(s) by some other process can cause this process to return from WAIT.

Errors: If all other processes are waiting, WAIT terminates the program.

```
PROCEDURE Awaited (s:SIGNAL): BOOLEAN;
```

- Test whether any process is waiting for a signal.

in: s the signal of interest. [Must have been Init'd]

out: TRUE if and only if at least one process is waiting for s.

```
PROCEDURE Init (VAR s: SIGNAL);
```

- Initialize a SIGNAL object.

in: s the signal to be initialized

out: s the initialized signal (ready to be used as above)

An object of type SIGNAL must be initialized with this procedure before it can be used with any of the other operations. After Init(S), Awaited(S) is FALSE.

END Processes.

```
DEFINITION MODULE ProgMessage;

  Write program status message to the terminal


FROM System IMPORT Status;

EXPORT QUALIFIED WriteStatus;
```

```
PROCEDURE WriteStatus (st: Status);

  - Write a short description of a program status on
    the terminal.

in:     st      a Status, as returned by Program.Call

The message may be up to 32 characters long.


END ProgMessage.
```

```
DEFINITION MODULE Program;
```

Sub-program loading and execution

Derived from the Lilith Modula-2 system developed by the group of Prof. N. Wirth at ETH Zurich, Switzerland.

Under Modula-2/86, programs can be divided into sub-programs (we call them 'programs') which are loaded upon request.

These programs are executed like procedures:

- they have only one entry-point (body of program's main module).
- after termination, their data do not exist any longer. In the case of programs the code also disappears and will be reloaded from disk upon the next activation.
- programs may themselves activate other programs.

```
FROM System IMPORT Status;

EXPORT QUALIFIED  Call, GetErrorInfo;
```

```
PROCEDURE GetErrorInfo (VAR msg: ARRAY OF CHAR);
```

- Obtain more information about a load error.

out: msg a string related to the last error.

After Call (below) has returned a Status value of 'modulenotfound' and 'incompatiblemodules', GetErrorInfo will return the name of the offending module. (length is up to 24 characters). It returns an empty string in all other cases.

```
PROCEDURE Call (programName: ARRAY OF CHAR;
                shared: BOOLEAN;
                VAR st: Status);
```

- Load and execute a (sub) program.

in:	programName	file specification for the program,
	shared	whether to share resources,
out:	st	terminating status of the subprogram.

The file whose name is given in 'programName' is opened loaded, and started. There is no default device or file type: these must be supplied by the caller. The file must contain a linked, relocatable program.

The load address is defined by the default allocation schema, in which programs are loaded on top of stack and a new stack is created for execution of the new program.

If 'shared' = TRUE then all sharable resources allocated by the called program are owned by the calling program (or possibly the caller of the caller...). Shared resources are not released upon termination of the new program.

Upon termination of the program, its memory is freed and the old stack is established. All the resources used by a terminating program are released, if they are not shared and if they have not been released explicitly by the program (files, heap, etc).

Any value of 'st' other than 'normal' indicates an abnormal termination of the subprogram. In some cases GetErrorInfo (above) will provide additional details.

- Cautions -

In case of abnormal termination, Call does NOT print any kind of error message.

Do not assign a procedure in the current program to a procedure variable which could still exist after the current program terminates (for example, a variable in a shared resource or in the calling program). When the current program terminates, all procedures in it must be considered to cease to exist.

The loader in this module is not reentrant. This means that interrupt processes must not load overlays!

```
END Program.
```

```
DEFINITION MODULE RS232Code;
```

High-speed interrupt-driven input/output via the serial port

This module provides interrupt-driven I/O via the serial port, but the Interrupt Service Routine is implemented using in-line code (as opposed to IOTRANSFER). This approach is NOT portable to other Modula-2 implementations, but it allows for treatment of interrupts with high frequency. There is a buffer of at least 128 characters for received data.

Derived from the Lilith Modula-2 system developed by the group of Prof. N. Wirth at ETH Zurich, Switzerland.

```
EXPORT  QUALIFIED  Init,  StartReading,  StopReading,
                BusyRead, Read, Write;
```

```
PROCEDURE Init (baudRate: CARDINAL;
                stopBits: CARDINAL;
                parityBit: BOOLEAN;
                evenParity: BOOLEAN;
                nbrOfBits: CARDINAL;
                VAR result: BOOLEAN);
```

- Initialize the serial port.

in:	baudRate	transmission speed,
	stopBits	number of stop bits (usually 1 or 2),
	parityBit	if TRUE, parity is used, otherwise not,
	evenParity	if parity is used, this indicates even/odd,
	nbrOfBits	number of data bits (usually 7 or 8),
out:	result	TRUE if the initialization was completed.

The legal values for the parameters depend on the implementation (e.g. the range of supported baud rates).

PROCEDURE StartReading;

- Allow characters to be received from the serial port.

This procedure initializes the communication controller to generate interrupts upon reception of a character. It also un-masks the corresponding interrupt level in the interrupt controller.

PROCEDURE StopReading;

- Disable receiving from the serial port.

A call to this procedure disables the communication controller from generating interrupts. In addition it terminates the coroutine which listens to the line. The old interrupt vector as well as the old state of the interrupt controller (mask) is restored.

PROCEDURE BusyRead (VAR ch: CHAR; VAR received: BOOLEAN);

- Read a character from serial port, if one has been received.

out: ch the character received, if any
received TRUE if a character was received.

If no character has been received, ch = 0C, received = FALSE.

PROCEDURE Read (VAR ch: CHAR);

- Read a character from the serial port.

out: ch the character received.

As opposed to BusyRead, Read waits for a character to arrive.

```
PROCEDURE Write (ch: CHAR);
 - Write a character to the serial port.
in:     ch      character to send.
Note: no interpretation of characters is made.

END RS232Code.
```

```
DEFINITION MODULE RS232Int;
```

Interrupt-driven input/output via the serial port

Interrupts are treated with the standard procedure IOTRANSFER. Received characters are stored in a buffer of 100H characters. The module initializes the serial port as follows:

```
baudRate = 1200, stopBits = 1,
parityBit = FALSE, evenParity = don't care,
nbrOfBits = 8
```

Derived from the Lilith Modula-2 system developed by the group of Prof. N. Wirth at ETH Zurich, Switzerland.

```
EXPORT QUALIFIED
  Init, StartReading, StopReading,
  BusyRead, Read, Write;
```

```
PROCEDURE Init (baudRate: CARDINAL;
                stopBits: CARDINAL;
                parityBit: BOOLEAN;
                evenParity: BOOLEAN;
                nbrOfBits: CARDINAL;
                VAR result: BOOLEAN);
```

- Initialize the serial port.

in:	baudRate	transmission speed,
	stopBits	number of stop bits (usually 1 or 2),
	parityBit	if TRUE, parity is used, otherwise not,
	evenParity	if parity is used, this indicates even/odd,
	nbrOfBits	number of data bits (usually 7 or 8),
out:	result	TRUE if the initialization was completed.

