# Modula-2 Handbook

**A Guide for Modula-2 Users and Programmers**

May 1982
October 1982 (revision)
November 1983 (PC revision)

**Lyle Bingham**
**Leo Geissman**
**Christian Jacobi**
**Svend Erik Knudsen**
**Rodney L. Riggs**
**Niklaus Wirth**

Modula
Research Institute
**950 N. University Avenue**
**Provo, Utah 84604**
**801-375-7402**

# Table of Contents

# Implementation notes for the M2M-PC Compiler

## And Interpreter running under IBM PC DOS •

**Rodney L. Riggs 3.9.83**

*These notes preface and refer to a generic version of a Modula-2 interpreter handbook. The reader may wish to familiarize himself with that handbook material (Chapters 1-8) before reading further.*

## 1. Getting Started

In addition to this handbook you should have received two diskettes labeled *M2MPC-I* and *M2MPC-II*. The following files should be found on the diskettes:

**M2MPC-I**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| SEK | ABS | COMINT | ABS | MODULA | ABS | INIT | ABS |
| PASS1 | ABS | PASS2 | ABS | PASS3 | ABS | PASS4 | ABS |
| LISTER | ABS | QLISTER | ABS | SYSTEM | SYM | CONFIG | SYS~ |
| SEK | SYM | SYMFILE | ABS | PROGRAM | SYM | INTERP | COM |
| INOUT | SYM | INOUT | OBJ | REALINOU | SYM | REALINOU | OBJ |
| MATHLIB0 | SYM | MATHLIB0 | OBJ | OUTFILE | SYM | OUTFILE | OBJ |
| CONVERSI | SYM | CONVERSI | OBJ | OPTIONS | SYM | OPTIONS | OBJ |
| FILENAME | SYM | FILENAME | OBJ | OUTTERMI | SYM | OUTTERMI | OBJ |
| BYTEBLOC | SYM | BYTEBLOC | OBJ | BYTEIO | SYM | BYTEIO | OBJ |
| STORAGE | SYM | STORAGE | OBJ | CARDINAL | SYM | TERMINAL | SYM |
| MONITOR | SYM | FILESYST | SYM | | | | |

**M2MPC-II**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| LINK | OBJ | XREF | OBJ | DECODE | OBJ | INOUT | MOD~~ |
| INSPECT | OBJ | CALENDAR | OBJ | TIME | OBJ | INOUT | DEF~~ |
| CLOCK | OBJ~ | COMMANDF | OBJ | INTERP | OBJ | INTEXT | OBJ |
| SYSTM | OBJ | NEWSYS | OBJ | READBTFL | OBJ | FLOAT | OBJ |
| ESCAPE | OBJ | LINKIN | BAT | ASMLNK | ASM | TL | MOD |
| TL | ASM | | | | | | |

~ *This file must be on the diskette used to boot the machine.*

~~ *Do not Compile!!!*

The diskettes are single sided double density. Before proceeding it is recommended that you **BACKUP BOTH DISKETTES**. To back up the diskettes put the original diskette (M2MPC-I) in drive 'A' and the backup diskette in drive 'B'. If your system uses double sided diskettes then type

```
A> copy *.* b:
```

and hit the <enter> key. When the process is finished remove M2MPC-I from drive A and insert M2MPC-II into drive A. Type the same command as above to complete the backup process.

*If your system uses* single sided diskettes then type

```
A> diskcopy a: b:\1
```

and hit the <enter> key. When the process is finished remove M2MPC-I from drive A and insert M2MPC-II into drive A. Also put the second backup diskette into drive B. Type the same command as above to complete the backup process.

The two diskettes you received do not have the resident DOS system files needed to boot up your system. For future convenience you will want to copy the distribution diskettes to a diskette(s) that also has the DOS system files resident. To do this you will first need to format a disk and install the DOS system files. This is done by typing the following:

```
A> format a:/s/[1]
```

If you are using double sided diskettes the '1' should be replaced with a '2'. After the diskette is formatted execute the backup procedure for double sided diskettes. DO NOT USE DISKCOPY, it will copy the entire source diskette and erase the just installed DOS system files from the destination diskette. There is a file, *CONFIG.SYS*, on M2MPC-I that must be on the diskette that is used to boot the system.

Now that you have a diskette with the system files and the interpreter files, you must soft boot the system so that CONFIG.SYS is executed. Make sure that the new copy of M2MPC-I is in drive 'A' and the new copy of M2MPC-II (if you have single sided disk drives) is in drive 'B'. When the diskettes are in place soft boot the system by pressing <ctrl><alt><del> simultaneously. When the prompt 'A>' appears the system is now ready to run the M2M-PC Interpreter. In the future, to run the interpreter you will only need to insert the diskettes in the proper drives and turn on the system.

On M2MPC-I you will find the executable file *INTERP.COM*, and its data file *SEK.ABS*. Both these files are necessary for the interpreter to run. If the file SEK.ABS is not on the same diskette with INTERP.COM the following error message will be displayed: *File not found, please retry....* One other necessary file is *COMINT.ABS*. If COMINT.ABS is not found, the program will search forever in a continuous loop trying to find the file. No error message is displayed. The rest of the files on M2MPC-I are support files for the Modula-2 compiler or language. The files on M2MPC-II are utility files or example files. A description of each of these files is found hereinafter.

To run the interpreter, type *interp* and hit the <enter> key. After approximately 20 seconds (for a double disk drive system, less for a hard disk system) the following will appear on the screen:

```
Comint  IV0  29.9.83

Modula - 2 Interpreter
Version 1.30 (C)Copyright Modula Research Institute 1983
*
```

The interpreter is now running and the *Command Interpreter* (COMINT.ABS) is waiting for a command. The '*' is the prompt. To run a program simply type in the program name and hit the <enter> key. The command interpreter then invokes the loader and, if found, the file is executed. This process is explained in more detail in chapter 2.

## 2. The Distribution diskettes

On your two distribution diskettes are all the files needed to run the interpreter, plus a number of programming tools in the form of modules which may be imported by other modules. The compiler consists of nine (9) files: *Modula, Init, Symfile, Pass1, Pass2, Pass3, Pass4, Lister and QLister*. Each of these files has an extension of *.ABS* (instead of the usual extension of .OBJ). They have been specially formatted to load five times faster than a normal *.OBJ* file. *.ABS* files have also been created for a few other system files to reduce execution time.

The *.MOD* and *.DEF* files for the module *InOut* have been included on the diskette, as well as the *.OBJ* and *.SYM* files, as an example of good Modula - 2 programming. These are ONLY an EXAMPLE. If you try to compile the *.DEF* file, InOut will become **INCOMPATIBLE WITH ALL OTHER UTILITY FILES ON THE DISK!** The OBJ files for the interpreter have been included to allow assembly language linking. The use of these files is discussed in chapter 9. Also found on the distribution diskettes are three demonstration programs: Clock, Time, and Calendar. *Clock* will set the PC's clock and *Time* will read the current time. *Calendar* is a simple program to demonstrate the screen output under the interpreter.

## 3. Program Creation and Execution

Currently there is not an editor available to run under the interpreter. To edit a program it is necessary to leave the interpreter and invoke the editor you normally use, then return to the interpreter (This should not be any real inconvenience because of the escape-to-the-operating-system feature explained in section 5 below). Another problem related to the absence of an editor is the creation of .LST files. Under the current system the <tab> character is handled rather clumsily. As a result, the error messages corresponding to an error in a line of source code with tabs (instead of spaces) will point to the wrong position. This is not a major problem, but is something to be aware of when correcting a program with syntax errors.

If by some accident (or on purpose) you *halt* the compiler (<ctrl-c> or <ctrl-break>) in the middle of a compilation, there will be several temporary files left on your diskette with the name *WWXXYYZZ.TM?*. These are intermediate files that the compiler creates and then renames or deletes before finishing. These files may be left alone or deleted. The compiler will recreate them new on each run regardless of whether they currently exist or not. When compiling there must be adequate space on the default diskette. If the diskette is full or almost full the compiler will hang because it is trying to write to a full diskette.

When executing a program, the object file may reside on either disk drive and in any directory. If the file is not found on the default drive in the default directory, then specify the complete path name as part of the file name (e.g. **B:\mod\calendar*). When executing a program, all imported modules must be found on the default drive in the current directory regardless of where the executable file is located. If the imported files are not found in the current directory, an error message will be displayed saying that the module(s) are not found, even though they exist on or in another drive or directory. It would be best to keep all executable files (debugged and runable) on a diskette separate from the one holding the interpreter, and all library modules on the same diskette as the interpreter. The two distribution diskettes have already been setup following this convention: the interpreter, compiler and all library modules are on M2MPC-I, and the utility and example files are on M2MPC-II.

## 4. Compiler Modifications

Instead of *.OBJ* files, *.ABS* files have been created for the compiler. The .ABS files have been restructured to significantly decrease the compile time (see section 2 above). Because of the nature of the *.ABS* files, the compiler can not be run from a command file. This is a small inconvenience when compared with a compile time that is 4 times faster than it would be with .OBJ files.

Another change in the compiler is the listing mode default value. Normally, as explained in chapter 5, the compiler automatically defaults to include the listing phase at the end of a compilation. Because of slow file IO, the compiler has been modified to default to the */nolist* option. The *nolist* option has also been changed to include a Quick-List pass. The QLister flags errors the same as the normal listing pass, but instead of writing the whole source file to the diskette with the errors marked, only the erroneous line of source code is written to the screen. Below it the error message pointing to the error is also written. By writing the lister phase to the screen instead of to a .LST file, and only writing the errors, a great amount of time is saved. If, for some reason, a normal *.LST* file is desired, the */list* option may be specified. Thus the compiler, if errors occur, always executes a listing pass whether the */list* or */nolist* option is specified. Under the nolist option (default mode) errors, if there are any, are written to the screen.

## 5. Escape to Operating System

In order to take advantage of the various utilities offered by DOS, an escape-to-the-operating-system sequence has been implemented. By typing exclamation mark (!) <enter> in response to the prompt (*) a secondary copy of the DOS command processor (see 10-9 in the DOS handbook) is invoked and the normal system prompt, 'A>', is displayed. It is now possible to execute any DOS commands including program execution, editing, copying etc. In order to facilitate this feature there must be a copy of COMMAND.COM on the default diskette. If COMMAND.COM is not found on the default disk, the error message *some load error* will appear and the normal prompt (*) will return. If you type an exclamation mark (!) followed by a DOS command, the command will be executed, if valid, and control will immediately return to the interpreter.

The only factor limiting this feature is the amount of memory in the machine. The escape-to-the-operating-system feature will not work reliably on machines with less than 192K. To return to the interpreter type *'exit'* and hit the <enter> key. Control will then return to the interpreter and the prompt (*) will be displayed. If a DOS command was entered after the exclamation mark the system prompt, 'A>', is never displayed, and *'exit'* does not need to be entered. The advantage of this feature is the saving of time. If, to get a directory listing, copy a file, rename a file, edit a program etc., it was necessary to quit the interpreter and then subsequently reinvoke it, much time would be lost. It takes approximately 20 seconds to boot the interpreter, and less than 2 seconds to return to the interpreter from the secondary copy of the command processor.

## 6. Program Name Restrictions

The DOS file system will allow file names of at most 8 characters, with an optional extension of up to three characters. Because of this restriction none of the LIBrary modules or SYStem modules will have the *LIB.* or SYS. prefixes used by other implementations. Also all modules with identifier names longer than 8 characters must be unique for the first 8 characters. The correct syntax for PC filenames follows:

```
FileName     = FileIdent .
FileIdent    = Ident { "." Extension } .
Ident        = Letter { Letter | Digit } .
Extension    = Letter { Letter | Digit } .
```

*Capital* and *lower case* letters are not treated as distinct by the DOS filer (but the compiler does make the distinction within the source code files).

## 7. File Procedures

**SetWrite, SetRead, SetOpen, SetModify, DoIo**

Because of the lack of compatibility between DOS and Medos-2, the above named procedures do not behave as described in this handbook. It would, when appropriate, be good programming practice to include these calls in your programs to keep the code compatible with other implementations of Modula-2, but these procedures are not needed on the PC; i.e. they act as dummy procedures with no effect on the files or the program.

*Final Note:*

*These instructions specify setup procedures for a system with two(2) floppy disk drives. For specifics on hard disk setup see the section in the IBM manuals on "Preparing Your Fixed Disk."*

# 1. Introduction

**Leo Geissmann 15.5.82**

**Revised Modula Research Institute 24.8.83**

This guide will give an introduction to the use of the *M-2 Interpreter* and the basic software environment running under it.

The readers of the handbook are *invited* to report detected errors to the authors. Any comments on content and style are also welcome.

## 1.1. Handbook Organization

As the range of users spans from the non-programmer, who wants only to execute already existing programs, to the active (system-) programmer, who designs and implements new programs and thereby extends the computer's capabilities, this guide is compiled such that general information is given at the beginning and more specific information toward the end. This allows the *non-programmer* to stop reading after chapter 4.

### 1.1.1. Overview of the Chapters

*Chapter 1* gives introductory comments on the handbook and on the M-2 Interpreter.

*Chapter 2* describes how programs are called with the command interpreter.

*Chapter 3* provides information about the general behaviour of programs.

*Chapter 4* is a collection of important utility programs needed by all M-2 Interpreter users.

*Chapter 5* describes the use of the Modula-2 compiler.

*Chapter 6* is a collection of library modules constituting the Medos-2 interface.

*Chapter 7* is a collection of further commonly used library modules.