The legal values for the parameters depend on the implementation (e.g. the range of supported baud rates).

PROCEDURE StartReading;

- Allow characters to be received from the serial port.

This procedure initializes the communication controller to generate interrupts upon reception of a character. It also un-masks the corresponding interrupt level in the interrupt controller.

PROCEDURE StopReading;

- Disable receiving from the serial port.

A call to this procedure disables the communication controller from generating interrupts. In addition it terminates the coroutine which listens to the line. The old interrupt vector as well as the old state of the interrupt controller (mask) is restored.

```
PROCEDURE BusyRead (VAR ch: CHAR;
                    VAR received: BOOLEAN);
```

- Read a character from serial port, if one has been received. out: ch the character received, if any, received TRUE if a character was received.

If no character has been received, ch = 0C, received = FALSE.

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

- Read a character from the serial port.

out: ch the character received.

As opposed to BusyRead, Read waits for a character to arrive.

```
PROCEDURE Write (ch: CHAR);
```

- Write a character to the serial port.

in: ch character to send.

Note: no interpretation of characters is made.

```
END RS232Int.
```

```
DEFINITION MODULE RS232Polling;

  Polled input/output via the serial port
```

Since this module does not use interrupts, it is the responsibility of the programmer to poll (by calling no characters are lost.

Derived from the Lilith Modula-2 system developed by the group of Prof. N. Wirth at ETH Zurich, Switzerland.

```
EXPORT QUALIFIED Init, BusyRead, Read, Write;
```

```
PROCEDURE Init (baudRate: CARDINAL; stopBits: CARDINAL;
                parityBit: BOOLEAN; evenParity: BOOLEAN;
                nbrOfBits: CARDINAL; VAR result: BOOLEAN);
```

- Initialize the serial port.

in:	baudRate	transmission speed,
	stopBits	number of stop bits (usually 1 or 2),
	parityBit	if TRUE, parity is used, otherwise not,
	evenParity	if parity is used, this indicates even/odd,
	nbrOfBits	number of data bits (usually 7 or 8),
out:	result	TRUE if the initialization was completed.

The legal values for the parameters depend on the implementation (e.g. the range of supported baud rates).

```
PROCEDURE BusyRead (VAR ch: CHAR;
                    VAR received: BOOLEAN);
```

- Read a character from serial port, if one has been received.

```
out:   ch          the character received, if any,
       received    TRUE if a character was received.
```

If no character has been received, ch = 0C, received = FALSE.

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

- Read a character from the serial port.

```
out:   ch      the character received.
```

As opposed to BusyRead, Read waits for a character to arrive.

```
PROCEDURE Write (ch: CHAR);
```

- Write a character to the serial port.

```
in:    ch      character to send.
```

Note: no interpretation of characters is made.

```
END RS232Polling.
```

```
DEFINITION MODULE RealInOut;

  Terminal input/output of REAL values

From 'Programming in Modula-2' by N. Wirth, 2nd edition.

EXPORT QUALIFIED
  ReadReal, WriteReal, WriteRealOct, Done;

 VAR Done: BOOLEAN;
```

```
PROCEDURE ReadReal(VAR x: REAL);

  - Read a REAL from the terminal.

out:     x        the number read.

The syntax accepted is:
        ["+"|"-"] digit {digit} ["." digit {digit}]
        ["E"["+"|"-"] digit [digit]]

If a number is found, Done is set to TRUE (otherwise FALSE).
At most 15 digits are significant, leading zeros not counting.
Maximum exponent is 307. Input terminates with <CR>. Also
accpeted are <ESC> and <^X> which terminate reading and set
'Done' to FALSE. The termination character is swallowed. DEL
may be used for backspacing.
```

```
PROCEDURE WriteReal(x: REAL; n: CARDINAL);

  - Write a REAL to the terminal, right-justified.

in:      x        number to write,
         n        minimum field width.

If fewer than n characters are needed to represent x,
leading blanks are output.
```

```
PROCEDURE WriteRealOct(x: REAL);
```

- Write a REAL to terminal, in octal form with exponent and mantissa.

```
END RealInOut.
```

```
DEFINITION MODULE Storage;

  Standard dynamic storage management

Storage management for dynamic variables.  Calls to the
Modula-2 standard procedures NEW and DISPOSE are
translated into calls to ALLOCATE and DEALLOCATE.  The
standard way to provide these two procedures is to import
them from this module 'Storage'.

Derived from the Lilith Modula-2 system developed by the
group of Prof. N. Wirth at ETH Zurich, Switzerland.


FROM SYSTEM IMPORT ADDRESS;

EXPORT QUALIFIED
    ALLOCATE, DEALLOCATE, Available, InstallHeap,
    RemoveHeap;
```

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

  - Allocate some dynamic storage.

in:     size    number of bytes to allocate,

out:    a       ADDRESS of allocated storage.

The actual number of bytes allocated may be slightly
greater than 'size', due to administrative overhead.  If
not enough space is available, the calling program is ter-
minated with the status 'heapovf'.
```

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

  - Release some dynamic storage.

in:     a       ADDRESS of the area to release,
        size    number of bytes to be released,

out:    a       set to NIL.
```

```
PROCEDURE Available (size: CARDINAL) : BOOLEAN;
- Test whether some number of bytes could be allocated.
in:     size    number of bytes
out:    TRUE if ALLOCATE(p,size) would succeed.
```

```
PROCEDURE InstallHeap;
  - Used by the loader -
```

```
PROCEDURE RemoveHeap;
  - Used by the loader -

END Storage.
```

```
DEFINITION MODULE Strings;

  Variable-length character strings handler.

    :

NOTE:  For most of these string handling  procedures,there
is  the  possibility  of the user not providing a variable
large enough to contain the result of a string  operation.
Should  this  possibility  arise  truncation may result,as
there will be no other error notification.  The  implemen-
tation  of  this  module  must not cause a range error, it
should instead silently truncate.

String variables have the following characteristics:
     They are ARRAY OF CHAR
     Lowest bound must be 0
     The size of the string is the size of the string
     variable unless the null character (0C) occurs in
     the string to indicate end of string.


EXPORT QUALIFIED
  Assign, Insert, Delete, Pos,
  Copy, Concat, Length, CompareStr;
```

```
PROCEDURE Assign (VAR source, dest: ARRAY OF CHAR);

- Assign the  contents  of  string  variable  source  into
string variable destination

in:     source

out:    dest
```

```
PROCEDURE Insert (substr: ARRAY OF CHAR;
                  VAR str: ARRAY OF CHAR;
                  inx: CARDINAL);

- Insert the string substr into str,starting at str[inx].

in:     substr
        str
        inx
```

out: str

If inx is equal or greater than Length(str) then substr is appended to end of dest.

```
PROCEDURE Delete (VAR str: ARRAY OF CHAR;
                  inx: CARDINAL;
                  len: CARDINAL);
```

- Delete len characters from str, starting at str[inx].

in: str
inx
len

out: str

If inx >= Length(str) then nothing happens. If there are not len characters to delete, characters to the end of string are deleted.

```
PROCEDURE Pos (substr, str: ARRAY OF CHAR): CARDINAL;
```

- Return the index into str of the first occurrence of the substr.

in: substr
str

Pos returns a value greater then HIGH(str) if no occurrence of the substring is found

```
PROCEDURE  Copy (str: ARRAY OF CHAR;
                 inx: CARDINAL;
                 len: CARDINAL;
                 VAR result: ARRAY OF CHAR);
```

- Copy at most len characters from str into result.

in: str source string,

```
        inx     starting position in 'str',
        len     maximum number of characters to copy,

out:    result  copied string
```

```
PROCEDURE Concat (s1, s2: ARRAY OF CHAR;
VAR result: ARRAY OF CHAR);

  - Concatenate two strings.

in:     s1          left string,
        s2          right string,

out:    result      receives left string followed by
                    right string.
```

```
PROCEDURE Length (VAR str: ARRAY OF CHAR): CARDINAL;

  - Return the number of characters in a string.

in:     str
```

```
PROCEDURE CompareStr (s1, s2: ARRAY OF CHAR): INTEGER;

  - Compare two strings.

in:     s1
        s2

Returns an integer value indicating the comparison result:

    -1 if s1 is less than s2;
     0 if s1 equals s2;
     1 if s1 is greater than s2


END Strings.
```

```
DEFINITION MODULE System;
```

Additional system-dependent facilities

This module may be seen as an extension of the standard pseudo-module SYSTEM.

Derived from the Lilith Modula-2 system developed by the group of Prof. N. Wirth at ETH Zurich, Switzerland.

```
FROM SYSTEM IMPORT ADDRESS, PROCESS;

EXPORT QUALIFIED
    EOL,
    Status, Terminate,
    ProcessDescriptor, ProcessPtr, curProcess,
    targetSystem,
    SetTime, GetTime, Time,
    TermProcedure, CallTermProc,
    InitProcedure, CallInitProc,
    RTSCall,
    RegAX,    RegBX,
    RegCX,    RegDX,
    RegSI,    RegDI,
    RegES,    RegDS,
    RegCS,    RegSS,
    RegBP,    RegSP;

CONST
    EOL = 36C;
```

This constant defines the internal name of the End-Of-Line character. Using this constant has the advantage, that only one character is used to specify line ends (as opposed to CR/LF). The standard I/O modules interpret this character and transform it into the End-Of-Line (sequence of) code(s) required by the device they support. See definition modules of 'Terminal' and 'FileSystem'.

```
TYPE
    Status = (normal, warned,                    no dump pro-
duced for these two cases

                stopped, asserted, halted, caseerr,
                stackovf, heapovf, functionerr,
                addressoverflow, realoverflow,
                cardinaloverflow, integeroverflow,
                rangeerr, dividebyzero, coroutineend,
```

```
            loaderr, callerr, programnotfound,
            modulenotfound, incompatiblemodule,
            filestructureerr, illegalinstr,
            RTSfunctionerr, interrupterr);
```

This type defines the possible values for a program's status. The meaning of these values can be printed to the terminal by means of ProgMessage.WriteStatus .

```
PROCEDURE Terminate (st: Status);
```

- Terminate the current (sub) program.

in: st terminating status.

If the value of 'st' is different from 'normal' or MEMORY.PMD, which can be used for subsequent debugging. The value of 'st' will be returned to the caller of the terminating program by means of the parameter 'st' of the procedure 'Program.Call'.

This procedure never returns to the caller.

```
TYPE
    ProcessDescriptor = RECORD
      AX, BX, CX, DX, SP, BP, SI, DI : CARDINAL;
      DS, SS, ES, CS, IP : CARDINAL;
      flags : BITSET;
      status : Status;
      programId, auxId, sharedId : CARDINAL;
      fatherProcess : PROCESS;
      stackLimit : CARDINAL;
      interruptMask : BITSET;
      retStack: CARDINAL;
      progEndStack : ADDRESS;
      intVector : CARDINAL;
      oldISR, interruptedProcess : ADDRESS;
      heapBase, heapTop : ADDRESS;
      modTable : ADDRESS;
    END;


TYPE
    ProcessPtr = POINTER TO ProcessDescriptor;


VAR
    curProcess: ProcessPtr;
```

Points at any moment to the current process's workspace. This variable is 'read-only' and must not be used in application programs.

WARNING:

improper use of this variable may cause unpredictable behaviour of the system.

```
CONST
    targetSystem = 0;     first implementation
```

May be used to check compatibility of file or programs with the present system.

```
TYPE Time = RECORD day, minute, millisec: CARDINAL; END;

   'day' is : Bits 0..4 = day of month (1..31),
              Bits 5..8 = month of the year (1..12),
              Bits 9..15 = year - 1900.
   'minute' is hour * 60 + minutes.
   'millisec' is second * 1000 + milliseconds,
              starting with 0 at every minute.
```

```
PROCEDURE GetTime (VAR curTime: Time);
```

- Return the current date and time.

out: curTime record containing date and time.

On systems which do not keep date or time, 'GetTime' returns a pseudo-random number.

```
PROCEDURE SetTime (curTime: Time);
```

- Set the current date and time.

in: curTime record containing date and time.

On systems which do not keep date or time, this call has no effect.

```
PROCEDURE TermProcedure (p: PROC);
```

- Declare a termination routine.

in: p termination procedure.

The procedure 'p' will be called upon termination of the current program or subprogram. Typical use is for drivers, which have to release resources used by the terminating program. Up to 20 termination routines can be installed.

```
PROCEDURE CallTermProc;
```

- Call all termination procedures for the current program.