*Chapter 8* describes the M-2 Interpreter-specific features of Modula-2.

## 1.2. Overview of M-2 Interpreter Software

The M-2 Interpreter allows the programming language *Modula-2*, which is defined in the Modula-2 manual [1], to be run on your machine. Some specifics of Modula-2 under the M-2 Interpreter are mentioned in chapter 8 of this handbook.

The operating system run under the interpreter is called *Medos-2*. It is responsible for program execution and general memory allocation. It also provides a general interface for input/output on files and to the terminal.

The M-2 Interpreter does not currently support a text editor. Therefore all creation and editing of files must be performed outside the interpreter.

A large number of utility programs and library modules already exist.

## 1.3. References

[1] **Programming in Modula-2**
N. Wirth, Springer-Verlag, Heidelberg, NewYork, 1982. ISBN 3-540-12206-0

[2] **The personal computer Lilith**
N. Wirth, in
- Sofware Development Environments, A.I. Wassermann, Ed., IEEE Computer Society Press, 1981.
- Proc. 5th International Conf. on Software Engineering, IEEE Computer Society Press, 1981.

# 2. Running Programs

**Svend Erik Knudsen 15.5.82**

**Revised Modula Research Institute 24.8.83**

This chapter describes, how programs are called with the *command interpreter* of the Medos-2 operating system.

## 2.1. The Command Interpreter

The *command interpreter* is the main program of the Medos-2 operating system. After the initialization of the operating system, the command interpreter *repeatedly* executes the following tasks:

- Read and interpret a command, i.e. read a program name and activate the corresponding program.
- Report errors which occured during program execution.

In order to keep the resident system small, a part of the command interpreter is implemented as a nonresident program. This fact, however, is transparent to most users of Medos-2.

### 2.1.1. Program Call

The command interpreter indicates by an asterisk * that it is ready to accept the next command. Actually, there exists only one type of command: *program calls.*

To call a program, type a program name on the keyboard and terminate the input by hitting the RETURN key.

```
*time
```

The program with the typed name is activated; i.e. loaded and started for execution. If the program was executed correctly, the command interpreter returns with an asterisk and waits for the next program call. If some load or execution error occured, an error message is displayed, before the asterisk appears.

```
*timex
 program not found
*time
 17.8.83   16:2:45                time program is running
*
```

A *program name* is an identifier or a sequence of identifiers separated by periods. An identifier itself begins with a letter (A .. Z, a .. z) followed by further letters or digits (0 .. 9). At most 16 characters are allowed for a program name, and capital and lower case letters are treated as distinct.

```
ProgramName    = Identifier {"."Identifier}.
Identifier     = letter { letter | digit }.
```

Programs are loaded from files on the disk cartridge. In order to find the file from which the program should be loaded, the Medos-2 loader converts the program name into a file name by appending the extension OBJ and searches for a file with this name. If no such file exists, the loader inserts the prefix SYS into the file name and searches for a file with this name

| | |
|---|---|
| Accepted name | `time` |
| First file name | `time.OBJ` |
| Second file name | `SYS.time.OBJ` |

If neither of the searched files exists, the command interpreter displays the error message `program not found`.

### 2.1.2. Typing Aids

The command interpreter provides some typing aids which make the calling of a program more convenient. Most typing errors are handled by simply *ignoring unexpected characters*. There are also some *special keys*.

*Special Keys*

While typing a program name, the command interpreter accepts some special keys which are immediately executed. These special keys and their definitions follow:

DEL
: Delete the last typed character.

CTRL-X
: *Cancel*. Delete the whole character sequence which has been typed

CTRL-L
: *Form feed*. Clear the screen and accept a new command at the upper left corner of the screen. This key must be typed directly *after an asterisk*. It is not accepted within a character sequence.

CTRL-C
: *Kill character*. This key may be typed at any time. The currently executed program will be *killed* and control will be returned to the computers original operating system. CTRL-C is NOT THE NORMAL WAY TO LEAVE A PROGRAM.

### 2.1.3. Loading and Execution Errors

Messages about loading and execution errors are displayed on the screen. They are reported either by the command interpreter, the resident system, or the running program itself.

*Loading Errors*

It is possible that a called program cannot be loaded. It may be that the corresponding file is not found, that some separate modules imported by the program are not found, or that the module keys of the separate modules do not match.

The following types of loading errors may be reported

| | |
|---|---|
| call error | *parameter error at program call* |
| program not found | |
| program already loaded | *a program must not be loaded twice* |
| module not found | |
| incompatible module | *a module found with a wrong module key* |
| not enough space | *program needs too much memory space* |
| too many modules | *maximal number of loaded modules exceeded* |
| illegal type of code | *code of a module is not from the same generation* |
| error in filestructure | *a file may be damaged* |
| some file error | |
| some load error | *maximal number of imported, not yet loaded modules exceeded* |

*Execution Errors*

If a program is successfully loaded, it is possible that the execution of the program is terminated abnormally. A run time overflow may occur or the program may call the standard procedure HALT.

The following types of execution errors may be reported

| | |
|---|---|
| stack overflow | *available memory space exceeded* |
| REAL overflow | |
| CARDINAL overflow | |
| INTEGER overflow | |
| range error | |
| address overflow | *illegal pointer access* |
| function return error | *function not terminated by a RETURN statement* |
| priority error | *call of a procedure on lower priority* |
| HALT called | *standard procedure HALT was called* |
| assertion error | *program terminated with an assertion error* |
| instruction error | *illegal instruction, i.e. the code may be overwritten* |
| warning | *program detected some unexpected errors -- no memory dump* |

*Errors Reported by the Command Interpreter*

The error messages displayed by the command interpreter are intended to be self-explanatory. They are written just before the asterisk which indicates that the next command will be accepted.

*Errors Reported by the Resident System*

The messages directly displayed by the resident system (and possibly other non-resident modules and programs) appear according to the following example:

```
- Storage.ALLOCATE: heap overflow
```

This example indicates that procedure `ALLOCATE` in module `Storage` had detected that the requested space could not be allocated in the heap.

Some modules (e.g. module *Program*) indicate on which execution level the error was detected by the number of hyphens in front of the message.

*Errors Reported by Other Programs*

It is possible that other programs report loading and execution errors in their own manner.

## 2.2. Command Files

It is possible that a sequence of program executions must be repeated several times. Consider for example the transfer of a set of files between two computers. Instead of typing all commands interactively, it is in this case more appropriate to substitute these commands as a batch to the procedures which normally read characters from the keyboard. For this purpose the operating system allows the substitution of *command files*.

A command file must contain exactly the same sequence of characters which originally would be typed on the keyboard. This includes the commands to call programs and the answers given in the expected dialog with the called programs. To initialize the command file input, the program *commandfile* must be started. This program prompts for the name of a command file (default extension is `COM`) and substitutes the accepted file to the input procedures.

```
*commandfile
 Command file> transfer.COM
*
```

*input characters are read from the command file, instead of from the keyboard*

After all characters have been read from the substituted command file, the input is read again from the keyboard. Reading from the command file is also stopped when a program does not load correctly or a program terminates abnormally.

With one exception, command files *must not be nested.* If the call of program *commandfile* and the subsequent file name are the last information on the current command file, it is possible to start a new command file. In all other cases the execution of the current command file would fail.

## 2.3. Program Loading

**This chapter is intended to be read by programmers only.**

Programs are normally executed on the top of the resident operating system. After the program name is accepted by the command interpreter, the loader of Medos-2 loads the program into the memory and, after successful loading, starts its execution. Medos-2 also allows a program to call another program. This chapter describes, how programs are loaded on the top of Medos-2. More details about program calls, program loading, and program execution are given in the description of module *Program* (see chapter 6.2.).

Usually, a program consists of several separate modules. These are the *main module*, which constitutes the main program, and all modules which are, directly or indirectly, imported by the main module.

Upon compilation of a separate module, the generated code is written on an *object file* (extension `OBJ`). This file can be accepted by the loader of Medos-2 directly. A program is ready for execution if it and all imported modules are compiled. To execute the program, the main module must be called. The loader will first load the main module from the substituted object file, and afterwards the imported modules from their corresponding object files.

The names of the object files belonging to the imported modules are derived from the module names (the number of unique characters in the module name depends upon the implementation). If a first search is not successful, a prefix `LIB` is inserted into the file name and the loader tries again to find the object file.

| | |
|---|---|
| Module name | `BufferPool` |
| First file name | `BufferPool.OBJ` |
| Second file name | `LIB.BufferPool.OBJ` |

A module cannot be loaded twice. If an imported module is already loaded with the resident system (e.g. module *FileSystem*), the loader connects the program with this module.

If a module cannot be loaded because of a missing object file, a loading error is signalled. The loader also signals an error if a module found on an object file is incompatible with the other modules. For correct program execution, it is important that the references across the module boundaries refer to the same interface descriptions, i.e. the same symbol file versions of the separate modules. The compiler generates for each separate module a *module key* (see chapter 5.7.) which is also known to the importing modules. For successful loading, all module keys refering to the same module must match.

After termination of the program, the memory space occupied by the previously loaded modules is released. This also happens with the resources used by the program (e.g. heap, files).

The loading speed may be improved if a program is *linked* before its execution. The linker collects the imported modules in the same manner as the loader and writes them altogether on one file. It is also possible, to substitute a user selected file name for an imported module to the linker. If a program is linked, the loader can read all imported modules from the same object file, and therefore it is not necessary to search for other object files. For a description of program *link* refer to chapter 4.7.

# 3. Things to Know

**Leo Geissmann 15.5.82**

**Revised Modula Research Institute 24.8.83**

This chapter provides you with information about different things which are worth knowing if you want to get along with M2-Interpreter. There are some conventions which have been observed when utility programs or library modules were designed. Knowing these should allow you to be more familiar with the behavior of the programs.

## 3.1. Special Keys

Consider the following situations: you want to stop the execution of your program, because something is going wrong; or, you want to cancel your current keyboard input, because you typed a wrong key; or, you want to get information about the active commands of a program because you actually forgot them. In all these situations it is very helpful to know a way out.

For these problems, several keys on the keyboard can have a special meaning, when they are typed in an appropriate situation. Some of these special keys are always active, others have their special meaning only if a program is ready to accept them. The following list should give you an idea of which keys are used for what features in programs and to invite you to use the same meanings for the special keys in your own programs.

**DEL**

Key to delete the last typed character in a keyboard input sequence. This key is active in most programs when they expect input from keyboard.

**CTRL-X**

Key to cancel the current keyboard input line. This key is active in special situations, e.g. when a file name is expected by a program.

**ESC**

Key to tell the running program that it should terminate more or less immediately in a soft manner. This key is active in most programs when they expect input from keyboard.

**CTRL-C**

Key to stop the execution of a program immediately. Typing CTRL-C is useful if the actions of a program are no longer under control. Nevertheless it is considered bad taste to terminate a program in this way.

**CTRL-L**

Key to clear the screen area on which a program is writing. This key is active in special situations, e.g. when the command interpreter is waiting for a new program name.

## 3.2. File Names

### 3.2.1. File Names Accepted by the Module FileSystem

Most programs work with files. This means that they have to assign files on a device. For this purpose the module *FileSystem* provides some procedures to identify files by their names. File names accepted by these procedures have the following syntax:

```
FileName            = FileIdent .
FileIdent           = Ident { "." Ident } .
Ident               = Letter { Letter | Digit } .
```

*Capital* and *lower case letters* are treated as distinct.

FileIdent means the name of a file under which it is registered in the name directory of the device.

### 3.2.2. File Name Extensions

The syntax of a FileIdent, with identifiers separated by periods, allows structuring of the file names. On Lilith, the following rule is respected by programs dealing with file names:

> The last identifier in a FileIdent is called the *extension* of the file name. If a FileIdent consists of just one identifier, then this is the extension.

File name extensions allow file categorization of specific types i.e. OBJ for object code files, SYM for symbol files. There are programs such as the compiler which automatically set the extension, when they generate new files.

### 3.2.3. File Name Input from Keyboard

Many programs prompt for the names of the files they work with. In this case you have to type a file name from the keyboard according to the following syntax:

```
InputFileName       = FileIdent .
```

Many programs offer a default file name or a default extension when they expect the specification of a file name. So, it is possible to solely press the **RETURN** key to specify the whole default file name, or to press the **RETURN** key after a period to specify the default extension.

*For programmers:* Module *FileNames* supports the reading of file names.

## 3.3. Program Options

To run correctly, programs often need, apart from a file name, some additional information which must be supplied by the user. For this purpose so-called *program options* are accepted by the programs. Program options are an appendix typed after the file name. The following syntax is applied.

```
FileNameAndOptions = InputFileName { ProgramOption } .
ProgramOption       = "/" OptionValue .
OptionValue         = { Letter | Digit } .
```

Every program has its own set of program options, and often a default set of OptionValues is valid. This has the advantage that for frequently used choices no options must be specified explicitly.

```
Harmony.MOD/query/nolist
```

*For programmers:* Module *Options* supports the reading of program options.

• *Depending on the machine, there are different rules for the filenames. Please see release notes for implementation variations.*

# 4. Utility Programs

**15.5.82**

This chapter gives an overview of some utility programs which provide important services under the interpreter. Utility programs are stored on the disk. Programs are called for execution by their name.

*List of the Programs*

| | | |
|---|---|---|
| inspect | Inspect the contents of a file | 4.1. |
| xref | Generate a reference list of a text file | 4.2. |
| link | Link separate modules to a program | 4.3. |
| decode | Disassembles object files | 4.4. |

Most programs operate on files; they will therefore prompt for a *file name* and probably also accept *program options*. The syntax of file names and program options is given in chapter 3.