Calls all procedures declared with 'TermProcedure' in the current program. 'CallTermProc' is automatically called at the termination of a program or subprogram.

```
PROCEDURE InitProcedure (p: PROC);
```

- Declare an initialization routine.

in: p initialization procedure.

Analoguous to 'TermProcedure', but for routines that have to be called before execution of a program. Up to 20 initialization routines can be installed.

```
PROCEDURE CallInitProc;
```

- Call all initialization procedures for the current program.

Analoguous to 'CallTermProc'.

```
CONST
    RTSCall = 228;
        Interrupt vector for general entry of RTS (for
        Run-Time Support). The RTS is a resident assembly
        program, providing the basic support for running
        Modula-2 programs.
```

```
CONST
        Define the processor's registers, which may be
        used as parameters for the standard procedures
        'SETREG' and 'GETREG'.

    RegAX = 0;     RegCX = 1;
    RegDX = 2;     RegBX = 3;
    RegSP = 4;     RegBP = 5;
    RegSI = 6;     RegDI = 7;
    RegES = 8;     RegCS = 9;
    RegSS = 10;    RegDS = 11;

END System.
```

```
DEFINITION MODULE Termbase;
```

Terminal input/output with redirection hooks

Derived from the Lilith Modula-2 system developed by the group of Prof. N. Wirth at ETH Zurich, Switzerland. [Private module of the Modula-2 system]

```
EXPORT QUALIFIED
  ReadProcedure, StatusProcedure, WriteProcedure
  AssignRead, AssignWrite, UnAssignRead,
  UnAssignWrite, Read, KeyPressed, Write;

TYPE ReadProcedure = PROCEDURE (VAR CHAR);
```

To assign a private read procedure (for redirection of input) a procedure of type 'ReadProcedure' must be provided. This procedure returns a character from the input device. It waits until a character hes been entered.

```
TYPE StatusProcedure = PROCEDURE (): BOOLEAN;
```

To assign a private status-procedure (for redirection of input) a procedure of type 'StatusProcedure' must be provided. This procedure returns TRUE, if a character is available to read, FALSE otherwise.

```
TYPE WriteProcedure = PROCEDURE (CHAR);
```

To assign a private write procedure (for redirection of output) a procedure of type 'WriteProcedure' must be provided. This is typically used to redirect output to a file or to the screen and a file (log file). Special interpretation of characters sent to the screen can be performed in such a private driver procedure.

```
PROCEDURE AssignRead (rp: ReadProcedure;
                      sp: StatusProcedure;
                      VAR done: BOOLEAN);
```

- Install read and status routines for terminal input.

in: rp read-a-character procedure
sp is-character-available function

out: done TRUE if the installation was done.

Initially the corresponding procedures of 'Keyboard' are installed.

Subsequent assignments of read and status procedures are handled in a stack oriented way. Up to six levels of assignments are supported. Done = FALSE if this depth is exceeded.

Upon termination of a (sub-)program the read and status procedures installed by that program are always removed, i.e. the stack is automatically set back to its level upon start of the (sub-)program. This also holds for 'shared' program calls (see module 'Program'). In this respect, read procedures are non-sharable resources. However, a (sub-)program that does not install a read procedure of its own, will by default use the read procedure most recently assigned by its 'father' program.

```
PROCEDURE AssignWrite (wp: WriteProcedure;
                       VAR done: BOOLEAN);
```

- Install write routine for terminal output.

in: wp character output procedure,
out: done set TRUE if the installation was done.

[See AssignRead above.] Initially the procedure Display.Write is assigned.

```
PROCEDURE UnAssignRead (VAR done: BOOLEAN);
```

- Undo the last AssignRead by the current program.

out: done set TRUE if there was something to unassign.

The previously valid procedures become active again.

```
PROCEDURE UnAssignWrite (VAR done: BOOLEAN);
```

- Undo the last AssignWrite by the current program.

```
out:    done    set TRUE if there was something to
                unassign.
```

The previously valid procedure becomes active again.

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

- Read a character using the current input procedure.

```
out:    ch      the character read, or NUL.
```

If no character is available, NUL (0C) is returned. Uses the current status-procedure and read-procedure.

```
PROCEDURE KeyPressed (): BOOLEAN;
```

- Test if a character is available from the current input.

Uses the current status-procedure, as assigned by AssignRead.

```
PROCEDURE Write (ch: CHAR);
```

- Write a character to the current output.

```
in:     ch      character to write.
```

Uses the current write-procedure as assigned by AssignWrite.

```
END Termbase.
```

```
DEFINITION MODULE Terminal;
```

Terminal Input/Output

Derived from the Lilith Modula-2 system developed by the group of Prof. N. Wirth at ETH Zurich, Switzerland.

```
EXPORT QUALIFIED
  Read, KeyPressed, ReadAgain, ReadString,
  Write, WriteString, WriteLn;
```

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

- Read a character from the terminal.

out: ch character that was read.

The character is not echoed. Code ASCII.cr from keyboard is transformed into System.EOL.

```
PROCEDURE KeyPressed (): BOOLEAN;
```

- Test if a character is available to Read from terminal.

```
PROCEDURE ReadAgain;
```

- Undo the last read: Make the last character be re-read.

```
PROCEDURE ReadString(VAR string: ARRAY OF CHAR);
```

- Read a line from the terminal.

out: string receives the text of the line

Characters are accepted (and echoed) from the keyboard until <cr> is entered. The <cr> is not returned or

echoed. <del> and <bs> can be used for editing. Tabs may be entered, but are expanded into blanks immediately. No other control characters may be entered.

```
PROCEDURE Write (ch: CHAR);
```

- Write a character to the terminal.

in: ch character to be written.

If terminal output has not been redirected, the following interpretations are made:

```
  System.EOL (36C) = go to beginning of next line
  ASCII.ff   (14C) = clear screen and set cursor home
  ASCII.del (177C) = erase the last character on the left
  ASCII.bs   (10C) = move 1 character to the left
  ASCII.cr   (15C) = go to beginning of current line
  ASCII.lf   (12C) = move 1 line down, same column
```

```
PROCEDURE WriteString (string: ARRAY OF CHAR);
```

- Write a string to the terminal.

in: string string to be written.

The string can be terminated by a NUL (0C).

```
PROCEDURE WriteLn;
```

- Write a new-line to the terminal. [Equivalent to Write(EOL)]

```
END Terminal.
```

INDICES

INDEX OF LIBRARY MODULES

INDEX OF PROCEDURES OF LIBRARY MODULES

in alphabetical order by procedure name

GENERAL INDEX

THE SYMBOLIC RUN-TIME DEBUGGER

Table of Contents

THE SYMBOLIC RUN-TIME DEBUGGER

1 Introduction

This chapter describes the symbolic run-time debugger (rtd) distributed as a separate package for LOGITECH Modula-2/86. The symbolic run-time debugger is a complement to, and cannot be used without, the LOGITECH Modula-2/86 Base Language System.

The symbolic run-time debugger allows the user to monitor the execution of a program. The user executes the program in steps, simulating slow motion. After each step, the user may inspect the data and the current status of the program. He can modify the values of the variables the program uses. There are several ways the user can step through the program. Depending on the situation, he may decide to execute in larger or smaller steps.

The structure and user interface of the run-time debugger are very similar to that of the post-mortem debugger. The run-time debugger uses the same windows and screen layout as the post-mortem debugger. The run-time debugger commands are a superset of the post-mortem debugger commands:

- All global commands of the post-mortem debugger are also valid in the run-time debugger.

- In any particular window, all local commands in the post-mortem debugger are also valid in the run-time debugger.

This chapter describes those features and functions which are specific to the run-time debugger. The post-mortem debugger is documented in another chapter. Please refer to this chapter for a description of the commands that are common to the post-mortem debugger and the run-time debugger.

2 How to Run the Run-Time Debugger

To initialize the run-time debugger, enter:

```
A> m2 rtd<CR>
```

The debugger responds with a sign-on message:

Modula-2/86 Run-Time Debugger

followed by the version number and a copyright notice. Then, the debugger asks for the name of the program the user wishes to debug. Enter the file name followed by <CR>. The debugger will then load your program into memory, and display the Module window. At this point, the program has not started to execute.

The user may set breakpoints before executing the program. The user instructs the debugger to start the execution of the program by entering one of the Go commands.

When the program is terminated, the debugger prompts the user to enter the name of the next program to debug. Enter <ESC> to exit.

2.1 Run-Time Debugger Options

When the user starts the run-time debugger, he may also specify, on the command line, various options. Options are denoted by a slash (/) followed by the first character of the option name. For example, to activate the query and swap options, the user enters:

A> m2 rtd/q/s<CR>

when starting the run-time debugger.

The following options are available:

Option	Action
/Query	The Query option indicates that reference and source files should be searched for according to the query strategy (see corresponding section of the Modula 2/86 manual for description). The user will be prompted to enter the reference and source file names. If the Query option is not specified, the debugger automatically searches for these files according to the default search strategy.

Option	Action
/Large	The Large option enlarges the internal workspace of the run-time debugger. This workspace is used for storing information on the program being debugged. In particular, it contains information for each module of the program. When debugging large programs consisting of many modules, the default workspace of the run-time debugger may be too small. This would lead to a stack or heap overflow in the debugger itself. The size of the default workspace is 16 K bytes. When the Large option is used, this size is increased to 32 K bytes.
/Swap	The Swap option enlarges the memory available to the program being debugged. This enlargement is made by swapping a part of the run-time debugger code with the program being debugged. A more complete description of the Swap option is given in the next section.
/Version	The Version option displays the date and version of the run-time debugger.

2.2 Memory Requirements and Swapping

The run-time debugger requires approximately 210 K bytes of memory to run. The remaining memory can be used by the program being debugged. For example, on a system with 256 K bytes of memory, the user can debug a program that uses approximately 55 K bytes.

The requirement of 210 K bytes includes approximately 34 K bytes for the operating system (DOS 2.0), 8 K bytes for the special verson of the MOdula-2/86 run-time support (file M2S.EXE), and 168 K bytes for the run-time debugger itself. If the user's operating system is larger than 34 K bytes, he should compute the requirements accordingly.

A special version of the Modula-2/86 run-time support is provided with the run-time debugger in the file M2S.EXE. The regular run-time support (M2.EXE) preserves 17 K bytes for the DOS command interpreter on top of the memory. This special version makes this memory available to Modula-2 programs. If the run-time debugger is run with the regular version of the run-time support, it will use approximately 227 K bytes. The only disadvantage of the special run-time support is that DOS will need to load from disk its command interpreter each time the debugger terminates.

With the Swap option, it is possible to enlarge the memory space available to the program being debugged by approximately 40 K bytes. On a system with 256 K bytes of memory, this allows the user to debug programs that use up to 95 K bytes.

When the user specifies the Swap option, part of the run-time debugger is kept in memory until needed. This part includes the handling of the Call window, the Module window, the Data window and the Raw window. When the program has been stopped and the user invokes one of these windows, the program is swapped out to disk. It will be swapped into memory as soon as the user resumes execution.

Note that the handling of the Text window belongs to the resident part of the run-time debugger. As long as the user activates this window only, no swapping will occur. This allows the user to step through a program avoiding the delay caused by swapping.

When the user chooses the Swap option, the debugger creates the two Swap files RTDSWAP.RTD and RTDPROG.RTD in the current directory of the current drive. Both files have a fixed size of approximately 45 K bytes. Therefore, when using the Swap option the user should make sure 90 K bytes of disk space are available.

2.3 Programs Taking Command Line Arguments

With the run-time debugger, the user can debug programs that take arguments on the command line. When the debugger asks for the program to be debugged, the user should enter the arguments in the usual way. For example:

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

A> m2 mycopy file1 file2<CR>

With the run-time debugger, following will start the program in the same way:

name of the program (MAIN.LOD)> mycopy file1 file2<CR>

3 Control of Program Execution

There are two ways the user can control the program being debugged. One is to set breakpoints on some specific statements of the program. The other is to step through the program, stopping at each statement or procedure call.

When the debugger stops the execution of the program, either at a breakpoint or after a step has been executed, the user can inspect and modify the content of variables in any part of the program. The user may examine any process, and he may view or change the data of any module or any active procedure.

3.1 Breakpoints

One way for the user to monitor program execution is to indicate to the run-time debugger certain points at which the execution of the program should stop. These points are called breakpoints. When the program executes a statement on which a breakpoint is set, the program stops and the user may examine the data structures and the status of the program.

The user may set a breakpoint on any statement of the program. The debugger sets no limit to the number of breakpoints. The user may set or remove breakpoints before he starts the execution of the program or any time the program is stopped.

Each breakpoint has an occurrence counter associated with it. Each time the user sets a breakpoint, the debugger prompts him to specify a limit for the occurrence counter. This counter tells the debugger how many times to execute the statement before stopping the program. Once an occurrence counter has reached its limit, the debugger stops the program each time it encounters this breakpont.