## 4.1. inspect

Peter Lamb 15.5.82
Revised Modula Research Institute 24.8.83

The program *inspect* displays the contents of a file in several formats on the screen. It is normally used to inspect files consisting of encoded information much like an editor. The program *repeatedly* prompts for a file name and for program options.

```
inspect> Salary.DATA/octal
```

If the file name is not specified, the previously accepted name is used. If no program options specifying the output format are given, the previous format is used. The default output format at the beginning is set according to the program options Octal and Word.

If more than one display format (Ascii, Octal or Hexadecimal) is given, each dumped item will be displayed in each of the formats given. For example

```
inspect> /byte/ascii/hex
```

will display bytes as both ASCII characters and hexadecimal numbers.

ASCII codes from 0C to 40C are displayed as the corresponding control code (1C is displayed as ↑A). ASCII codes >= 177C are displayed as octal numbers.

The leftmost column of the output is the *address of the data* and is in octal, unless program option Hexadecimal has been used, and then it is in hexadecimal. Unless program option OUtput is used, the dump will appear on the screen.

The output may be paused by typing any character except ESC or CTRL-C and restarted by typing another character. Typing ESC will stop the printout and ask for another file to dump.

*Program options*

**Byte**
: Information on file is displayed as a sequence of bytes.

**Word**
: Information on file is displayed as a sequence of words. *Default.*

**Ascii**
: Displayed values are represented as ASCII characters.

**Octal**
: Displayed values are represented as octal numbers. *Default.*

**Hexadecimal**
: Displayed values are represented as hexadecimal numbers.

**Startaddress**
: Information is displayed from this file position. Will prompt for specification of the start position. Default value is the beginning of the file.

**Endaddress**
: Information is displayed until this file position. Will prompt for specification of the start position. Default value is the end of the file.

**OUtput**
: Information is written on an output. Will prompt for a file name.

**HELP**
: Program will display information concerning its operation.

Capitals mark the abbreviations of the option values.

## 4.2. xref

Leo Geissmann 15.5.82
Revised Modula Research Institute 24.8.83

Program *xref* generates *cross reference information tables* of text files, especially of Modula-2 compilation units.

The program reads a text file and generates a table with line number references to all identifiers occuring in the text. It respects the Modula-2 syntax. This means that all word symbols of Modula-2 are omitted from the table. The program also skips *strings* (enclosed by quote marks " or apostrophes ') and *comments* (from (* to the corresponding *)).

The program prompts for the name of the input file. Default extension is LST.

```
*xref
 input file> BinaryTree.LST
```

The generated table is listed on a *reference file* in alphabetical order. In identical character sequences, capitals are defined *greater* than lower case letters.

If the lines on the input file start with a number, these numbers are taken as referencing line numbers, otherwise a *listing file* with line numbers is generated (see also program options L and N).

The names of the output files are derived from the input file name with the extension changed as follows

XRF for the reference file
LST for the listing file

*Program Options*

**S**

Display statistics on the terminal.

**L**

Generate a listing file with *new* line numbers.

**N**

Generate no listing file. The line numbers in the reference table will refer to the line numbers on the input file. All lines on the input file without leading line numbers are skipped (e.g. error message lines).

## 4.3. link

Svend Erik Knudsen 15.5.82
Revised Modula Research Institute 24.8.83

The program *link* collects the codes of separate modules of a program and writes them on one file. The program *link* is called *linker* in this chapter. Upon compilation of a separate module, the code generated by the Modula-2 compiler is written to an *object file*. An object file may be loaded by Medos-2 directly.

As a program usually consists of several separate modules, the loader reads the code of the modules from several object files which are searched according to a *default strategy*. On the one hand, this is time consuming because several files must be searched, on the other hand, it allows subsitution of a module from a file with a non-default name.

The linker simulates the loading process and collects the codes of all (nonresident) modules which are, directly or indirectly, imported by the so-called *main module*, i.e. the module which constitutes the main program. The linker applies the same default strategy as the loader to find an object file. A file name is derived from the module name (the number of unique characters depends upon the implementation). If a first search is not successful, the prefix `LIB` is inserted into the file name, and a file with this name is searched.

| | |
|---|---|
| Module name | `Options` |
| First default file name | `Options.OBJ` |
| Second default file name | `LIB.Options.OBJ` |

The linker first prompts for the object file of the main module (default extension `OBJ`). Next, it displays the name of the main module. If the file already contains some linked modules, the names of these modules are displayed next. Afterwards, a name of a not yet linked imported module is displayed, followed by the file name of the corresponding object file. On the next lines the names of the modules linked from this file are listed. This is repeated until all imported modules are linked.

```
*link
 Linker V3.1 for MEDOS-2 V3
 object file> delete.OBJ
      Delete                                main module
 NameSearch: LIB.NameSearch.OBJ             second default file name
      NameSearch
~Options: Options.OBJ                       first default file name
      Options
      FileNames                             module was linked to Options
 end of linkage
```

After successful linking, all linked modules are written on the object file of the main module!

The linker accepts the program option `Q` (query) when it prompts for the main module. If this option is set, the linker also prompts for the file names of the imported modules. Type a file name (default extension `OBJ`) or simply press the **RETURN** key to apply the default strategy. A prompt is repeated until an adequate object file is found, or the **ESC** key is pressed. The latter means that this module should not be linked. With the query option the linker also asks whether or not a module on a object file should be linked. Type **y** or **RETURN** to accept the module, otherwise type **n**.

```
 object file> delete.OBJ/q                  query option set
      Delete
 NameSearch> NameSearch.new.OBJ             own file substituted
      NameSearch   ?  yes
 Options> Options.OBJ                       default file name
      Options      ?  yes
      FileNames    ?  no                    module not linked from this file
 FileNames> FileNames.own.OBJ
      FileNames    ?  yes
```

## 4.4. decode

Christian Jacobi 10.5.82
Revised Modula Research Institute 24.8.83

Program *decode* disassembles an object file.

The program reads an object code file and generates a textfile with mnemonics for the machine instructions. It respects the structure of the object file as generated from the compiler.

The program prompts for the name of the input file. The default extension is OBJ.

```
*decode
 decode > program.OBJ
```

The name of the output file is derived from the input file name with the extension changed to DEC.

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

# 5. The Compiler

**Leo Geissmann 15.5.82**

**Revised Modula Research Institute 24.8.83**

This chapter describes the use of the Modula-2 compiler. For the language definition refer to *Programming in Modula-2* (see 1.3). M2-Interpreter specific language features are mentioned in chapter 8 of this handbook.

## 5.1. Glossary and Examples

*Glossary*

**compilation unit**
: Unit accepted by compiler for compilation, i.e. definition module or program module (see Modula-2 syntax in [1]).

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

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

**source file**
: Input file of the compiler, i.e. a compilation unit. The default extension is **MOD**.

**listing file**
: Compiler output file with list of the compiled unit. The assigned extension is **LST**.

**symbol file**
: Compiler output file with symbol table information. This information is generated during compilation of a definition module. The assigned extension is **SYM**.

**reference file**
: Compiler output file with debugger information, generated during compilation of a program module. The assigned extension is **REF**.

**object file**
: Compiler output file with the generated M-code in loader format. Assigned extension is **OBJ**.

*Examples*

The examples given in this chapter to explain compiler execution refer to following compilation units:

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

MODULE Prog2;
BEGIN
  a := 2
END PROG2.

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

```
IMPLEMENTATION MODULE Prog3;
  IMPORT Storage;
  ...
END Prog3.
```

## 5.2. Compilation of a Program Module

The compiler is called by typing ***modula***. After displaying the string `source file>` the compiler is ready to accept the filename of the compilation unit to be compiled.

```
*modula
source file> Prog1.MOD              name Prog1.MOD is accepted
p1
p2                                  the succession of the activated
p3                                  compiler passes is indicated
p4
lister
end compilation
*
```

Default extension is MOD.

If syntactic errors are detected by the compiler, the compilation is stopped after the third pass and a listing file with error messages is generated.

```
*modula
source file> Prog2.MOD
p1
 ---- error                         error detected by pass1
p2
p3
 ---- error                         error detected by pass3
lister
end compilation
*
```

## 5.3. Compilation of a Definition Module

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

```
*modula
source file> Prog3.DEF              definition module
p1
p2
symfile
lister
end compilation
*
```

## 5.4. Symbol Files Needed for Compilation

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

At compilation of the implementation part of the module.

At compilation of another unit, importing objects from this separate module.

According to a program option, set when the compilation is started (see chapter 5.6.), the compiler either explicitly prompts for the names of the needed symbol files, or searches for a needed symbol file (extension SYM) by a default name, which is constructed from (the first 16 characters of) the module name. In the former case the query for a symbol file is repeated until an adequate file is found or the ESC key is typed. If in the latter case the search fails, the default name is combined with a prefix LIB and the compiler tries again to find a corresponding file. A second failure would cause an error message.

| | |
|---|---|
| Module name | `Storage` |
| Object of First file name search | `Storage.SYM` |
| Object of Second file name search | `LIB.Storage.SYM` |

If all needed symbol files are not available, the compilation process is stopped immediately.

```
*modula
source file> Prog3.MOD                    implementation module
p1
 Prog3: Prog3.SYM
 Storage: LIB.Storage.SYM
p2
p3
p4
lister
end compilation
*
```

## 5.5. Compiler Output Files

Several files are generated by the compiler. They get the same file name as the source file with an extension changed as follows

`LST` listing file
`SYM` symbol file
`REF` reference file
`OBJ` object file

The reference file may be used by a debugger to obtain names of objects.

## 5.6. Program Options for the Compiler

When reading the source file name, the compiler also accepts some program options from the keyboard. Program options are marked with a leading character / and must be typed sequentially after the file name (see chapter 3.).

The compiler accepts the option values:

**LIST**

A listing file must be generated.

**N**

No listing file must be generated. *Default.*

**Q**

the compiler explicitly promps for the names of the needed symbol files, belonging to modules imported by the compiled unit.

**NOQ**

No query for symbol file names. Files are searched corresponding to a default strategy. *Default.*

V

The compiler has to display information about the running version of processor and operating system flags.

## 5.7. Compilation Options in Compilation Units

Comments in a Modula-2 compilation unit may be used to specify certain *compilation options* for tests.

The following syntax is accepted for compilation options:

```
CompOptions   = CompOption { "," CompOption } .
CompOption    = "$" Letter Switch .
Switch        = "+" | "-" | "=" .
```

Compilation options must be the first information within a comment. They are not recognized by the compiler, if other information precedes the options.

*Letter*

R Subrange and type conversion test.
T Index test (arrays, case).

*Switch*

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

All switches are set to + by default.

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

## 5.8. Module Key

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

For an implementation module the key of the associated definition module is adopted. The module keys of imported modules are also recorded on the generated symbol files and the object files.

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

### WARNING

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

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

## 5.9. Program Execution

Programs are normally executed on the top of the resident operating system *Medos-2*. The *command* *interpreter* accepts a program name and causes the *loader* to load the module on the corresponding object file into the memory and to start its execution.

If a program consists of several separate modules, no explicit linking is necessary. The object files generated by the compiler are merely ready to be loaded. The *main module*, the module which is called to be executed and therefore constitutes the main program, as well as all modules which are directly or indirectly imported, is loaded, . The loader establishes the links between the modules and organizes the initialization of the loaded modules.

Usually some of the imported modules are part of the already loaded, resident, Medos-2 operating system (e.g. module FileSystem). In this case the loader sets up the links to these modules, but prohibits their reinitialization. A module cannot be loaded twice.

After termination of the program, all separate modules which have been loaded together with the main module are removed from the memory. More details concerning program execution are given in chapter 2.

Although it is not necessary to link programs explicitly, it is sometimes more convenient to do so. Linking collects all modules which are to be loaded together and writes them to the same file. If a program is pre-linked, it will accelerate the loading. Linking is provided by the program *link* (see chapter 4.7).

Medos-2 also supports a type of *program stack*. A program may call another program, which will be executed on the top of the calling program. After termination of the called program, control will be returned to the calling program. For more details refer to the library module *Program* (see chapter 6.2.).

## 5.10. Value Ranges of the Standard Types

The value ranges of the Modula-2 standard types under the interpreter are defined according to a 16 bit word size.

`INTEGER`

The value range of type `INTEGER` is `[-32768..32767]`. Sign inversion is an operation within constant expressions. Therefore the compiler does not allow the direct definition of `-32768`. This value must be computed indirectly; for example: `-32767-1`.

`CARDINAL`

The value range of type `CARDINAL` is `[0..65535]`.

`REAL`

Values of type `REAL` are represented in 2 words. The value range expands from `-1.7014E38` to `1.7014E38`.

`CHAR`

The character set of type `CHAR` is defined according to the ISO - ASCII standard with ordinal values in the range `[0..255]`. The compiler processes character constants in the range `[0C..377C]`.

`BITSET`

The type `BITSET` is defined as `SET OF [0..15]`. Consider that sets are represented from the high order bits to the low order bits, i.e. {15} corresponds to the ordinal value 1.

## 5.11. Differences and Restrictions

For the implementation of Modula-2 under the interpreter some differences and restrictions must be considered.

**Constant expressions with real numbers**

Constant expressions with real numbers are *not* evaluated by the compiler (except sign inversion). The compiler generates an error message.