For example, the user sets the limit of the counter for a particular breakpoint to five. The run-time debugger will execute the program until the fifth time it reaches the statement on which this breakpoint is set. If the user continues the execution of the program, the debugger will stop the program each time this breakpoint is encountered.

3.2 Step Mode

The user can also instruct the debugger to execute the program statement by statement or procedure call by procedure call. The debugger 'steps' through the program stopping its execution at the beginning of the next statement or procedure call. Another possible step is to execute the program up to the return from the current procedure. If a breakpoint is encountered during the execution of a step, the program will stop at the breakpoint. Anytime the program is stopped, the user may examine its current status and data.

3.3 Overview of Run-time Debugger Commands

There are five global commands which most clearly distinguish the run-time debugger from the post-mortem debugger. These commands allow the user to control the execution of the program by stopping at specific points in the program. Whenever the program is stopped, the user can examine its current status, and display and modify its data. In this way the user can determine more specifically the location and cause of problems in his program.

The five global commands are described in detail in the corresponding section. The following list briefly defines each command. The user invokes these global commands by entering the letters of the command name, shown in upper case on the command line. For example, the user activates the Go Breakpoint command by typing 'GB'.

- Go Breakpoint
  Stop at the next breakpoint
- Go<CR>
  Same meaning as 'Go Breakpoint'
- Go Statement
  Stop on the next statement
- Go Procedure
  Stop on the next procedure call
- Go Return
  Stop on the return from the current procedure
- Go End
  Execute the program until the end, ignoring breakpoints

3.4 Run-Time Errors

When a run-time error occurs in the program being debugged or when the program calls the standard procedure HALT, the run-time debugger gains control and displays the Call window. No memory dump (file MEMORY.PMD) is generated. The run-time debugger also indicates in the Call window the cause of the run-time error. The user can now inspect the program, but he cannot resume the execution. When the user activates a Go command, the debugger displays the following message:

```
Note: Program stopped due to error or HALT
```

Then, the debugger asks for a new program to debug, as when the program terminates normally.

3.5 Stopping Programs During Execution

A program being debugged with the run-time debugger should not be linked such that its object file contains module 'Break'. Module 'Break' is already linked into the run-time debugger itself.

see errata

The program being debugged can be stopped by typing <Ctrl-break> when it runs into an infinite loop or at any other time it is executing. The run-time debugger will then display the Call window, in the same way as when a run-time error occurs. The run-time debugger handles the execution of a program stopped with <Ctrl-break> in the same way as programs that stop because of a run-time error.

3.6 Debugging Programs That Use Overlays

Each time an overlay is called, the run-time debugger stops the execution when the overlay has been loaded, but before it has started execution. This is similar to what happens when the user starts debugging a program. The debugger displays the Module window when the overlay has been loaded. The user may then set breakpoints or start the execution of the overlay in step mode.

4 Window Format

The run-time debugger has the same windows as the post-mortem debugger - the Call window, the Module window, the Data window, the Text window and the Raw window. As in the post-mortem debugger, the first two lines of each window indicate the commands available.

RUN-TIME DEBUGGER USER'S MANUAL

Errata

page 7 (replace sub-section 3.5 by the follwoing)

A program being debugged, with or without the run-time debugger, should import module 'Break', so that its object file will include this module.

The program being debugged can be stopped by typing <Ctrl-C> when it is waiting for input, or <Ctrl-break> at any other time it is executing, for instance when it runs in an infinite loop. If a program that contains module 'Break' is stopped in this way, then the run-time debugger handles this situation in the same way as when a run-time error occurs. It displays the Call window, and you can inspect the status and the data of the program as they were when <Ctrl-C> or <Ctrl-break> was typed. It is not possible to resume the execution of the program. Upon the next Go command, the run-time debugger will display a message. It then prompts you to enter the name of the next program to debug, as when the program terminates normally.

If a program that does not contain module 'Break' is stopped by <Ctrl-C> or <Ctrl-break>, then the run-time debugger will not display the Call window. Instead, it will just terminate the program, and prompt for the next program to debug.

4.1 Markers

As in the post-mortem debugger, the greater-than (>) sign is used in the run-time debugger as an execution marker to indicate active code. It appears in the Call, the Module and the Text windows and its meaning is the same in the run-time debugger as in the post-mortem debugger.

In the Call, the Module and the Text windows certain lines are marked with an asterisk (*) to indicate where the user has set breakpoints throughout the program. A breakpoint can be set at any statement in any procedure or module.

The breakpoint at which a program stops is marked with a pound sign (#) which replaces the asterisk.

4.2 Selecting an Item for Display

Like the post-mortem debugger the run-time debugger displays the position of the selected item in the lower part of each window. The user may select a different item using the cursor keys or by entering a new position.

5 The Run-Time Debugger Commands

Like the post-mortem debugger, the run-time debugger has two types of commands - global and local. The same definitions apply to these commands in the run-time debugger as in the post-mortem debugger. Local commands are only applicable to the particular window in which they appear and are explained in the appropriate sections.

5.1 Global Commands

In addition to the global commands available in the post-mortem debugger, six new global commands are available in the run-time debugger. The global commands appear on the second line of each window, below the window name and local commands:

```
=[Call Mod Data Text Raw Init] Hexa Quit
  #[P L N] Go[End Bpt Ret Proc Stat]
```

The following describes the global commands available in the run-time debugger only:

Command	Action
GoEnd	Instructs the debugger to execute the program until the end, ignoring all breakpoints.
GoBreakpoint	Instructs the debugger to execute the program until the next breakpoint.

For the following commands, the debugger stops the program at the next breakpoint it encounters, or after the specified step has been completed, whichever comes first:

GoReturn	Instructs the debugger to execute the program until the return from the current procedure, or to the next breakpoint.
GoProcedure	Instructs the debugger to execute the program until the next procedure call, or to the next breakpoint.
GoStatement	Instructs the debugger to execute the program until the next statement, or to the next breakpoint.

5.2 Activating the Step Mode

When the user invokes the Go Statement or the Go Procedure command, the step mode is active only in certain modules. The debugger executes the program and stops at each statement or procedure in those modules in which the user has enabled the step mode. Unless a breakpoint is encountered, the program will not stop in a module where the step mode is not enabled. When a program is loaded by the debugger, by default the step mode is disabled in all modules that belong to the system library. For all other modules, the step mode is enabled.

In the Module window, the run-time debugger marks modules where the step mode is enabled with a plus sign (+) preceeding the module name. It does not mark modules with step mode disabled. The user may change the default and enable or disable the step mode in any module when the Module window is displayed.

5.3 Display of Information

When the debugger stops executing the program at a breakpoint or after a step has been performed, it displays the same window which was shown when the user initiated the execution of the program. For example, if the user invokes the 'Go Procedure' command from the Text window, the debugger will again display the Text window when it stops executing the program.

5.4 Use of the Step Mode in a Multi-Process Program

If the program to be debugged contains more than one process, the step mode is only applicable to one process at a time. The commands Go Statement, Go Procedure and Go Return always refer to the current process only. When the user invokes one of these commands, the debugger will stop the program in the current process - the same process in which it was stopped the last time.

If the user wishes to stop the program in another process, he must set a breakpoint on a statement in a procedure that will be executed by this other process. When the debugger encounters this breakpoint, the user selects the appropriate step mode to examine this new process. The step mode is then only applicable to the new process. Whenever the debugger stops the execution of the program the user can set appropriate breakpoints to stop the program in the original process, or in any other process.

6 Run-time Debugger Windows

The following sections describe those aspects of the run-time debugger windows which differ from the post-mortem debugger windows. They explain the local commands which are available in the run-time debugger only.

6.1 Call Window

The Call window in the run-time debugger has the same major components and functions as in the post-mortem debugger. It displays the chain of procedure calls of a process. In the run-time debugger the Call window cannot be invoked before the user has started the program with the Go command. Because no procedure of the program is active at that time, the Call window would be empty. When this error occurs, the debugger displays the following message:

Error: Cannot display Call window during loading

to indicate that the program has been loaded into memory but has not been started yet. Because no procedure of the program is active at that time, the Call window is empty.

There are no local commands available in the run-time debugger Call window.

The following example shows the Call window. The message 'Status: procedure step' indicates that the program stopped after it completed a Go Procedure command. The two procedures marked with an asterisk have breakpoints in them.

```
CALL |
=[Call Mod Data Text Raw Init] Hexa Quit #[P L N] Go[End Bpt Ret Proc Stat]
--------------------------------------------------------------------------------
Status: procedure step
 procedure step for this process

  1 *> RecursiveOne      in Demo      stops at line   36 , statement  1
  2 *> RecursiveOne      in Demo           at line   38 , statement  1
  3  > FirstOne          in Demo           at line   24 , statement  1
  4  > initialization    of Demo           at line   57 , statement  1
  5  > PROCESS

   Position > 1
    Note: Execution until next procedure or next breakpoint
```

SAMPLE SCREEN 1

6.2 Module Window

The Module window displays the list of modules that constitute the program being debugged. The modules in which the step mode is enabled are marked with a plus sign (+).

Local Commands in the Module Window

There are two local commands in the Module window of the run-time debugger. The user invokes them by entering the first character (shown in upper case) of each command name. The local commands appear on the first line of the window, to the right of the window name as follows:

MODULE Enablestep Disablestep

Command	Action
Enablestep	Enables the step mode in the selected module. When the user invokes the Go Procedure and Go Statement commands to step through the program, the program will only stop in the modules where the step mode is enabled.
Disablestep	Disables the step mode in the selected module. For all modules of the system library, the step mode is disabled by default.

6.3 Data Window

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

Local Commands in the Data Window

There is one additional local command in the Data window of the run-time debugger. The Modify command appears at the end of the first line of the window after the window name and the other local commands.

Command	Action
Modify	Modifies the contents of the selected variable or parameter. The debugger prompts the user to enter the new value according to the type of the data item: o CARDINAL, INTEGER, REAL The user enters the new value which must be of the same type. o BYTE, WORD The user enters the new value as a CARDINAL number. o ADDRESS, POINTER The user enters the new value in the form :<offset>. Both parts are four digit, hexadecimal numbers.

Command Action

o BOOLEAN
The user changes items of type BOOLEAN by entering a T for TRUE or an F for FALSE.

o CHAR
The user modifies items of type CHAR by entering a character in quotes, such as 'a' or "a", or by entering an octal value.

o BITSET
The user modifies items of type BITSET by entering a binary number. The binary number consists of up to 16 digits of 'one' or 'zero', indicating that the corresponding bit should or should not be set. If the user does not wish to modify a certain bit, he can enter an X at this position and the debugger will retain the original value for this bit.

o SET
The user modifies items of type SET by invoking the Son command to list the contents of the set. The run-time debugger then lists the possible elements in the set and indicates whether each element is in the set or not. To change the elements included in the set, the user must select a particular element and activate the Modify command. By responding with T for TRUE or an F for FALSE to the prompt 'In set?' he can then include or exclude that element into or from the set.

o Enumeration
The user modifies the value by entering the name of the element to which he wants to set the value. The element name must be given as defined by the declaration of the enumeration type.

The following sample screens show the path the user follows to modify the content of an array element with a record structure. First, he invokes the Son command to view the elements of the variable 'node' of the module 'Demo'. (Sample Screens 2 & 3) Next, he again invokes the Son command to display the fields of the record 'node[1]', and the value and type of each field. (Sample Screen 4) Finally, the user modifies the value of the first field which is of type CARDINAL. He invokes the Modify command and enters a 6 to change the value from 1 to 6. Sample Screen 5 shows the modified data.

```
DATA | Son Father Left(dec index) Right(inc) Var X Addr Examine(process) Modify
=[Call Mod Data Text Raw Init] Hexa Quit #[P L N] Go[End Bpt Ret Proc Stat]
-------------------------------------------------------------------------------
Demo.

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

   Position > 4
```

SAMPLE SCREEN 2

```
DATA | Son Father Left(dec index) Right(inc) Var X Addr Examine(process) Modify
=[Call Mod Data Text Raw Init] Hexa Quit #[P L N] Go[End Bpt Ret Proc Stat]
-------------------------------------------------------------------------------
Demo.node

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

   Position > 1
```

SAMPLE SCREEN 3

```
DATA | Son Father Left(dec index) Right(inc) Var X Addr Examine(process) Modify
=[Call Mod Data Text Raw Init] Hexa Quit #[P L N] Go[End Bpt Ret Proc Stat]
-------------------------------------------------------------------------------
Demo.node[1]

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

   Position > 1    new value (cardinal) > 6
```

SAMPLE SCREEN 4

```
DATA | Son Father Left(dec index) Right(inc) Var X Addr Examine(process) Modify
=[Call Mod Data Text Raw Init] Hexa Quit #[P L N] Go[End Bpt Ret Proc Stat]
-------------------------------------------------------------------------------
Demo.node[1]

  1  data1                              6  CARDINAL
  2  data2          2.0000000000E+000  REAL
  3  data3                              3  INTEGER

   Position > 1
```

SAMPLE SCREEN 5

6.4 Text Window

The Text window displays the text of the module or procedure in which the debugger stops the program. The greater-than sign (>) indicates the line in which the debugger stopped the program, the call of the next procedure, or where the last process transfer or interrupt occurred.