**Character arrays**

In arrays with element type CHAR two characters are packed into one word. This implies the restriction that a variable parameter of type CHAR *may not* be substituted by an element of a character array.

**Sets**

Maximal ordinal value for set elements is 15.

**FOR statement**

The values of both expressions of the for statement must not be greater than 32767 (77777B). The values are checked at run time, if the compilation option **R+** is specified. The step must be within the range [-128..127], the value 0 excepted.

**CASE statement**

The labels of a case statement must not be greater than 32767 (77777B).

**Value ARRAY OF WORD parameter**

Constants (with the exception of constant strings) must not be substituted for a value dynamic ARRAY OF WORD parameter.

**Function procedures**

The *result type* of a function procedure must neither be a record nor an array.

## 5.12. Compiler Error Messages

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

20 : identifier expected
21 : integer constant expected
22 : ']' expected
23 : ';' expected
24 : block name at the END does not match
25 : error in block
26 : ':=' expected
27 : error in expression
28 : THEN expected
29 : error in LOOP statement
30 : constant must not be CARDINAL
31 : error in REPEAT statement
32 : UNTIL expected
33 : error in WHILE statement
34 : DO expected
35 : error in CASE statement
36 : OF expected
37 : ':' expected
38 : BEGIN expected
39 : error in WITH statement
40 : END expected
41 : ')' expected
42 : error in constant
43 : '=' expected
44 : error in TYPE declaration
45 : '(' expected
46 : MODULE expected
47 : QUALIFIED expected
48 : error in factor
49 : error in simple type
50 : ',' expected
51 : error in formal type
52 : error in statement sequence
53 : '.' expected
54 : export at global level not allowed
55 : body in definition module not allowed
56 : TO expected
57 : nested module in definition module not allowed
58 : '}' expected
59 : '..' expected
60 : error in FOR statement
61 : IMPORT expected

70 : identifier specified twice in importlist
71 : identifier not exported from qualifying module

72 : identifier declared twice
73 : identifier not declared
74 : type not declared
75 : identifier already declared in module environment
76 :
77 : too many nesting levels
78 : value of absolute address must be of type CARDINAL
79 : scope table overflow in compiler
80 : illegal priority
81 : definition module belonging to implementation not found
82 : structure not allowed for implementation of hidden type
83 : procedure implementation different from definition
84 : not all defined procedures or hidden types implemented
85 : name conflict of exported object or enumeration constant in environment
86 : incompatible versions of symbolic modules
87 :
88 : function type is not scalar or basic type
89 :
90 : pointer-referenced type not declared
91 : tagfieldtype expected
92 : incompatible type of variant-constant
93 : constant used twice
94 : arithmetic error in evaluation of constant expression
95 : incorrect range
96 : range only with scalar types
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

200 : compiler error
201 : implementation restriction
202 : implementation restriction: for step too large
203 : implementation restriction: boolean expression too long
204 : implementation restriction: expression stack overflow,
i.e. expression too complicated or too many parameters
205 : implementation restriction: procedure too long
206 : implementation restriction: packed element used for var parameter
207 : implementation restriction: illegal type conversion

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

# 6. The Medos-2 Interface

**Svend Erik Knudsen 15.5.82**

**Revised Modula Research Institute 24.8.83**

This chapter describes the interface to the Medos-2 operating system. It contains the following modules:

# 6.1. Module FileSystem

**Svend Erik Knudsen 15.5.82**

## 6.1.1. Introduction

A (Medos-2) file is a sequence of bytes stored on a certain medium. Module *FileSystem* is the interface the normal programmer should know in order to use files. The definition module is listed in chapter 6.1.2. The explanations needed for simple usage of sequential (text or binary) files are given in chapter 6.1.3. The file system supports several implementations of files.

## 6.1.2. Definition Module FileSystem

```
DEFINITION MODULE FileSystem;    (* Medos-2 V3  S. E. Knudsen  1.6.81 *)

  FROM SYSTEM IMPORT ADDRESS, WORD;

  EXPORT QUALIFIED
    File, Response,

    Create, Close,
    Lookup, Rename,
    SetRead, SetWrite, SetModify, SetOpen,
    Doio,
    SetPos, GetPos, Length,

    Reset, Again,
    ReadWord, WriteWord,
    ReadChar, WriteChar,


  TYPE
    Response      = (done, notdone, notsupported, callerror,
                      unknownmedium, unknownfile, paramerror,
                      toomanyfiles, eom, deviceoff,
                      softparityerror, softprotected,
                      softerror, hardparityerror,
                      hardprotected, timeout, harderror);

    File          = RECORD
                        id: CARDINAL;
                       eof: BOOLEAN;
                       res: Response;
                     END;


  PROCEDURE Create(VAR f: File;  mediumname: ARRAY OF CHAR);
  PROCEDURE Close(VAR f: File);

  PROCEDURE Lookup(VAR f: File; filename: ARRAY OF CHAR; new: BOOLEAN);
  PROCEDURE Rename(VAR f: File; filename: ARRAY OF CHAR);

  PROCEDURE ReadWord(VAR f: File; VAR w: WORD);
  PROCEDURE WriteWord(VAR f: File; w: WORD);
  PROCEDURE ReadChar(VAR f: File; VAR ch: CHAR);
  PROCEDURE WriteChar(VAR f: File; ch: CHAR);
```

```
  PROCEDURE Reset(VAR f: File);
  PROCEDURE Again(VAR f: File);
  PROCEDURE SetPos(VAR f: File; highpos, lowpos: CARDINAL);
  PROCEDURE GetPos(VAR f: File; VAR highpos, lowpos: CARDINAL);
  PROCEDURE Length(VAR f: File; VAR highpos, lowpos: CARDINAL);

  PROCEDURE SetRead(VAR f: File);
  PROCEDURE SetWrite(VAR f: File);
  PROCEDURE SetModify(VAR f: File);
  PROCEDURE SetOpen(VAR f: File);
  PROCEDURE Doio(VAR f: File);

END FileSystem.
```

## 6.1.3. Simple Use of Files

### 6.1.3.1. Opening, Closing, and Renaming of Files

A file is either *permanent* or *temporary*. A permanent file remains stored on its medium after it is closed and normally has an external (or symbolic) name. A temporary file is removed from the medium as soon as it is no longer referenced by a program, and normally it is nameless. Within a program, a file is referenced by a variable of type *File*. From the programmer's point of view, the variable of type *File* simply is the file. Several routines connect a file variable to an actual file (e.g. on a disk). The actual file either has to be *created* or *looked up* by its file name. The syntax of *file name* is

```
identifier   = letter { letter | digit } .

file name    = local name .
local name   = identifier { "." identifier } .
```

Capital and lower case letters are treated as being different. The local name is the name of the file on a specific medium. The last (and maybe the only) identifier within a local file name is often called the *file name extension* or simply *extension*. The file system does, however, *not* treat file name extensions in a special way. Many programs and users use the extensions to classify files according to their content and treat extensions in a special way (e.g. assume defaults, change them automatically, etc.).

SYS.directory.OBJ

File name of file *SYS.directory.OBJ*. Its extension is *OBJ*.

Create(f, mediumname)

Procedure *Create* creates a new temporary (and nameless) file. *mediumnam* is a dummy parameter left over from the Lilith. After the call the variable f.res has the following value:

| | |
|---|---|
| f.res = done | if file f is created, |
| f.res = ... | if some error occured. |

Close(f)

Procedure *Close* terminates any actual input or output operation on file f and disconnects the variable f from the actual file. If the actual file is temporary, *Close* also deletes the file.

Lookup(f, filename, new)

Procedure *Lookup* looks for the actual file with the given file name. If the file exists, it is connected to f (opened). If the requested file is not found and new is TRUE, a permanent file is created with the given name. After the call the variable f.res has the following value:

| | |
|---|---|
| f.res = done | if file f is connected, |
| f.res = notdone | if the named file does not exist, |
| f.res = ... | if some error occured. |

Rename(f, filename)

Procedure *Rename* changes the name of file f to filename. If filename is empty, f is changed to a temporary and nameless file. If filename contains a local name, the actual file will be permanent after a successful call of *Rename*. After the call the variable f.res has the following value:

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

*Related Module*

Module *FileNames* makes it easier to read file names from the keyboard (i.e. from module *Terminal*, see chapter 6.4.) and to handle defaults (see chapter 7.11.). I

#### 6.1.3.2. Reading and Writing of Files

At this level of programming, we consider a file to be either a sequence of characters (text file) or a sequence of words (binary file), although this is *not* enforced by the file system. The first called routine causing any input or output on a file (i.e. ReadChar, WriteChar, ReadWord, WriteWord) determines whether the file is to be considered as a text or a binary file.

Characters read from and written to a text file are from the ASCII set. Lines are terminated by character 36C (= *eol, RS*).

Reset(f)

Procedure *Reset* terminates any actual input or output and sets the *current position* of file f to the beginning of f.

WriteChar(f, ch), WriteWord(f, w)

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

ReadChar(f, ch), ReadWord(f, w)

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

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

If f.eof is TRUE:

| | |
|---|---|
| f.res = done | *End of file* has been reached |
| f.res = ... | Some error occured |

Again(f)

This procedure is not supported by the M-2 Interpreter.

*Related Modules*

Module *ByteIO* provides routines for reading and writing of bytes on files. This is valuable for the packing of information on files, if it is known that the ordinal values of the transferred elements are in the range 0 .. 255.

Module *ByteBlockIO* makes it easier (and more efficient) to transfer elements of any given type (size).

#### 6.1.3.3. Positioning of Files

All input and output routines operate at the *current position* of a file. After a call to *Lookup*, *Create* or *Reset*, the current position of a file is at its beginning. Most of the routines operating upon a file change the current position of the file as a normal part of their action. Positions are encoded into *long cardinals*, and a file is positioned at its beginning, if its current position is equal to zero. Each call to a procedure, which reads or writes a character (a word) on a file, increments the current file position by 1 (2) for each character (word) transferred. A character (word) is stored in 1 (2) byte(s) on a file, and the position of the element is the number of the (first) byte(s) holding the element. By aid of the procedures *GetPos*, *Length* and *SetPos* it is possible to get the current position of a file, the position just after the last element in the file, and to change explicitly the current position of a file.

SetPos(f, highpos, lowpos)

A call to procedure *SetPos* sets the current position of file f to *highpos * 2**16 + lowpos*. The new position must be less or equal the length of the file. If the last operation before the call of *SetPos* was a write operation (i.e. if file f is in the writing state), the file is cut at its new current position, and the elements from current position to the end of the file are lost.

GetPos(f, highpos, lowpos)

Procedure *GetPos* returns the current file position. It is equal to *highpos * 2**16 + lowpos.*

Length(f, highpos, lowpos)

Procedure *Length* gets the position just behind the last element of the file (i.e. the number of bytes stored on the file). The position is equal to *highpos * 2**16 + lowpos.*

### 6.1.3.4. Examples

Writing a Text File

```
  .
VAR
  f: File;
  ch: CHAR; endoftext: BOOLEAN;
  .
  .
Lookup(f, "newfile", TRUE);
IF f.res <> done THEN
  (* f was not created by this call to "Lookup" *)
  IF f.res = done THEN Close(f) END
ELSE
  LOOP
    (* find next character to write --> endoftext, ch *)
    IF endoftext THEN EXIT END;
    WriteChar(f, ch)
  END;
  Close(f)
END
  .
  .
```

Reading a Text File

```
  .
VAR
  f: File;
  ch: CHAR;
  .
  .
Lookup(f, "oldfile", FALSE);
IF f.res <> done THEN
  (* file not found *)
ELSE
  LOOP
    ReadChar(f, ch);
    IF f.eof THEN EXIT END;
    (* use ch *)
  END;
  Close(f)
END
  .
  .
```

SetOpen(f) *

A call to *SetOpen* flushes all changed buffers assigned to file f, and the file is set into state *opened.* A call to *SetOpen* is needed only if it is desirable for some reason to flush the buffers (e.g. within database systems or for "replay" files), or if the file is in state *writing,* and it has to be positioned backward without truncation. If an I/O error occured since the last time the file was in state *opened,* this is indicated by field *res.*

| | |
|---|---|
| f.res = done | Previous I/O operations successful |
| f.res = ... | An error has occured since the last time the file was in state *opened.* |

SetRead(f) *

A call to *SetRead* sets the file into state *reading*. This implies that a buffer is assigned to the file and the byte at the current position is in the assigned buffer.

SetWrite(f) *

A call to *SetWrite* sets the file into state *writing*. In this state, the length of a file is *always (set)* equal to its current position, i.e. the file is *always* written at its end, and the file will be *truncated*, if its current position is set to a value less than its length.

SetModify(f) *

A call to *SetModify* sets the file into state *modifying*. This implies that a buffer is assigned to the file and the byte at the current position is read into the buffer. The length of the file might hereby be increased but never decreased!

Doio(f) *

Not implemented.

* *The implementations of these procedures vary greatly between machines. Please see the release notes for implementation specifics.*

# 6.2. Module Program

**Svend Erik Knudsen 15.5.82**

## 6.2.1. Introduction

A Modula-2 program consists of a *main* module and of all separate modules imported directly or indirectly by the main module. Module *Program* provides facilities needed for the execution of Modula-2 programs upon Medos-2. The definition module is given in chapter 6.2.2. The program concept and explanations needed for the activation of a program are given in chapter 6.2.3. The *heap* and two routines handling the heap are explained in chapter 6.2.4. Possible error messages are listed in 6.2.5. The object file format may be inspected in 6.2.6.

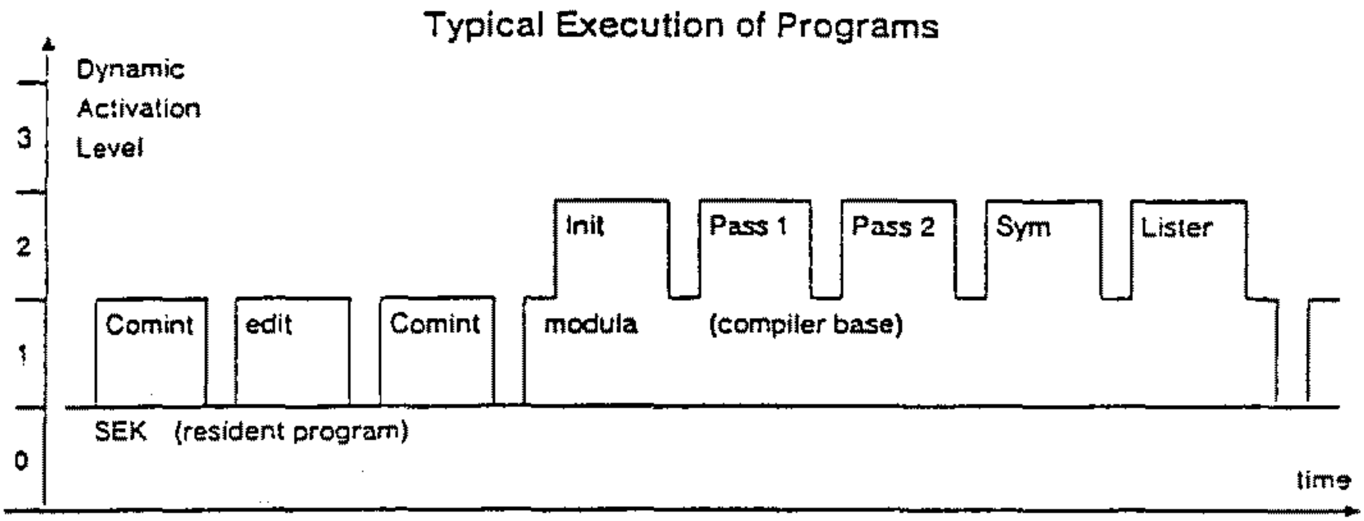
## 6.2.2 Definition Module Program

```
DEFINITION MODULE Program;      (* Medos-2 V3  S. E. Knudsen  1.6.81 *)

  FROM SYSTEM IMPORT ADDRESS;

  EXPORT QUALIFIED
    Call, Terminate, Status,
    MainProcess,
    CurrentLevel, SharedLevel,
    AllocateHeap, DeallocateHeap;

  TYPE
    Status = (normal,
              instructionerr, priorityerr, spaceerr, rangeerr, addressoverflow
              realoverflow, cardinaloverflow, integeroverflow, functionerr,
              halted, asserted, warned, stopped,
              callerr,
              programnotfound, programalreadyloaded, modulenotfound,
              codekeyerr, incompatiblemodule, maxspaceerr, maxmoduleerr,
              filestructureerr, fileerr,
              loaderr);

  PROCEDURE Call(programname: ARRAY OF CHAR; shared: BOOLEAN; VAR st: Status);
  PROCEDURE Terminate(st: Status);

  PROCEDURE MainProcess(): BOOLEAN;
  PROCEDURE CurrentLevel(): CARDINAL;
  PROCEDURE SharedLevel(): CARDINAL;

  PROCEDURE AllocateHeap(quantum: CARDINAL): ADDRESS;
  PROCEDURE DeallocateHeap(quantum: CARDINAL): ADDRESS;

END Program.
```

## 6.2.3. Execution of Programs

A Modula *program* consists of a *main* module and all separate modules imported directly and/or indirectly by the main module. Within Medos-2, any *running* program may activate another program just like a call of a procedure. The calling program is suspended while the called program is running, and it is resumed, when the called program terminates.

All active programs form a stack of activated programs. The first program in the stack is the resident part of the operating system, i.e. the (resident part of the) command interpreter together with all imported modules. The topmost program in the stack is the currently running program.




The figure illustrates, how programs may be activated. At any time, the *dynamic activation level* or simply the *level* identifies the active program in the stack.

Some essential differences exist, however, between programs and procedure activations.

A program is identified by a computable *program name*.

The calling program is resumed, when a program terminates (exception handling).

Resources like memory and connected files are owned by programs and are retrieved again, when the owning program terminates (resource management).

At any given time only one instance of a program can be active (programs are *not* reentrant).

The code for a program is *loaded*, when the program is activated and is removed, when the program terminates.

A program is activated by a call to procedure *Call*. Whenever a program is activated, its main module is loaded from a file. All directly or indirectly imported modules are also loaded from files, if they are not used by already active programs i.e. if they are not already loaded. In the latter case, the just called program is *bound* to the already loaded modules. This is analogous to nested procedures where the scope rules guarantee that objects declared in an enclosing block may be accessed from an inner procedure.

After the execution of a program, all its resources are returned. The modules, which were loaded, when the program was activated, are removed again.

The calling program may, by a parameter to *Call*, specify that the called program shares resources with the calling program. This means, that all sharable resources allocated by the called program are actually owned by the active program on the deepest activation level, which still shares resources with the currently running program. The most common resources, namely dynamically allocated memory space (from the heap) and (connected) files, are shared. Any feature implemented by use of procedure variables can essentially not be sharable, since the code for an assigned routine may be removed, when the program containing it terminates.

A program is identified by a *program name*, which consists of an identifier or a sequence of identifiers separated by periods. (see implementation notes for specifics)

```
Program name    = Identifier { "." Identifier } .
Identifier      = Letter { Letter | Digit } .
```

In order to find the *object code file*, from which a program must be loaded, the program name is converted into a file name as follows: The extension *.OBJ* is appended after the program name. If no such file exists, prefix *SYS.* is inserted, and a second search is carried out.

An object code file may contain the object code of several separate modules. Imported but not already loaded modules are searched sequentially on the object code file, which the loader is just reading.

Missing object code to imported modules is searched for like programs. The module name is converted to a file name by appending the extension *.OBJ* to it. If the file is not found, a second search is made after the prefix *LIB.* has been inserted. If the object code file is not yet found, the object code file for another missing module is searched. This is tried once for all imported and still not loaded modules.

| | |
|---|---|
| Program name | time |
| First searched file | time.OBJ |
| Second searched file | SYS.time.OBJ |
| | |
| Module name | Storage |
| First searched file | Storage.OBJ |
| Second searched file | LIB.Storage.OBJ |

Call(programname, shared, status)

Procedure *Call* loads and starts the execution of program *programname*. If *shared* is TRUE, the called program shares (sharable) resources with the calling program. The *status* indicates if a program was executed successfully.

| | |
|---|---|
| status = normal | Program executed normally |
| status in {instructionerr .. stopped} | Some execution error detected |
| status in {callerr .. loaderr} | Some load error detected |

Terminate(status)

The execution of a program may be terminated by a call to *Terminate*. The *status* given as parameter to *Terminate* is returned as status to the calling program.

CurrentLevel(): CARDINAL

Function *CurrentLevel* returns the (dynamic activation) *level* of the running program.

SharedLevel(): CARDINAL

Function *SharedLevel* returns the *level* of the lowest program, which shares resources with the current program.

MainProcess(): BOOLEAN

Function *MainProcess* returns TRUE if the currently executed coroutine (Modula-2 PROCESS) is the one which executes the initialisation part of the main module in the running program.

*Implementation Notes*

The current implementation of procedure *Call* may only be called from the *main* coroutine, i.e. the coroutine within which function *MainProcess* returns TRUE.

The module *Storage* may be loaded several times by module *Program*. This is the only exception to the rule, that a module may be loaded only once. Module *Storage* may be loaded once for each set of shared programs (i.e. once for each heap).

Only up to 96 modules may be loaded at any time. The resident part of Medos-2 consists of 6 modules.

The loader can handle up to 40 already imported but not yet loaded modules.

The maximum number of active programs is 16.

*Related Program*

The program *link* collects the object code from several separate modules onto one single object code file. *link* enables the user to substitute interactively an object code file with a non-default file name. "Linked" object code files might also be loaded faster and be more robust against changes and errors in the environment.

Example: Command Interpreter

```
MODULE Comint;        (* SEK  15.5.82 *)

  FROM Terminal IMPORT Write, WriteString, WriteLn;
  FROM Program IMPORT Call, Status;

  CONST
    programnamelength = 16;  (* This number will vary depending on the impleme

  VAR
    programname: ARRAY [0..programnamelength-1] OF CHAR;
    st: Status;

BEGIN
  LOOP
    Write('*');
    (* read programname *)
    Call(programname, TRUE, st);
    IF st <> normal THEN
      WriteLn;
      WriteString("- some error occured"); WriteLn
    END
  END (* LOOP *)
END Comint.
```

## 6.2.4. Heap

The main memory under the interpreter is divided into two parts, a stack and a heap. The stack grows from address 0 towards the *stack limit*, and the heap area is allocated between the stack limit and the highest address of the machine (64k-1). The stack and the heap are separated by the stack limit.

The area between the actual *top of stack* and the stack limit is free and may be allocated for both the stack and the heap.

Module *Program* handles the heap simply as a "reverse" stack, which may be enlarged by decrementing the stack limit address or reduced by incrementing it. This may be achieved by the routines *AllocateHeap* and *DeallocateHeap*.

Whenever a program is called, an *activation record* for that program is pushed onto the stack. Currently the activation record contains beside the "working stack" *(main process)* also the code and data for all modules loaded for the called program. The activation record of the running program is limited at the high end by top of stack.

If the call is a *shared* call, i.e. if the parameter *shared* of procedure *Call* is set TRUE, nothing specially is made with the heap: The heap may grow and shrink as if no new program had been activated. If the call is *not shared*, however, (parameter *shared* set to FALSE) the current value of stack limit is saved, and a new heap is created for the program on the top of the previous heap, i.e. at stack limit.

When a program terminates, its activation record is popped from the stack, and if the program is not shared with its calling program, its heap is released as well.

AllocateHeap(quantum): ADDRESS

Function *AllocateHeap* allocates an area to the heap by decrementing *stack limit* by *MIN(available space, quantum)*. The resulting stack limit is returned.

DeallocateHeap(quantum): ADDRESS

Function *DeallocateHeap* deallocates an area in the heap by incrementing *stack limit* by *MIN(size of heap, quantum)*. The resulting stack limit is returned.

*Implementation Note*

The current implementation of the functions *AllocateHeap* and *DeallocateHeap* may only be called from the *main* coroutine, i.e. the coroutine, within which function *MainProcess* returns TRUE.

*Related Module*

Module *Storage* is normally used for the allocation and deallocation of variables referenced by pointers. It maintains a list of free areas in the heap.

Examples: Procedures ALLOCATE and DEALLOCATE

```
PROCEDURE ALLOCATE(VAR addr: ADDRESS; size: CARDINAL);
  VAR top: ADDRESS;
BEGIN
  top := AllocateHeap(0);  (* current stack limit *)
  addr := AllocateHeap(size);
  IF top - addr < size THEN
    top := DeallocateHeap(top - addr);
    WriteString("- Heap overflow"); WriteLn;
    Terminate(spaceerr)
  END
END ALLOCATE;

PROCEDURE DEALLOCATE(VAR addr: ADDRESS; size: CARDINAL);
BEGIN
  addr := NIL
END DEALLOCATE;
```

## 6.2.5. Error Handling

All detected errors are normally handled by returning an error-indicating *Status* to the caller of procedure *Call*. Some errors detected by the loader are also displayed on the screen in order to give the user more detailed information. This is done according to the following format:

- `Program.Call:` *error indicating text*

The number of hyphens at the beginning of the message indicates the level of the called program.

`- Program.Call:  incompatible module`
`'`*module name*`' on file '`*file name*`'`

Imported module *module name* found on file *file name* has an unexpected module key.

```
- Program.Call:  incompatible module
    'module1 name' imported by 'module2 name' on file 'file name'
```

Module *module1 name* imported by *module2 name* on file *file name* has another key as the already loaded (or imported but not yet loaded) module with the same name.

```
- Program.Call:  module(s) not found:
  module1 name
  module2 name
  .
  .
```

The listed modules were not found.

## 6.2.6. Object Code Format

The format of the object code file generally has the following syntax:

```
LoadFile        = { Frame }.
Frame           = FrameType FrameSize { FrameWord }.
FrameType       = "200B" | "201B" | .... | "377B".
FrameSize       = Number.  /number of FrameWords/
FrameWord       = Number.
```

The load file is a word file. *FrameType* and *Number* are each represented in one word.

The object code file obeys a syntactic structure, called *ObjectFile*.

```
ObjectFile      = Module { Module } .
Module          = [ VersionFrame ] HeaderFrame [ ImportFrame ]
                  { ModuleCode | DataFrame }.
VersionFrame    = VERSION FrameSize VersionNumber.
FrameSize       = Number.
VersionNumber   = Number.
HeaderFrame     = MODULE FrameSize ModuleName DataSize.
ModuleName      = ModuleIdent ModuleKey.
ModuleIdent     = Letter { Letter | Digit } { "0C" }.
ModuleKey       = Number Number Number.
DataSize        = Number. /in words/
ImportFrame     = IMPORT FrameSize {ModuleName}.
ModuleCode      = CodeFrame [ FixupFrame ].
CodeFrame       = CODETEXT FrameSize WordOffset { CodeWord }.
WordOffset      = Number. /in words from the beginning of the module/
CodeWord        = Number.
FixupFrame      = FIXUP FrameSize {ByteOffset}.
ByteOffset      = Number. /in bytes from the beginning of the module/
DataFrame       = DATATEXT FrameSize WordOffset { DataWord }.
DataWord        = Number.
VERSION         = "200B".
MODULE          = "201B".
IMPORT          = "202B".
CODETEXT        = "203B".
DATATEXT        = "204B".
FIXUP           = "205B"
```

Currently the *VersionNumber* is equal to 3.