Local Commands in the Text Window

The three local commands specific to the Text window of the run-time debugger allow the user to set and delete breakpoints.

Command	Action
Setbreakpoint	Sets a breakpoint in the selected line. If more than one statement is on the line, the run-time debugger prompts the user to indicate on which statement he wishes to set the breakpoint. The run-time debugger also prompts the user to set a limit for the occurrence counter associated with the breakpoint. The user may type <CR> for the default value for this limit which is 1. If a breakpoint is already set on the selected statement the debugger replaces the old value of the occurrence counter with the new one.
Clearbreakpoint	Removes a breakpoint on the selected line. If more than one statement is on the line, the debugger prompts the user to indicate from which statement he wishes the breakpoint to be removed.
Killallbreakpoint	Removes all breakpoints from the program.

6.5 Raw Window

The Raw window displays the memory contents around a given address. The initial address of the selected memory location depends on the window from which the user invokes the Raw window. The values are set the same way as in the post-mortem debugger.

Local Commands in the Raw Window

There are three additional local commands in the Raw window of the run-time debugger.

Command	Action
Input/ Output	Used to read in and write out data through an I/O port. The debugger prompts whether to read in or write out a byte or a word. It then asks the user to enter the address of the serial port which will be used.
Modify	Allows the user to modify the memory contents at the selected address. The debugger asks the user for the new value and specifies in which format it should be entered. The format to be used depends on the format in which the Raw window currently displays the memory contents.

Appendix 1

ERROR MESSAGES IN RUN-TIME DEBUGGER

The following is an alphabetical list of the run-time debugger error messages. When error messages are caused by certain commands only, these commands are listed in brackets. For error messages not listed in this chapter, please refer to the list of error messages of the post-mortem debugger.

- **Can not display call window during loading**
  [Call command]

  The Call window shows the chain of active procedures. If the program has not started execution, no procedures are active; thus, the Call window would be empty.

- **Local data can not be modified until past BEGIN**
  [Data window, Modify command]

  The local data of a procedure does not exist before the procedure entry code, to which the 'BEGIN' corresponds, has been executed.

- **No breakpoint in definition module**
  [Text window, Set breakpoint command]

  A breakpoint can only be set on a statement. Because the definition module does not contain statements, no breakponts can be set there.

- **No breakpoint to clear**
  [Text window, Clear breakpoint command]

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

- **No statement in this line**
  [Text window, Set breakpoint command]

  The selected line does not contain any statements. A breakpoint can only be set on a statement. A line that contains only a symbol like 'END', 'IF', 'CASE', 'LOOP' or similar is considered to contain no statement.

- **Not modified (new value out of range)**
  [Data and Raw windows, Modify command]

  The new value the user entered is not within the valid range. No modification has been made.

- Process decriptor can not be modified
  [Data window, Modify command]

  The run-time debugger does not allow the user to modify process descriptors. Modification of the process descriptors may cause unpredictable behavior of the program and the run-time debugger.

- Structured data cannot be modified (use Son command)
  [Data window, Modify command]

  The Modify command is applicable only when the selected data item is of a simple type. The user should invoke the Son command and select those elements or fields he wants to modify.

- This type of data can not be modified
  [Data window, Modify command]

  Procedure variables, variables of opaque types and variables of type PROCESS cannot be modified.

LOGITECH

LOGITECH MODULA-2/86

DISTRIBUTION DISKS

RUN-TIME-DEBUGGER

Modula-2/86 Run-Time Debugger, Release 1.10

Logitech SA, January 1985

Format: IBM-PC, double sided, 9 track (360K),
PC-DOS 2.0 or later

Number of disks: 1

LOGITECH Modula-2/86 Run-Time Debugger (RTD)

Release 1.10 - Dec 1984

Here is the list of the files on the RTD distribution disk,
together with some explanations what they contain:

Modula-2/86 Run-Time Support (RTS):

M2.EXE	standard RTS (same as on disk 1 of Modula-2/86 Base Language System)
M2SMALL.EXE	special version of RTS: This version of the RTS increases the maximum size of an appliaction program that can be debugged within a given memory size by 17K bytes. When using this version, the MS-DOS Command Interpreter is overwritten by the RTD and therefore we can use its memory (17K bytes) for the RTD. After termination of the RTD, DOS will reload its Command Interpreter (file COMMAND.COM) from disk. Note: The version M2SMALL has be created with the sources of the RTS (contained on disk 1 of Modula-2/86 Base Language System) by changing the constant 'KEEP_COM' in file RTS.INC from TRUE to FALSE.

Executable Modula-2/86 Run-Time Debugger (RTD):

RTD.LOD	main program of RTD
M2RTD.LOD	overlay of RTD
RTDOVLAY.LOD	overlay of RTD

The Run-Time Debugger source files:

These RTD source modules allow a user to customize the RTD
to a specific hardware (keyboard, size of RAM, etc).
For details please refer to the commented source files.
After modification of these source modules, they must be compiled
by means of the command 'M2 comp DBXINOUT', resp. 'M2 comp RTDWS'.
After successfull compilation, the main program of the RTD
must be re-linked (see below).
WARNING: DO NOT MODIFY OR RECOMPILE THESE DEFINITION MODULES!
They are included for documentation only.

DBXINOUT.DEF DBXINOUT.SYM DBXINOUT.MOD	Contains definitions for keyboard and screen.
RTDWS.DEF RTDWS.SYM RTDWS.MOD	Contains definitions for the RTD's workspace. With these definitions the RTD may be adapted to the memory size.

The Run-Time Debugger object files:

These files are needed to re-link the RTD after modification. To re-link the RTD, use the command 'M2 link rtd'. In addition to the object files listed below, the RTD needs certain modules of the Modula-2 Standard Library (contained on the distribution disk 3 ('Linker') of the Modula-2/86 Base Language System:

RTD.LNK
DBTYPE.LNK
DBXINOUT.LNK
DBXCLI.LNK
RTDX.LNK
RTDWS.LNK

The file containing this text:

READ-ME.RTD