The *ByteOffsets* in *FixupFrame* point to bytes in the code containing *local* module numbers. The local module numbers must be replaced by the *actual* numbers of the corresponding modules. Local module number 0 stands for the module itself, local module number i (i > 0) stands for the i'th module in the *ImportFrame*.

A program is activated by a call to procedure 0 of its main module.

## 6.3. Storage

Svend Erik Knudsen 15.5.82

Calls to the Modula-2 standard procedures NEW and DISPOSE are translated into calls to ALLOCATE and DEALLOCATE. The standard way of doing this is to import ALLOCATE and/or DEALLOCATE from module Storage.

```
DEFINITION MODULE Storage;   (* Medos-2 V3  1.6.81  S. E. Knudsen *)

  FROM SYSTEM IMPORT ADDRESS;

  EXPORT QUALIFIED ALLOCATE, DEALLOCATE, Available;

  PROCEDURE ALLOCATE(VAR a: ADDRESS; size: CARDINAL);
  PROCEDURE DEALLOCATE(VAR a: ADDRESS; size: CARDINAL);
  PROCEDURE Available(size: CARDINAL): BOOLEAN;

END Storage.
```

*Explanations*

ALLOCATE(addr, size)

Procedure *ALLOCATE* allocates an area of the given size and assigns its address to *addr*. If no space is available, the calling program is killed.

DEALLOCATE(addr, size)

Procedure *DEALLOCATE* frees the area with the given size at address *addr*.

Available(size): BOOLEAN

Function *Available* returns TRUE if an area of the given size is available.

*Example*

```
MODULE StorageDemo;      (* SEK  15.5.82 *)

  FROM Storage IMPORT ALLOCATE;

  TYPE
    Pointer = POINTER TO Element;
    Element = RECORD next: Pointer; value: INTEGER END;

  VAR  root: Pointer;

  PROCEDURE NewInteger(i: INTEGER);
    VAR p: Pointer;
  BEGIN
    NEW(p);    (* implicit call to ALLOCATE *)
    p↑.next := root; p↑.value := i;
    root := p
  END NewInteger;

BEGIN
  root := NIL;
  (* ... *)
END StorageDemo.
```

*Restrictions*

The behaviour of the given implementation is only defined, if its procedures are (directly or indirectly) activated by the main program (and not from one of its coroutines).

*DEALLOCATE* checks only roughly the validity of the call.

Module *Storage* can only handle the heap for the running program. Other heaps created for programs not sharing the heap with the running program can not be handled by module *Storage* (see module *Program*, chapter 6.2.).

*Loading of Module Storage*

Module *Storage* may be loaded once for each heap it should handle. For more details see module *Program*, chapter 6.2.

*Error Messages*

```
- Storage.ALLOCATE: heap overflow
- Storage.DEALLOCATE: bad pointer
```

*Imported Modules*

*SYSTEM*
*Program*
*Terminal*

*Algorithms*

Procedure*Storage* maintains a list of available areas sorted by addresses in the heap. When an element has to be allocated, the list is searched from the highest towards lower addresses for a large enough available area. If such an area is found, the needed memory space is allocated in that area (first fit algorithm). Otherwise *Storage* tries to get more memory space allocated from module *Program (Program.AllocateHeap)*.

Procedure *DEALLOCATE* inserts the deallocated area into the sorted list of available areas. Adjacent available areas are collapsed during the insertion.

## 6.4. Terminal

Svend Erik Knudsen 15.5.82

Module *Terminal* provides the routines normally used for reading from the keyboard (or a commandfile) and for the sequential writing of text on the screen.

```
DEFINITION MODULE Terminal;        (* Medos-2 V3  S. E. Knudsen  1.6.81 *)

  EXPORT QUALIFIED
    Read, BusyRead, ReadAgain,
    Write, WriteString, WriteLn;

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

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

END Terminal.
```

*Explanations*

Read(ch)

Procedure *Read* gets the next character from the keyboard (or the commandfile) and assigns it to *ch*. Lines are terminated with character 36C (=*eol*, RS). The procedure *Read* does not "echoe" the read character on the screen.

BusyRead(ch)

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

Write(ch)

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

| | | |
|---|---|---|
| eol | 36C | Sets the writing position at the beginning of the next line |
| CR | 15C | Sets the writing position at the beginning of the current line |
| LF | 12C | Sets the writing position to the same column in the next line |
| FF | 14C | Clears the screen and sets the writing position into its upper left corner |
| BS | 10C | Sets the writing position one character backward |
| DEL | 177C | Sets the writing position one character backward and erases the character there |

WriteString(string)

Procedure *WriteString* writes out the given string. The string may be terminated with character 0C.

WriteLn

A call to procedure *WriteLn* is equivalent to the call *Write(eol)*.

# 7. Library Modules

**15.5.82**

This chapter is a collection of some commonly used library modules under the interpreter. For each library module a *symbol file* and an *object file* is stored on the distribution disk. The file names are derived from (the first 16 characters of) the module name, beginning with the prefix `LIB` and ending with the extension `SYM` for symbol files and the extension `OBJ` for object files. It is possible that some object files are pre-linked and therefore also contain the code of the imported modules.*

| | |
|---|---|
| Module name | `FileNames` |
| Symbol file name | `LIB.FileNames.SYM` |
| Object file name | `LIB.FileNames.OBJ` |

List of the Library Modules



The first two modules are considered to be used by small programs and for introductory exercises. They provide access to the terminal and to files by a simple interface.

* The 'LIB' prefix is ommitted on systems where the resident filesystem does not allow such filenames.

## 7.1. InOut

Niklaus Wirth 15.5.82

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

*Imported Library Modules*

Terminal
FileSystem

*Definition Module*

```
DEFINITION MODULE InOut;    (*NW 11.10.81*)
  FROM FileSystem IMPORT File;
  EXPORT QUALIFIED
    EOL, Done, in, out, termCH,
    OpenInput, OpenOutput, CloseInput, CloseOutput,
    Read, ReadString, ReadInt, ReadCard,
    Write, WriteLn, WriteString, WriteInt, WriteCard, WriteOct, WriteHex;

  CONST EOL = 36C;
  VAR Done:  BOOLEAN;
    termCH:  CHAR;
    in, out: File;

  PROCEDURE OpenInput(defext: ARRAY OF CHAR);
    (*request a file name and open input file "in".
      Done := "file was successfully opened".
      If open, subsequent input is read from this file.
      If name ends with ".", append extension defext*)

  PROCEDURE OpenOutput(defext: ARRAY OF CHAR);
    (*request a file name and open output file "out"
      Done := "file was successfully opened.
      If open, subsequent output is written on this file*)

  PROCEDURE CloseInput;
    (*closes input file; returns input to terminal*)

  PROCEDURE CloseOutput;
    (*closes output file; returns output to terminal*)

  PROCEDURE Read(VAR ch: CHAR);
    (*Done := NOT in.eof*)

  PROCEDURE ReadString(VAR s: ARRAY OF CHAR);
    (*read string, i.e. sequence of characters not containing
      blanks nor control characters; leading blanks are ignored.
      Input is terminated by any character <= " ";
      this character is assigned to termCH.
      DEL is used for backspacing when input from terminal*)

  PROCEDURE ReadInt(VAR x: INTEGER);
    (*read string and convert to integer. Syntax:
        integer = ["+"|"-"] digit {digit}.
      Leading blanks are ignored.
```

```
    Done := "integer was read"*)

  PROCEDURE ReadCard(VAR x: CARDINAL);
   (*read string and convert to cardinal. Syntax:
      cardinal = digit {digit}.
    Leading blanks are ignored.
    Done := "cardinal was read"*)

  PROCEDURE Write(ch: CHAR);

  PROCEDURE WriteLn;    (*terminate line*)

  PROCEDURE WriteString(s: ARRAY OF CHAR);

  PROCEDURE WriteInt(x: INTEGER; n: CARDINAL);
   (*write integer x with (at least) n characters on file "out".
    If n is greater than the number of digits needed,
    blanks are added preceding the number*)

  PROCEDURE WriteCard(x,n: CARDINAL);
  PROCEDURE WriteOct(x,n: CARDINAL);
  PROCEDURE WriteHex(x,n: CARDINAL);
END InOut.
```

## 7.2. RealInOut

Niklaus Wirth 15.5.82

Library module for formatted input/output of real numbers on terminal or files. It works together with the module InOut. A description of this module is included in Programming in Modula-2 [1].

*Imported Library Module*

InOut

*Definition Module*

```
DEFINITION MODULE RealInOut;   (*N.Wirth  16.8.81*)
  EXPORT QUALIFIED ReadReal, WriteReal, WriteRealOct, Done;

  VAR Done: BOOLEAN;

  PROCEDURE ReadReal(VAR x: REAL);
  (*Read REAL number x from keyboard according to syntax:

    ["+"|"-"] digit {digit} ["." digit {digit}] ["E"["+"|"-"] digit [digit]]

    Done := "a number was read".
    At most 7 digits are significant, leading zeros not
    counting.  Maximum exponent is 38. Input terminates
    with a blank or any control character.  DEL is used
    for backspacing*)

  PROCEDURE WriteReal(x: REAL; n: CARDINAL);
  (*Write x using n characters. If fewer than n characters
    are needed, leading blanks are inserted*)

  PROCEDURE WriteRealOct(x: REAL);
  (*Write x in octal form with exponent and mantissa*)

END RealInOut.
```

## 7.3. MathLib0

Niklaus Wirth 15.5.82

Library module providing some basic mathematical functions. A description of this module is included in Programming in Modula-2 [1].

*Imported Library Module*

Terminal

*Definition Module*

```
DEFINITION MODULE MathLib0;
  (*standard functions;  J.Waldvogel/N.Wirth,  10.12.80*)

  EXPORT QUALIFIED sqrt, exp, ln, sin, cos, arctan, real, entier;

  PROCEDURE sqrt(x: REAL): REAL;
  PROCEDURE exp(x: REAL): REAL;
  PROCEDURE ln(x: REAL): REAL;
  PROCEDURE sin(x: REAL): REAL;
  PROCEDURE cos(x: REAL): REAL;
  PROCEDURE arctan(x: REAL): REAL;
  PROCEDURE real(x: INTEGER): REAL;
  PROCEDURE entier(x: REAL): INTEGER;
END MathLib0.
```

## 7.4. OutTerminal

Christian Jacobi 15.5.82

This module contains a small collection of output conversion routines for numbers and strings. The output is written to the terminal.

Procedures:

| | |
|---|---|
| Write | writes a character |
| WriteLn | writes an end of line |
| WriteT | writes a string (T=text) |
| WriteI | writes an integer |
| WriteC | writes a cardinal |
| WriteO | writes octal |

length 0: one leading blank
<>0: no leading blank, the output is right adjusted in a field of "length" characters; if the field is too small its size is augmented.
WriteT does left adjustment and has no leading blanks

**Definition Module**

```
DEFINITION MODULE OutTerminal; (* Ch. Jacobi, S.E. Knudsen 18.8.80 *)
  FROM SYSTEM IMPORT WORD;
  EXPORT QUALIFIED
    Write, WriteLn, WriteT,
    WriteI, WriteC, WriteO;
  PROCEDURE Write(ch: CHAR);
  PROCEDURE WriteLn;
  PROCEDURE WriteT(s: ARRAY OF CHAR; length: CARDINAL);
  PROCEDURE WriteI(value: INTEGER; length: CARDINAL);
  PROCEDURE WriteC(value: CARDINAL; length: CARDINAL);
  PROCEDURE WriteO(value: WORD; length: CARDINAL);
END OutTerminal.
```

**Imported Module**

Terminal

## 7.5. OutFile

Christian Jacobi 15.5.82

This module contains a small collection of output conversion routines for numbers and strings to a file.

The procedures have different names than the corresponding procedure of the module OutTerminal. This simplifies combined imports of the module OutFile with one of the other formatting modules.

Procedures for formatted output onto the files:

| | |
|---|---|
| WriteChar | writes a character |
| WriteLine | writes an end of line |
| WriteText | writes a string |
| WriteInt | writes an integer |
| WriteCard | writes a cardinal |
| WriteOct | writes octal |

length 0: one leading blank
<>0: no leading blank, the output is right adjusted in a field of "length" characters; if the field is too small its size is augmented.
WriteText does left adjustment and has no leading blanks

**Definition Module**

```
DEFINITION MODULE OutFile; (* Ch. Jacobi, S.E. Knudsen 18.8.80 *)
  FROM SYSTEM IMPORT WORD;
  FROM FileSystem IMPORT File;
  EXPORT QUALIFIED
    WriteChar, WriteLine, WriteText,
    WriteInt, WriteCard, WriteOct;
  PROCEDURE WriteChar(VAR f: File; ch: CHAR);
  PROCEDURE WriteLine(VAR f: File);
  PROCEDURE WriteText(VAR f: File; s: ARRAY OF CHAR; length: CARDINAL);
  PROCEDURE WriteInt(VAR f: File; value: INTEGER; length: CARDINAL);
  PROCEDURE WriteCard(VAR f: File; value: CARDINAL; length: CARDINAL);
  PROCEDURE WriteOct(VAR f: File; value: WORD; length: CARDINAL);
END OutFile.
```

**Imported Module**

FileSystem

## 7.6. ByteIO

Svend Erik Knudsen 15.5.82

Module *ByteIO* provides routines for reading and writing bytes on files. This is valuable for the packing of information on files, if it is known that the ordinal values of the transferred elements are in the range 0..255.

```
DEFINITION MODULE ByteIO;     (* Medos-2 V3  S. E. Knudsen  1.6.81 *)

  FROM FileSystem IMPORT File;
  FROM SYSTEM IMPORT WORD;

  EXPORT QUALIFIED ReadByte, WriteByte;

  PROCEDURE ReadByte(VAR f: File; VAR w: WORD);
  PROCEDURE WriteByte(VAR f: File; w: WORD);

END ByteIO.
```

*Explanations*

ReadByte(f, w)

Procedure *ReadByte* reads a byte from file f and assigns its value to w, i.e. 0 <= ORD(w) <= 255.

WriteByte(f, w)

Procedure *WriteByte* writes the low order byte of w (bits 8..15) on file f.

*Example*

```
MODULE ByteIODemo;      (* SEK  15.5.82 *)

  FROM FileSystem IMPORT File, Lookup, Close;
  FROM ByteIO IMPORT ReadByte, WriteByte;

  VAR
    inf, outf: File;
    byte: CARDINAL;

BEGIN
  Lookup(inf, 'Demo.from', FALSE);
  Lookup(outf, 'Demo.to', TRUE);
  LOOP
    ReadByte(inf,byte);
    IF inf.eof THEN EXIT END;
    WriteByte(outf, byte);
  END;
  Close(outf);
  Close(inf)
END ByteIODemo.
```

*Imported Modules*

SYSTEM
FileSystem

## 7.7. ByteBlockIO

Svend Erik Knudsen 15.5.82

Module *ByteBlockIO* provides routines for efficient reading and writing of elements of any type on files. Areas, given by their address and size in bytes, may be transferred efficiently as well.

```
DEFINITION MODULE ByteBlockIO;       (* Medos-2 V3  S. E. Knudsen  1.6.81 *)

  FROM FileSystem IMPORT File;
  FROM SYSTEM IMPORT WORD, ADDRESS;

  EXPORT QUALIFIED
    ReadByteBlock, WriteByteBlock,
    ReadBytes, WriteBytes;

  PROCEDURE ReadByteBlock(VAR f: File; VAR block: ARRAY OF WORD);
  PROCEDURE WriteByteBlock(VAR f: File; VAR block: ARRAY OF WORD);

  PROCEDURE ReadBytes(VAR f: File; addr: ADDRESS; count: CARDINAL;
                      VAR actualcount: CARDINAL);
  PROCEDURE WriteBytes(VAR f: File; addr: ADDRESS; count: CARDINAL);

END ByteBlockIO.
```

*Explanations*

ReadByteBlock(f, block); WriteByteblock(f, block)

*ReadByteBlock* and *WriteByteBlock* transfer the given block (ARRAY OF WORD) to or from file *f*. The bytes are transferred according to the description given for *ReadBytes* and *WriteBytes*.

ReadBytes(f, addr, count, actualcount); WriteBytes(f, addr, count)

*ReadBytes* and *WriteBytes* transfer the given area (beginning at address *addr* and with *count* bytes (stored in (count+1) DIV 2 words) to or from the file f. The number of the actually read bytes is assigned to *actualcount*. *ReadBytes* and *WriteBytes* transfer two bytes to or from each word; first the high order byte (bits 0..7), afterwards the low order byte (bits 8..15). If *actualcount* is odd, only the high order byte is transferred to or from the last word.

*Example*

```
MODULE ByteBlockIODemo;       (* SEK  15.5.82 *)

  FROM FileSystem IMPORT File, Response, Lookup, Close;
  FROM ByteBlockIO IMPORT ReadByteBlock;

  VAR r: RECORD (*...*) END;
      f: File;

BEGIN
  Lookup(f, 'Demo', FALSE);
  IF f.res = done THEN
    LOOP
      ReadByteBlock(f, r);
      IF f.eof THEN EXIT END;
      (* use r *)
    END;
    Close(f)
```

```
  ELSE (* file not found *)
  END
END ByteBlockIODemo.
```

*Restriction*

The longest block which can be transferred by a single call to *ReadByteBlock* or *WriteByteBlock* contains 2**15 - 1 words.

*Imported Modules*

SYSTEM
FileSystem

*Algorithm*

The routines repeatedly determinates the longest segment of bytes, which can be moved to or from the file buffer and move this segment by use of a CODE-procedures (MOV, LXB and SXB-instructions).

## 7.8. FileNames

Svend Erik Knudsen 15.5.82

Module *FileNames* makes it easier to read in file names from the keyboard (i.e. from module *Terminal*) and to handle defaults for such file names.

```
DEFINITION MODULE FileNames;    (* Medos-2 V3  S. E. Knudsen  1.6.81 *)

  EXPORT QUALIFIED
    ReadFileName, Identifiers, IdentifierPosition;

  PROCEDURE ReadFileName(VAR fn: ARRAY OF CHAR; dfn: ARRAY OF CHAR);

  PROCEDURE Identifiers(fn: ARRAY OF CHAR): CARDINAL;
  PROCEDURE IdentifierPosition(fn: ARRAY OF CHAR; identno: CARDINAL): CARDINAL;

END FileNames.
```

*Explanations*

ReadFileName(fn, dfn)

Procedure *ReadFileName* reads the file name *fn* according to the given default file name *dfn*. If no valid file name could be returned, fn[0] is set to 0C. The character typed in in order to terminate the file name, may be read after the call to *ReadFileName*. One of the characters eol, " ", "/", CAN and ESC terminates the input of a file name. If CAN or ESC has been typed, fn[0] is set 0C too.

Identifiers(filename)

Function *Identifiers* returns the number of identifiers in the given file name.

IdentifierPosition(filename, identifierno)

Function *IdentifierPosition* returns the index of the first character of the identifier *identifierno* in the given file name. The first identifier in the file name is given number 0. The length of a given file name *fn* is returned by the following function call: *IdentifierPosition(fn, Identifiers(fn))*.

*Syntax of the Different Names*

```
FileName            = [ LocalFileName ] [ 0C | " " ] .
LocalFileName       = [ QualIdentifier "." ] Extension .
QualIdentifier      = Identifier { "." Identifier } .
Extension           = Identifier .
Identifier          = WildcardLetter { Letter | Digit } .

DefaultFileName     = [ MediumName ] [ "." [ DefaultLocalName ] ] [ 0C | " " ] .
DefaultLocalName    = [ [ QualIdentifier ] "." ] Extension .

InputFileName       = [ "#" [ MediumName ] [ "." InputLocalName ] | InputLocalName ] .
InputLocalName      = [ QualInput "." ] Extension .
QualInput           = [ QualIdentifier [ "." ] ] [ "." QualIdentifier ] .
```

The scanning of the typed in *InputFileName* is terminated by the characters ESC and CAN or at a syntatically correct position by the characters *eol*, " " and "/". The termination character may be read after the call. For correction of typing errors, DEL is accepted at any place in the input. Typed in characters not fitting into the syntax are simply ignored and not echoed on the screen.

For routine *ReadFileName* a file name consists of a *medium name* part and of an optional *local file name* part. The local file name part consists of an extension and optionally of a sequence of identifiers delimited by periods before the extension.

When typing in an *InputFileName*, an omitted part in the *InputFileName* is substituted by the corresponding part in the given default file name whenever the part is needed for building a syntactically correct *FileName*. If the corresponding part in the default file name is empty, the part must be typed.

*Examples*

| | |
|---|---|
| `ReadFileName(fn, ".MOD")` | Defaults for medium name and extension |
| `ReadFileName(fn, "Temp.MOD")` | Defaults for all parts of a file name |

*Error Message*

```
ReadfileName called with incorrect default
```

*Imported Module*

Terminal

## 7.9. Options

Leo Geissmann 15.5.82

Library module for reading a ***file name*** followed by ***program options*** from the keyboard. File name and options are accepted according to the syntax given in 4.2.3. and 4.3.

*Imported Library Modules*

Terminal
FileNames

*Definition Module*

```
DEFINITION MODULE Options; (* AKG 28.05.80; LG 10.10.80 *)

  EXPORT QUALIFIED Termination, FileNameAndOptions, GetOption;

  TYPE Termination = (normal, empty, can, esc);

  PROCEDURE FileNameAndOptions(default: ARRAY OF CHAR; VAR name: ARRAY OF CHAR
                               VAR term: Termination; acceptOption: BOOLEAN);

  PROCEDURE GetOption(VAR optStr: ARRAY OF CHAR; VAR length: CARDINAL);

END Options.
```

Procedure ***FileNameAndOptions*** reads a file name and, if ***acceptOption*** is TRUE, options from the terminal. It reads all characters from terminal until one of the keys **RETURN**, **BLANK** (space-bar), **CTRL-X**, or **ESC** is typed. For the file name, a ***default*** file name may be proposed. The accepted name is returned with parameter ***name***, and ***term*** indicates, how the input was terminated. The meaning of the values of type ***Termination*** is

- **normal** input normally terminated
- **empty** input normally terminated, but name is empty
- **can** CTRL-X was typed, input line is cancelled
- **esc** ESC was typed, no file is specified.

Procedure ***GetOption*** may be called repeatedly after FileNameAndOptions to get the accepted options. It returns the next option string in ***optStr*** and its length in ***length***. The string is terminated with a 0C character, if length <= HIGH(optStr). Length gets the value 0, if no option is returned.

# 8. Modula-2 under the M-2 Interpreter

**Leo Geissmann 15.5.82**

**Revised Modula Research Institute 24.8.83**

Differences in programming under various implementations can be attributed to the following causes:

1. Extensions of the language proper, i.e. new syntactic constructs.

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

3. Differences in the internal representation of data.

4. Differences in the sets of available library 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 reflects directly an entire system's available resources in software as well as hardware. This chapter gives an overview of the M-2 Interpreter specific low-level features.

### WARNING

The following feature should be applied with **utmost care** since it is easy to introduce errors into the internal stack if not used properly.

## 8.1. Code Procedures

A code procedure is a declaration in which the procedure body has been replaced by a (sequence of) code number(s), representing machine instructions (see Lilith report [2]). Code procedures are a facility to make micro-coded routines available at the level of Modula-2.

This facility is reflected by the following extension to the syntax of the procedure declaration :

```
$ ProcedureDeclaration   = ProcedureHeading ";" (block | codeblock) ident.
$ codeblock              = CODE CodeSequence END .
$ CodeSequence           = code {";" code}.
$ code                   = [ConstExpression].
```

The following are typical examples of code procedure declarations:

```
PROCEDURE ShiftLeft(VAR num: CARDINAL; count: INTEGER);
  (* Shift 'num' left 'count' places *)
CODE 276B
END ShiftLeft

PROCEDURE ShiftRight(VAR num: CARDINAL; count: INTEGER);
  (* Shift 'num' right 'count' places *)
CODE 277B
END ShiftRight
```

Parameters of code procedures are written on the *expression stack* of the Lilith machine, where they must be read by the code instructions. The compiler does not check to insure that the parameters correspond to the instructions. **The responsibility is left to the programmer.**

## 8.2. The Module SYSTEM

The module *SYSTEM* offers additional tools for Modula-2. Most of them are implementation and/or processor dependent. Such tools are sometimes necessary for *low-level programming*. SYSTEM also contains types and procedures which allow very basic coroutine handling.

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

For more detailed information refer to *Programming in Modula-2* (see 1.3).

Objects Exported from Module SYSTEM

*Types*

`WORD`

Representation of an individually accessible storage unit (one word). No operations are allowed for variables of type `WORD`. A `WORD` parameter may be substituted by an actual parameter of any type that uses one word in storage.

`ADDRESS`

Word address of any location in the storage. The type `ADDRESS` is compatible with all pointer types and is itself defined as `POINTER TO WORD`. All integer arithmetic operators apply to this type.

`PROCESS`

Type used for process handling.

*Procedures*

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

Procedure to instantiate a new process. At least 50 words are needed for the workspace of a process.

`TRANSFER(VAR p1, p2: PROCESS)`

Transfer of control between two processes.

*Functions*

`ADR(variable): ADDRESS`

Storage address of the substituted variable.

`SIZE(variable): CARDINAL`

Number of words used by the substituted variable in the storage. If the variable is of a record type with variants, then the variant with maximal size is assumed.

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

Number of words used by a variable of the substituted type in the storage. If the type is a record with variants, then tag constants of the last *FieldList* (see Modula-2 syntax in [1]) may be substituted in their nesting order. If tag constants are not specified or are partially specified, then the remaining variant with maximal size is assumed.

## 8.3. Data Representation and Parameter Transfer

### 8.3.1. Data Representation

The basic memory unit for data is the word. One word contains 16 bits. Every word in data memory can be accessed explicitly. In the following list for each data type the number of words needed in memory and the representation of the values is indicated. The bits within a word are enumerated from left to right, i.e. the ordinal value 1 is represented by bit 15.

`INTEGER`

Integer variables are represented in one memory word. Minint = -32768 (octal INTEGER(100000B)); maxint = 32767 (octal 77777B). Bit 0 is the *sign bit*; bit 1 the *most significant bit.*

CARDINAL

Cardinal variables are represented in one memory word. Maxcard = 65535 (octal 177777B). Bit 0 is the *most significant bit.*

BOOLEAN

Boolean variables are represented in one memory word. This type must be considered as an enumeration (FALSE, TRUE) with the values FALSE = 0 and TRUE = 1 (bit 15). Other values may cause errors.

CHAR

Character variables are represented in one memory word. In arrays two characters are *packed* into one word. The ISO - ASCII character set is used with ordinal values in the range [0..256] (octal [0B..377B]). The compiler accepts character constants in the range [0C..377C].

REAL

Real variables are represented in two memory words (32 bits). Bit 0 of the first word is the *sign bit.* Bits 1..8 of the first word represent an *8-bit exponent* in *excess 128 notation.* Bits 9..15 of the first word represent the *high part of the mantissa* and the second word represents the *low part of the mantissa.* The mantissa is assumed to be normalized (0.5 <= mantissa < 1.0). The most significant bit of the mantissa is not stored (it is always 1).

Enumeration Types

Enumerations are represented in one memory word. The first value of the enumeration is represented by the integer value 0; the subsequent enumeration values get the subsequent integer values accordingly.

Subrange Types

Subranges are represented according to their base types.

Array Types

Arrays are usually accessed *indirectly.* A pointer to an array points to the first element of the array. In *character arrays* two characters are packed into one word. The first character is stored in the high order byte of the first word (bits 0..7), the second character in the low order byte (bits 8..15), etc.

Record Types

Records are usually accessed indirectly. A pointer to a record points to the first field of the record. Consecutive fields of a record get consecutive memory locations. Every field needs at least one word.

Set Types

Sets are implemented in one word. The set element i is represented in bit i, i.e. {15} corresponds to the ordinal value 1. INCL(s, i) means: bit i in s is set to the value 1.

Pointer Types

Pointers are represented in one memory word. They are implemented as absolute addresses. The pointer constant NIL is represented by the ordinal value 177777B.

Procedure Types

Procedure Types are represented in one memory word. The high order byte (bits 0..7) represents the module number, the low order byte (bits 8..15) the procedure number of the assigned procedure.

**Warning** *Do not use this information.*

Opaque Types

Opaque Types are represented in one memory word.

`WORD`

Word variables are represented in one memory word.

`ADDRESS`

Address variables are represented in one memory word. The value is an absolute address.

`PROCESS`

Process variables are represented in one memory word. The value is an absolute address pointing to a process descriptor.

### 8.3.2. Parameter Transfer

*Variable Parameters*

The address is transferred to the expression stack.

For *dynamic arrays* also the value `HIGH` is submitted to the expression stack. The push operation for the address is executed first.

*Value Parameters*

`Records and Arrays`

The address is transferred to the expression stack (regardless of size). The procedure allocates the memory space and copies the parameter.

For *dynamic arrays* the value `HIGH` is submitted to the expression stack. The push operation for the address is executed first.

`REAL`

The value itself is passed to the expression stack (two words). The procedure copies the value into its proper location.

`Other Types with One Word Size`

The value itself is passed to the expression stack. The procedure copies the value into its proper location.

# 9. Assembly Language Interface

**Rod Schiffman 22.11.83**
**Rod Riggs 19.12.83**

This chapter describes the assembly language interface for the M-2 Interpreter. It allows external programs to be written in 8088 assembly lanuage and to be called from Modula programs. This chapter will describe how the interface works, how to pass parameters between 8088 assembly and Modula and, finally, how the program linkage works at an assembly language level. Under normal circumstances, it is not necessary for a programmer to use the information in this chapter. It is provided as a service to experienced programmers who must access special features of the host operating system that are not supported by the interpreter. It can also be used if it is absolutely necessary that short sections of a Modula program must run in a more real-time environment than possible using only the interpreter.

### 9.1 General Description

As can be expected, the procedure calling conventions used between procedures written in Modula and the calling conventions between 8088 procedures are incompatible. Therefore, the interpreter provides special code that facilitates the linkages. This is done through the use of Modula *code* procedures. The Escape M-Code allows up to 256 different routines written in 8088 assembly to be linked into the interpreter and called by Modula programs. Before an assembly procedure can be called by a Modula program information about the procedure must be made available to the interpreter. This is done through a table that can be accessed by both the interpreter and an external program.

The interpreter is supplied in both a linked and executable image, as well as in an unlinked form that allows new procedures to be linked into the interpreter. When a new procedure is to be made available for use by a Modula program, there are two main steps to follow. First the procedure must be written and assembled. Then it must be bound into the interpreter. The binding is accomplished by making an entry into a table in the program *ASMLNK.ASM*, then assembling ASMLNK.ASM and linking all of the object files of the interpreter into a single executable program. This process is described in more detail below.

### 9.2 Implementation

It is possible, through the use of Code Procedures, to access various special purpose M-Codes that the compiler does not generate. These are described in section 8.1 of the manual. The Escape M-Code (246b) is the M-Code that provides the linkage to external programs. The Escape M-Code takes the next byte of code following it as an entry into a table that contains information about the assembly procedure that is to be executed. The table contains four entries. The first is the offset of the procedure in the code segment, the second is the code segment of the procedure. The third is the number of parameters and the fourth indicates whether the procedure is a function and returns a value. If the entry in the table is non-zero, the specified number of parameters are removed from the internal interpreter stack and placed on the machine stack. Upon return from the procedure, the returned value, if it exists, is placed on the internal interpreter stack and control is returned to the Modula program.

The table that contains the information about the procedure to be called is in the program *ASMLNK.ASM*. It has been supplied in source form, and contains an example procedure entry. The example procedure is called *TestLink.ASM* and is also supplied in source form. The table in ASMLNK.ASM is called *ESCTAB* and contains 256 entries. Each entry is formatted as follows:

```
DW   OFFSET testlink, SEG testlink, 1, 1
     ↑                ↑             ↑  ↑ ···· 1 = Function, 0 = Not a Function
     ↑                ↑             ↑ ···· Number of Parameters
     ↑                ↑ ···· Stores the value of testlink's Code Segment
     ↑ ···· Stores the OFFSET of testlink in its Code Segment
```

The maximum number of parameters is 16, and the function return value must fit into one 16 bit word.

### 9.3 Parameter Passing

Modula allows parameters to be passed by both value and by reference. A parameter passed by value can be modified without reflecting the changes in the original. This is the default method of parameter passing in Modula. A VAR in the formal parameter list declares a parameter that is passed by reference. When a reference parameter is modified, the changes may be reflected in the original. Generally, a *value* parameter is passed by placing a copy of the parameter on the stack, and a *reference* parameter is passed by placing a pointer to the original value on the stack. This is important to know when an assembly language procedure is receiving parameters from a Modula procedure.

Even though Modula has two different types of parameter passing, there are several ways different types of parameters are passed; i.e. an array is passed differently than a single parameter. Also, Modula allows unbounded arrays to be used as formal parameters, and they have additional information on the stack. Section 8.3 describes how each different Modula type is represented in memory, and it describes how parameters are passed. The important distinction to be made is the difference between dynamic arrays and types with known sizes. All types and variables with a known size can be passed without the size being passed, because the size is known at the compile time. All unbounded array types must pass a length with the actual value or pointer because the actual length is not known until run time. This value can be accessed in a Modula procedure through the standard procedure HIGH. It is also used by the virtual machine to know how to copy a value parameter with different lengths each time the procedure is called. Whenever an external procedure accepts an unbounded parameter like ARRAY OF CHAR or ARRAY OF WORD, it must also handle the length word that will be on the stack.

When the actual value for a parameter is passed the values on the interpreter stack are removed in reverse order and pushed onto the machine stack. This means that a parameter must fit into one word. Currently the only type requiring more than one word is REAL. If a REAL is to be passed as a parameter, it must be passed by reference and not by value. When an unbounded array is passed as a parameter the address of the array will be first and the length will be second. A character passed as a value parameter will be in the bottom 8 bits. The first parameter in the formal parameter list of a procedure declaration will be the first parameter on the stack when the assembly procedure receives control.

Problems with passing pointer parameters can be avoided if one carefully remembers the following: The M-Codes in the interpreter reside in a separate address space from the rest of the process, and Modula pointers and assembly pointers are different. Thus, Modula pointers must be mapped into the process address space before they are used. Since, the interpreter does not understand what parameters are values and what parameters are pointers this must be done by the 8088 procedure being called. This is done through the use of the procedure *VMAP* that is global to the interpeter. It accepts a Modula pointer from an 8088 procedure in AX and returns an 8088 pointer. The segment value will be returned in AX and the offset will be returned in BX. The following example should help make things clearer.

### 9.4 An Example

This section contains an example procedure written in assembly and called by a Modula program. The 8088 procedure we will use as an example will accept a character as a reference parameter and, if it is a lower case letter it will return TRUE and change the letter to upper case otherwise it will return FALSE.

The first step is to write the assembly procedure and verify that it works.

```
    EXTRN  VMAP:FAR                    ;Make VMAP accessible

WORKAREA SEGMENT BYTE PUBLIC 'DATA'
   ;The data segment MUST be named WORKAREA and be, PUBLIC,with class of 'DATA'

TOS   DW   ?                           ;Storage for the return address
NOS   DW   ?                           ;Storage for the return address
SVES  DW   ?                           ;Storage for interpreter's ES value
SDS   DW   ?                           ;Storage for interpreter's DS value
WORKAREA ENDS
;**** for 1 parameter, this is a function ***
UserSeg SEGMENT BYTE PUBLIC 'PROG'
   ;The code segment MUST be named UserSeg and be, PUBLIC,with class of 'PROG'

    ASSUME  CS:UserSeg,DS:WORKAREA

PUBLIC TestLink                        ;Must be public to be called externally
TestLink PROC  FAR                     ;Must be FAR since it is in a different code segment
    MOV   BX,DS                        ;Get the DS value for the interpreter
    MOV   AX,WORKAREA                  ;Get the DS value for this procedure
    MOV   DS,AX                        ;Store it into DS
    MOV   SDS,BX                       ;Store the old DS after current DS value has been loaded
    POP   CX                           ;Save the SEG and OFFSET values
    MOV   TOS,CX                       ;for the return to the interpreter.
    POP   CX
    MOV   NOS,CX
    POP   AX                           ;Get the first parameter off the stack
    CALL  VMAP                         ;It is a reference parameter, so get the pointer to it
                                       ;BX contains the offset of the parameter
                                       ;AX contains the SEG value
    MOV   SVES,ES                      ;Save the old interpreter ES value
    MOV   ES,AX                        ;Load the SEG value into ES
    MOV   AX,ES:[BX]                   ;Get the actual value of this parameter
    MOV   CX,00H                       ;Assume it's not lower case, return value of FALSE
    CMP   AX,141O                      ;Is it less than a lower case 'a'?
    JL    TCNT
    CMP   AL,172O                      ;Is it greater than a lower case 'z'?
    JG    TCNT
    SUB   AL,32                        ;Sub 32 to get Upper case letter
    MOV   ES:[BX],AX                   ;Store in the original variable
    MOV   CX,01H                       ;It is TRUE that we changed the value
TCNT:
    MOV   DX,NOS                       ;Restore the RETurn address
    PUSH  DX
    MOV   DX,TOS
    PUSH  DX
    MOV   AX,SVES                      ;Restore the interpreters ES value
    MOV   ES,AX
    MOV   AX,SDS                       ;Restore the interpreters DS value
    MOV   DS,AX
    MOV   AX,CX                        ;Functions return values in AX
    RET
TestLink    ENDP
UserSeg ENDS
```

The Data Segment for all assembly programs linked into the interpreter must be declared with the same name and parameters as in the example above. The code segment must also be exactly the same as the example.

Next the assembly procedure must be made available to the interpreter. This is done by putting an entry in ASMLNK.ASM.

```
DW     OFFSET testlink, SEG testlink, 1, 1
```

Next assemble ASMLNK.ASM and your assembly language procedure. When all necessary files have been assembled you must now re-link the interpreter. *LINKIN.BAT* has been provided. Simply type *LINKIN* followed by the list of .OBJ files to also be linked into the interpreter. There will be one warning error that there is no stack segment. This is expected since the interpreter has been written to share the data segment with the stack and therefore does not have a separate stack segment. The files that make up the core of the interpreter are INTERP, INTEXT, READBTFL, SYSTM, NEWSYS, FLOAT and ESCAPE. The .OBJ file for each of these has been provided. One additional file needed to use the linkage facility is ASMLNK. The source for this file has been provided. ASMLNK.ASM will need to be re-assembled each time a new assembly procedure is added.

Finally, write and compile a Modula-2 program to use the procedure. The procedure declaration and simple program are listed below.

```
MODULE TL;

FROM Terminal IMPORT Read, WriteLn, WriteString, Write;

VAR ch: CHAR;
  lowercase: BOOLEAN;

PROCEDURE TestLink(VAR ch: CHAR):BOOLEAN;
CODE 246B; 0
END TestLink;

BEGIN
 LOOP
  WriteLn;
  WriteString('Character> ');
  Read(ch); Write(ch);
  IF CAP(ch) = 'Q' THEN EXIT END;
  lowercase := TestLink(ch);
  IF lowercase THEN
   WriteLn;
   WriteString('Converted to : ');
   Write(ch)
  END;
 END;
END TL.
```

The program will loop until a 'q' is hit.